KYOTO UNIVERSITY

DOCTORAL THESIS

An Integrated Theory of Type-Based Static and Dynamic Verification

Author:
Taro Sekiyama

Supervisor:
Professor Atsushi Igarashi

Department of Communications and Computer Engineering
Graduate School of Informatics

February 26, 2016

KYOTO UNIVERSITY

Abstract

Graduate School of Informatics
Department of Communications and Computer Engineering

Doctor of Informatics

An Integrated Theory of Type-Based Static and Dynamic Verification

by Taro Sekiyama

For development of reliable software, many verification methods have been studied so far. One of the most successful approaches is type systems, which have been tied to various kinds of programming languages from dynamically typed ones through dependently typed ones. Each of dynamic, static, and dependent typing has its own pros and cons and is not always sufficient for development of practical, reliable software.

Our goal is to introduce a full-fledged programming language where dynamically, statically, and dependently typed code coexist and interact safely. In this thesis, we focus on three universal features in programming: delimited control, parametric polymorphism, and algebraic datatypes. These features are bases of current programming languages—delimited control provides control effects such as exception handling; polymorphism plays a key role in type-based abstraction and reuse of program components; algebraic datatypes are an usual tool to represent data structures. We study how these features are incorporated with mechanisms for integrating a certified and an uncertified worlds, based on gradual typing, which combines static and dynamic typing, and manifest contracts, which does static and dependent typing. We first study delimited control in integration of static and dynamic typing; this extension needs monitoring of capture and call of delimited continuations. We also investigate parametric polymorphism in a combination of static and dependent typing and show our extension is sound, in particular, parametricity does hold. Finally, an extension with algebraic datatypes lets us compare two major approaches to giving specifications to data structures from the point of view of computational efficiency. We believe that these extensions and insights obtained from them will contribute to achievement of our goal.

Acknowledgements

First of all, I want to thank Atsushi Igarashi for being my supervisor and a patient collaborator. He gave me a chance to study manifest contracts, which I have worked over in both of my master's and doctoral courses. He also gave many fruitful, influential comments to my work—especially, the work on manifest contracts with algebraic datatypes started with his insight to the relationship between refinements on type constructors and data constructors. My attitude to research has been affected deeply by him. He was patient with my English writing, speaking, and listening skills and helped my job hunting.

I am grateful to Akihiro Yamamoto and Yasuo Okabe for the attention of my committee. Their comments improve the quality of my thesis significantly.

Koji Nakazawa educated me about mathematics, logic, and mathematical logic. Kohei Suenaga gave advice on job hunting. They also gave comments which improved my work and presentations significantly. Kentaro Okumura kept on discussing work and counseled me when I was in difficulty. Yuki Nishida developed an experimental implementation to demonstrate our work. I am very grateful to all members of the Computer Software Group at Kyoto University. I enjoyed life in the laboratory thanks to them.

Michael Greenberg is a collaborator on manifest contracts. The discussion with him was exciting and the content of Chapter 3 is the joint work with him. He also improved my presentation and encouraged me in the presentation at an international conference. Yoshiyuki Kameyama and Kenichi Asai advised and indicated a direction of my work on logical relations for statically typed languages with shift/reset (that work is not presented in this thesis, though). Kenichi Asai also taught me that a study of shift/reset should be based on CPS transformation; in fact, the transformation works as a guide to introduce the monitoring system given in Chapter 2.

Finally, I appreciate my family, who have supported me from both of financial and mental sides since childhood. Without their helps, I would never be here.

List of Publications

Chapter 2 consists of:

Taro Sekiyama, Soichiro Ueda, and Atsushi Igarashi. Shifting the blame - A blame calculus with delimited control. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems*, volume 9458 of *Lecture Notes in Computer Science*, pages 189–207. Springer-Verlag, 2015.

Chapter 3 consists of:

Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. Polymorphic manifest contracts, revised and resolved. *ACM Transactions on Programming Languages and Systems*, 2015. Accepted with major revision.

Chapter 4 consists of:

Taro Sekiyama, Yuki Nishida, and Atsushi Igarashi. Manifest contracts for datatypes. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 195–207, 2015.

Taro Sekiyama, Yuki Nishida, and Atsushi Igarashi. Manifest contracts for algebraic datatypes. *The 16th Workshop on Programming and Programming Languages*, 2014.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software Verification

Software has been used in various situations and dealt with critical things such as human life, business, privacy, and so on. In such software, it is crucial to verify that the software behaves *correctly*. For example, an operating system should provide guest users with only data offered publicly and, more technically, programs should write and read data to and from a file via an opened file descriptor.

The most obvious way to check the behavior of a program is probably to execute it actually and observe what it outputs and what computational resources (e.g., files, sockets, and processes) it uses. This naive approach, however, has many defects in verifying practical software. First, when something unexpected is observed, it is often difficult to identify the cause of the error because software bugs and their causes often seem to be unrelated in program texts in many cases. Worse, that approach does not reach the goal completely, namely, a program verified by it does not always behave correctly because it checks only *some* execution paths of the program, not *all*. For example, let us consider an (artificial) Ruby program as follows:

```
def f(b) =
  if b then
    1
  else
    "foo"
  end
end

x = f(File.exists?("bar"))
print Math.sqrt(x)
```

The function `f` takes a Boolean value and examines its truth; if the argument is `true`, `f` will return an integer; otherwise, it will return a string. The program queries whether the file named `"bar"` exists, passes its result to function `f`, and prints the square root of the value returned by `f`. Obviously, a run of this program would never raise errors if the file `"bar"` exists since, in that case, `f` will return integer `1` and its square root is also `1`. This program will be aborted, however, in the case that the file does not exist, because `f` will return a string but `Math.sqrt` cannot calculate the square root of a string value. This is the case that *all* execution paths in a program should be checked. Unfortunately, we cannot expect it in general because such checks give rise to combinatorial explosion easily and, worse, there may be infinite execution paths, in particular, in nonterminating programs.

Alternatively, *static verification* has been studied and used as approaches to verifying programs in a more systematic, modular, and dependable way. Static verification

techniques give program components *specifications*, which represent how the components should behave, and check that the components follow the specifications with the aid of computers before running the program. Since it does not need execution of programs, static verification can detect errors in an early stage of development—the early error detection reduces efforts made by programmers to find software bugs from a large codebase. Most approaches to static verification are also exhaustive, i.e., if they accept a program, it is guaranteed that the program never has certain errors. Forms of specifications vary with properties we want programs to have and verification approaches.

1.2 Type-Based Software Verification

Types, which intuitively denote kinds of data, are one of the most successful specification forms. Verification mechanisms using types are usually called *type systems*, which assign types to program components and check that the components are manipulated with only operations allowed by their types. Many programming languages such as Java [42], C++ [56], C♯ [73], Standard ML [76], Haskell [68], etc. support static type systems which guarantee that programs accepted by the systems do not have some kinds of errors—e.g., "method-not-found" errors in Java and call of nonfunctional values in Standard ML and Haskell. The static discrimination among data makes it possible to not only detect errors caused by applying unexpected operations at very early stages but also optimize programs (in fact, this is the first motivation of introducing the type system in FORTRAN [12]). Another benefit of type systems is that types are specifications (comparatively) easy to read and understand and so they work as machine-verified documentation; this often plays an important role in maintainability of software. Although there are these advantages in static typing, it is often too strict and rejects even semantically correct programs. For example, it is difficult for many statically typed languages to deal with heterogeneous lists safely,[1] though (unsafe) workarounds to avoid the problem are provided in most languages.

Type systems have been so suited to verify programs and closely tied to various kinds of programming languages from so-called dynamically typed ones through so-called dependently typed ones. On one hand, dynamically typed languages, which are also called script languages and include Python, Ruby, ECMAScript (JavaScript), many dialects of Lisp, and so on, do not perform type checking statically and instead defer it to run time. If a run-time check fails, then an exception will be raised to notify the failure. In other words, dynamic typing reports errors only when they happen *actually*, unlike static typing, which estimates errors possible to happen. For example, an execution of the Ruby program above will be aborted by raising a run-time exception for failure of the run-time check that `Math.sqrt` should take floating-points as an argument. Dynamically typed languages do not receive the advantages of static typing and their programs consume more computational resources than ones in statically typed languages for the need to record and check kinds of data at run time, whereas dynamically typed languages enable prototypes of software applications to be developed rapidly and exploit features, such as print of formatted string, macro, run-time reflection, serialization/deserialization of objects, etc., difficult to deal with in static typing since their programs are not restricted by rigid constraints imposed by static type systems. On the other hand, dependently typed languages, such as DML [123], Cayenne [11] F* [108], Coq [112], Agda [4], etc., have more powerful static type systems

[1] Some languages, e.g., Haskell [60], can.

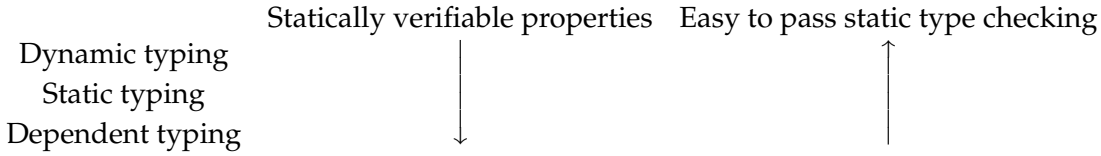

FIGURE 1.1: Comparison of static verification in typing styles.

than usual nondependently, statically typed languages such as Java; we call such nondependently typed languages statically typed ones and their typing styles static typing simply if what they mean is clear from the context. The type systems of dependently typed languages allow types to be dependent on values in programs. For example, Coq can give a type `Vector n` to denote lists of length `n` exactly:

```
Inductive Vector : int -> Type :=
  nil  : Vector 0
| cons : forall n : int, int -> Vector n -> Vector (n+1)
```

Since `n` is an integer to denote the length of lists, the empty list `nil` is given type `Vector 0` and cons constructor returns a list of `Vector (n+1)` when given a sublist of `Vector n`. It is important to notice that `Vector` depends on integer values denoted by `n`. This dependency makes it possible to represent more precise specifications. For example, using `Vector`, Coq would provide a concatenation function of two finite lists with the following type:

```
Vector m -> Vector n -> Vector (n+m)
```

whereas OCaml [66], a statically typed language, would provide it with the following:

```
int list -> int list -> int list
```

where `int list` is a type for finite lists of integers. The former type is more precise than the latter in the sense that the former presents the length of the concatenation but the latter does not. Use of precise specifications leads to early notice of incorrect implementations. Dependently typed languages also accept features, such as print of formatted strings, difficult to give types in usual statically typed languages [11]. Support for dependent types, however, makes type checking difficult, at worst, undecidable [11]. Perhaps worse, powerful type systems require programmers to write enormous type annotations. Figure 1.1 shows a summary of the comparison above: dynamic typing guarantees nothing statically but enables "flexible" programming; static typing detects certain errors statically but rejects apparently correct programs; dependent typing can detect more errors and accept more correct programs but imposes significantly more burdens on both programmers and type checkers.

1.3 Integration of Static and Dynamic Verification

We have seen three kinds of typing styles: dynamic typing, static typing, and dependent typing. Each typing style has its own pros and cons and has been adopted by many programming languages, including ones for research, already. It is natural that combining these typing styles yields more powerful programming styles. Combining static and dynamic typing, we can start with developing a prototype in a dynamically typed language and rewrite parts where specifications become stable to a statically

typed one. Combining static and dependent typing, we can write only critical parts in dependently typed languages for rigorous verification and other parts in a statically typed language to avoid burdens due to dependent typing. In fact, there has been much work to unify static and dynamic typing and do static and dependent typing. For example, programming languages supporting both static and dynamic typing have been developed actively [73, 74, 114, 18, 118, 31] and integration of static and dependent typing has been investigated over a decade [82, 37, 46, 45, 62, 64, 44, 14, 125, 43, 61, 111].

In what follows, we briefly introduce two integration mechanisms which we work over: *gradual typing*, which combines static and dynamic typing, and *manifest contracts*, which does static and dependent typing. Both of these mechanisms allow a certified and an uncertified sides to interact by giving and taking values. This interaction, however, poses a challenge: how is it verified that values flown from the uncertified side follow the specifications in the certified side? The key approach to the challenge is *dynamic verification*—the values from the uncertified side is checked at run time.

1.3.1 Gradual Typing

Gradual typing [101, 113, 120] is a framework to integrate static and dynamic typing, allowing statically typed (typed for short) and dynamically typed (untyped for short) code to coexist. Gradually typed languages allow programmers to write an untyped program at an early stage for rapid development, rewrite parts where specifications become stable to statically typed ones, and obtain a fully typed program finally. For example, let us see how the following untyped, ML-like program is rewritten to a fully typed program:

```
let f g x = g (g x)
let h x = f (fun y -> y + 1) x
let x = h 1
```

where gray parts are untyped. Since integers are typed at `int` and operation `(+)` can be applied to only integers, we can rewrite a few parts in the program to statically typed ones as follows:

```
let f g x = g (g x)
let h x = f (fun (y:int) -> y + 1) x
let x = h 1
```

where white parts are statically typed. Since functions `h` and `f` take only arguments of `int` and `int → int`, respectively, the program is rewritten as follows:

```
let f (g:int→int) x = g (g x)
let h (x:int) = f (fun (y:int) -> y + 1) x
let x = h 1
```

By continuing similar reasoning, we can obtain the fully typed program:

```
let f (g:int→int) (x:int) : int = g (g x)
let h (x:int) : int = f (fun (y:int) -> y + 1) x
let x : int = h 1
```

All of the four programs above are legitimate in gradual typing. In addition to smooth transformation from fully untyped programs to fully typed ones, gradual typing also enables "flexible" programming, like dynamically typed languages. For example, let us consider the following program:

```
let x = if (f 0) then (+) else 1
let y:int = if (f 0) then x 1 2 else x
```

where `f` is supposed to be a statically typed function of `int → bool`. In the first `let` declaration, what value is bound to `x` rests on the conditional expression `f 0`; if `f 0` evaluates to true, then function `(+)` is bound to `x`; otherwise, integer `1` is bound. The body of the second `let` declaration refers to variable `x` appropriately, so no run-time errors will happen. Since `(+)` and `1` have different types, the type of a value of variable `x` cannot be determined statically in simple type systems and so statically typed languages (with simple type systems) would reject this program. By contrast, gradually typed languages accept it because they can suppose that the `let` declaration is untyped.

The key notion for safe interaction between a typed and an untyped parts is monitoring of value flows between the two parts to check at run time that values in the untyped side satisfy the type specifications given by the typed side—in languages with "sound" gradual typing, all run-time value flows are monitored. Perhaps interestingly, value flows from typed parts to untyped parts also have to be monitored when higher-order functions are supported. If it is detected that untyped values did not conform to the specification of a typed part, "blame" [35, 113, 120, 27] (a kind of uncatchable exceptions) will be raised to notify that something unexpected happened.

1.3.2 Manifest Contracts

Manifest contracts [37, 44, 64, 43] are a framework for integrating static and dynamic checking of type specifications including refinement types,[2] which are a kind of dependent types. In manifest contracts, finer-grained specifications than simple types such as int are expressed by using *software contracts* [83, 72, 35] (contracts for short). Contracts were originally introduced to represent formal specifications between a supplier and a client of a software component. In this thesis, we follow the style of Eiffel [72]; contracts mean predicates described in the language used to write programs, allowing programmers to express more precise specifications than simple types and able to be checked at run time since they are program expressions. For example, a contract to denote that variable `x` is positive is written as `x > 0`. In addition to such simple specifications, contracts can represent finer-grained ones such as preconditions, which denote what a function requires, postconditions, which denote what a function guarantees, and invariants, which denote what always holds. As an example of such contracts, let us express a specification of division function. In mathematics, given a dividend and a nonzero divisor, division returns the quotient such that the dividend is equal to the multiplication of the quotient and the divisor. When division function is implemented as the form `let div dividend divisor = ...` in a program, the precondition that a divisor is nonzero is expressed as `divisor <> 0` and the postcondition that the dividend is equal to the multiplication of the quotient and the divisor is expressed as `dividend = result * divisor` where `result` is a variable to denote the quotient.

Many programming languages support contract systems with run-time checking mechanisms as libraries (such as in the C language [55]) or dedicated constructs (such as in Eiffel [72]). The most widely accepted checking mechanism is probably assertion, which takes a Boolean expression and checks its truth at run time. For example,

[2] Refinement types are also known as subset types.

OCaml [66] supports the `assert` construct and, given division function `div`, function `div'` that enforces the pre- and the post-conditions for division is written as follows:

```
let div' dividend divisor =
  assert (divisor <> 0);
  let result = div dividend divisor in
  assert (dividend = result * divisor);
  result
```

If zero is given to `div'` as a divisor, the precondition `divisor <> 0` is checked at run time and an exception is raised to notify failure of the assertion; similarly, if `div` does not return a value satisfying the postcondition, an exception is also raised to notify failure of the postcondition; otherwise, `div'` works in the same way as `div`. Eiffel [72] succeeds in incorporating contracts into interfaces of program components and encourages making pre- and post-conditions of methods and invariants of classes explicit—this approach is called "Design by Contracts" concisely. Racket [38] has a state-of-the-art contract system, which supports higher-order contracts [35], lazy contracts [36], parametric contracts [48], and so on.

Unlike traditional contract systems, dubbed *latent* contracts by Greenberg et al. [44], with only run-time checking mechanisms, *manifest* contracts take advantage of contracts for both static and dynamic verification by embedding contract information into types. The key type construct is *refinement types*. A refinement type $\{x{:}B \mid e\}$ intuitively represents a set of values v of base type B such that the expression $[v/x]e$, which is obtained by substituting v for x in the Boolean expression e, evaluates to true. It is worth noting that e, called a contract, a refinement, or a predicate, can be an *arbitrary* Boolean expression and so refinement types can specify any subset of base-type constants as long as a constraint to specify the subset can be written as a program expression. For example, positive integers are represented as $\{x{:}\mathsf{int} \mid x > 0\}$ and prime numbers are as $\{x{:}\mathsf{int} \mid \mathsf{prime?}\, x\}$ using a user-defined primality test function $\mathsf{prime?}$. Support for *dependent function types*, which allow us to express input/output relationships of functions, makes specification languages more expressive. In general, dependent function types take the form $x{:}T_1 \to T_2$, which means, when taking a value v of T_1, functions of the type return a value of $[v/x]\,T_2$. For example, using type $\mathsf{rational}$ to denote rational numbers, a type of division function div is represented as

$$\begin{aligned} &\mathit{dividend}{:}\mathsf{rational} \to \\ &\quad \mathit{divisor}{:}\{x{:}\mathsf{rational} \mid x \neq 0\} \to \{\mathit{result}{:}\mathsf{rational} \mid \mathit{dividend} = \mathit{result} * \mathit{divisor}\}. \end{aligned}$$

The second argument and the return types denote the pre- and the post-conditions of division, respectively. Thanks to dependency of function types, the return type can describe the relationship of inputs $dividend$ and $divisor$ and the output $result$—the first argument $dividend$ should be equal to the multiplication of the output and the second argument $divisor$. Dependent function types give application terms precise types. For example, the quotient $(div\ 42\ 6)$ of dividend 42 and divisor 6 is typed at $\{\mathit{result}{:}\mathsf{rational} \mid 42 = \mathit{result} * 6\}$.

Specification checking in manifest contracts is expressed as type checking since specifications are denoted by types with contract information. For example, when we want to ensure that an implementation of division function satisfies the contract for division, it is checked that the implementation can be given the above dependent function type. Manifest contracts advocate *hybrid type checking*, an integration of both

static and dynamic type checking, which resolves whether program components satisfy specified contracts statically as much as possible and defers checking to run time if a problem instance is not resolved statically.

What plays an essential role in static type checking is subtype checking. Intuitively, type T_1 is a subtype of T_2 when contract information on T_1 implies that on T_2. Static subtype checking between refinement types is, thus, formalized as an implication relation between the Boolean predicates on the refinement types. For example, the quotient of dividend 42 and divisor 6 is typed at $\{result{:}\mathsf{rational} \mid 42 = result * 6\}$ and the predicate $42 = result * 6$ implies $result > 0$, so the quotient would be also given type $\{result{:}\mathsf{rational} \mid result > 0\}$ statically. Static subtype checking rests on only type information and guarantees that *any* term of a type is regarded as one of its supertype without run-time checks.

By contrast, dynamic type checking uses run-time values in addition to type information and checks that *some* run-time value satisfies contracts on types at which we expect the value to be typed. A key mechanism for dynamic checking is type coercions, more commonly called *casts*. A cast $\langle\, T_1 \Leftarrow T_2 \,\rangle^{\ell}$ performs a run-time check that, when taking a value of T_2, the value satisfies contracts on T_1. If the check fails—i.e., the value violates the contracts—then blame will be raised for notifying failure of the check.

The key to connect static and dynamic checking is in the typing rule for casts, which gives a cast application $\langle\, T_1 \Leftarrow T_2 \,\rangle^{\ell}\, e$ type T_1 on which we expect e to satisfy the contract information. This rule is justified by the cast semantics—if the cast application results in a value, it suggests success of the cast, namely, the target value satisfies the contracts on T_1. Even if the cast application causes blame, there are no problems because at such a time program execution will be aborted.

1.4 This Thesis

Our goal is to introduce a full-fledged programming language where dynamically, statically, and dependently typed code coexist and interact safely. For this goal, we study a theory to advance gradual typing and manifest contracts. Although their key ideas have been established, it is not trivial to extend them so that static and dynamic verification can interact cooperatively because enhancement of dynamic aspects often introduces an undesirable interaction between a certified and an uncertified worlds and breaks foundations of static verification. For example, as pointed out by Ahmed et al. [7], naive addition of polymorphism to gradual typing results in violation of parametricity, a key property of statically typed lambda calculi with polymorphism [41, 89, 90, 119]. A study of extending integration mechanisms with a programming construct also lets us reconsider the construct from the point of view of both certified and uncertified sides and provides new insights about it.

In this thesis, we focus on three universal features in programming: delimited control, parametric polymorphism, and algebraic datatypes. First, we study gradual typing with delimited control. Combining delimited-control operations and integration of certified and uncertified worlds has not been studied in depth (except for Takikawa et al. [110]). Since gradual typing seems to be more straightforward than manifest contracts, we start with extending gradual typing with delimited control. Second, we introduce polymorphism to manifest contracts. Polymorphism has been studied in the context of static and dependent typing for a long time [41, 89, 13] whereas it does not work well naively in both dynamic typing [92] and gradual typing [69, 7]. So, extending manifest contracts with polymorphism is expected to be nontrivial and, in fact, *is*

nontrivial. Finally, we extend manifest contracts with algebraic datatypes. This extension lets us compare two major approaches to giving specifications to data structures from the point of view of computational efficiency.

In what follows, we describe overviews of our extensions briefly.

1.4.1 Gradual Typing with Delimited Control

We study integration of static and dynamic typing in the presence of delimited-control operators. Delimited control is so powerful, a well known programming construct—for example, various operations, such as exception handling, to manipulate call stacks (sequences of activation frames) can be expressed by using delimited continuations. However, control operators make it tricky to monitor the borders between typed and untyped parts, though gradually typed languages should monitor them; in fact, as is pointed out by Takikawa, Strickland, and Tobin-Hochstadt [110], communications between the two parts via continuations captured by control operators can be overlooked under a standard monitoring system.

We propose an extension of a blame calculus [113, 120], a model calculus for gradual typing, with Danvy and Filinski's delimited-control operators *shift* and *reset* [25] and give a new cast-based mechanism to monitor all communications between typed and untyped parts. The idea of the new cast comes from Danvy and Filinski's type system [24] for shift/reset, where type information about contexts is considered. Using types of contexts, our system can monitor all communications.

As a proof of correctness of our idea, we investigate two important properties. One is Blame Theorem [113, 120], which states that values that flow from typed code never trigger run-time type errors. The other property is soundness of continuation passing style (CPS) transformation—investigating properties about CPS is important in study of shift/reset because the origin and foundations of these operators come from CPS [24, 25, 57, 8]. Especially, after giving a CPS transformation for our calculus, we show that it preserves well-typedness and, for any two source terms such that one reduces to the other, their transformation results are equivalent in the target calculus. It turns out that we need a few axioms about casts in addition to usual axioms, such as (call-by-value) β-reduction, for equality in the target calculus.

1.4.2 Manifest Contracts with Parametric Polymorphism

We study manifest contracts with *parametric polymorphism* [41, 89, 90], which is a key device of type-based abstraction in functional programming—for example, it is well known that polymorphism can encode abstract datatypes (ADTs) [78]. Manifest contracts are a sensible choice for combining contracts and type-based abstraction mechanisms including ADTs. ADTs already use the type system to mediate access to abstractions; manifest contracts allow types to exercise a still finer grained control. We will see an example motivating the combination of contracts and ADTs in Chapter 3.

To study combination of contracts and polymorphism, we introduce a polymorphic manifest contract calculus $\mathrm{F}^{\sigma}_{\mathrm{H}}$. Our calculus supports, in addition to polymorphism, "general refinements," where the underlying type T of refinement type $\{x{:}T \mid e\}$ can be arbitrary, unlike earlier manifest contracts [37, 44, 64], where T is restricted to be base types. Thanks to support for general refinements, abstract datatypes can be implemented by any type. General refinements also allow intuitive specifications to be described by stating what a function produces when given specific arguments. For example, let us consider a type of a root-finding algorithm, which, given a continuous

function f over real numbers, calculates an approximation of a real number x such that $f\,(x) = 0$; such x is called a root of f. Naturally, as a precondition, the algorithm requires argument function f to have a root. This precondition can be expressed as $f\,(a) < 0 \,\mathsf{and}\, 0 < f\,(b)$ for some real numbers a and b—since f is continuous, it implies the existence of a root of f. When we suppose that such a and b along with f are given by users, the root-finding algorithm is given the dependent function type of:[3]

$$a{:}\mathsf{real} \to b{:}\mathsf{real} \to f{:}\{f{:}\mathsf{real} \to \mathsf{real} \mid (f\ a) < 0 \,\mathsf{and}\, 0 < (f\ b)\} \to \{x{:}\mathsf{real} \mid \mathsf{abs}\,(f\ x) < \epsilon\}$$

where real is the type for real numbers, abs returns the absolute value of an argument, and ϵ is an approximation error. Without general refinements, this type would not be permitted because $\mathsf{real} \to \mathsf{real}$ is not a base type.

We establish fundamental properties including type soundness and relational parametricity; in fact, our calculus is the first sound polymorphic manifest contract calculus—though some earlier work studied manifest contracts with polymorphism, they have some metatheoretical problems. Our key observation for soundness is that cast semantics should not be affected by substitution—to design such cast semantics, our calculus uses delayed substitution on casts and a new type conversion relation.

1.4.3 Manifest Contracts with Algebraic Datatypes

We extend manifest contracts with algebraic datatypes. Our calculus supports two simple approaches to giving refinements to data structures: one gives refinements to type constructors and the other to data constructors. For example, let us see what type is given to lists of positive integers in each approach. In the former, using predicate function for_all which returns whether all elements of a given list satisfy a given predicate, a type of lists of positive integers can be described as:

$$\{l{:}\mathsf{int\ list} \mid \mathsf{for\_all}\,(\lambda y.y > 0)\ l\}$$

where type constructor int list is refined. In the latter, such lists can be described by defining a new datatype pos list:

$$\mathsf{type\ pos\ list} = \mathsf{PNil} \mid \mathsf{PCons}\ :\ \{x{:}\mathsf{int} \mid x > 0\} \times \mathsf{pos\ list}$$

where data constructors PNil and PCons, which correspond to the nil and the cons constructors of lists, are refined.

The two approaches are complementary. The former makes it easier for a programmer to write types because writing program predicates on data structures is comparatively easy. The latter enables more efficient contract checking because we can find what contracts hold for data structures from contract information on types of data constructors.

Our goal is to take the best of both approaches and transform programs using refinements on type constructors to ones using refinements on data constructors automatically. In this thesis, as a stepping stone to the goal, we propose two mechanisms for achieving the program transformation. First, a syntactic translation from refinements on type constructors to equivalent refinements on data constructors. Using this translation, for example, we can derive the definition of pos list from $\{l{:}\mathsf{int\ list} \mid \mathsf{for\_all}\,(\lambda y.y > 0)\ l\}$ automatically. Second, dynamically checked casts between different

[3] The type presented here does not state that f is continuous but we can by using dependent products, which are provided in Chapter 4.

but compatible datatypes such as int list and pos list. Such casts are useful when, say, we want to apply list-processing functions to inhabitants of pos list. As technical development, we define a manifest contract calculus $\lambda^{H}_{\mathbf{dt}}$ to formalize the semantics of the casts and prove that the translation is correct. The formalization of $\lambda^{H}_{\mathbf{dt}}$ is slightly different from $\mathrm{F}^{\sigma}_{\mathrm{H}}$ (the calculus for manifest contracts with parametricity)—in particular, $\lambda^{H}_{\mathbf{dt}}$ does not rest on delayed substitution, differently from $\mathrm{F}^{\sigma}_{\mathrm{H}}$, and, in this sense, the metatheory of $\lambda^{H}_{\mathbf{dt}}$ is simpler than that of $\mathrm{F}^{\sigma}_{\mathrm{H}}$; we will discuss it in detail in Section 4.2.

1.5 Organization

The rest of this thesis is organized as follows. In Chapter 2, we study gradual typing with delimited control. After seeing how delimited control in a standard monitoring system causes overlooking interaction between typed and untyped parts, we describe an idea to monitor such interaction. To formalize the idea, we introduce a blame calculus with CPS-based delimited-control operators shift and reset; the cast semantics of the calculus monitors capture and call of delimited continuations by considering types of contexts. Then, we show standard properties: Type Soundness, Blame Theorem, and soundness of our CPS transformation. Chapter 3 extends manifest contracts with parametric polymorphism. We start with reviewing prior work on manifest contracts—in particular, we see that earlier polymorphic manifest contracts [14, 43] are not sound. In Section 3.2, we introduce the first conjecture-free, sound polymorphic manifest contract calculus with the help of delayed substitution and a new type conversion relation. Finally, we show that our calculus has a crucial property, called parametricity, in polymorphic lambda calculi. Chapter 4 proposes a manifest contract calculus with algebraic datatypes. The calculus supports two approaches to giving refinements data structures: refinements on type constructors and refinements on data constructors. Section 4.3 relates these two approaches via the calculus. Chapter 5 discusses work related to our work and Chapter 6 concludes this thesis, describing future work.

This thesis is constituted by papers which have been presented or submitted already. Chapter 2 and Chapter 4 are based on the papers presented at APLAS 2015 [99] and POPL 2015 [98], respectively. Chapter 3 is based on the one submitted to TOPLAS [97].

We state only key lemmas and theorems in the body of the thesis; all of lemmas and theorems and their proofs are presented in Appendix.

Chapter 2

Gradual Typing with Delimited Control

We study gradual typing in the presence of delimited-control operators. As discussed in Section 1.3.1, the run-time system of a gradually typed language should monitor all flows of values between typed and untyped parts. Traditionally, *casts* [101, 37, 52, 102, 120, 7, 54, 105, 104] (or *contracts* [35, 113, 109, 27, 110]) play an important role in such a monitoring system. A source program that contains typed and untyped parts is compiled to an intermediate language such that casts are inserted in points where typed and untyped code interacts. Casts are a run-time mechanism to check that a program component satisfies a given type specification. For example, when typed code imports a certain component from untyped code as integer, a cast is inserted to check that it is actually an integer at run time. If it is detected that a component did not follow the specification, blame (a kind of uncatchable exceptions) will be raised to notify that something unexpected has happened. Tobin-Hochstadt and Felleisen [113] originated a *blame calculus* to study integration of static and dynamic typing and Wadler and Findler [120] refined the theory of blame on its variant.

It is well known that use of delimited continuations as first-class values has many applications—e.g., it is possible to implement various control effects such as exception handling [106], backtracking [25], monads [34], generators [106], etc. However, control operators make it tricky to monitor the borders between typed and untyped parts; in fact, as is pointed out by Takikawa, Strickland, and Tobin-Hochstadt [110], communications between the two parts via continuations captured by control operators can be overlooked under standard cast semantics.

In this chapter, we propose a blame calculus, based on Wadler and Findler [120], with Danvy and Filinski's delimited-control operators *shift* and *reset* [25] and give a new cast-based mechanism to monitor all communications between typed and untyped parts. The idea of the new cast comes from Danvy and Filinski's type system [24] for shift/reset, where type information about contexts is considered. Using types of contexts, our cast mechanism can monitor all communications.

As a proof of correctness of our idea, we investigate two important properties. One is Blame Theorem [113, 120], which states that values that flow from typed code never trigger run-time type errors. The other property is soundness of CPS transformation: it preserves well-typedness and, for any two source terms such that one reduces to the other, their transformation results are equivalent in the target calculus. It turns out that we need a few axioms about casts in addition to usual axioms, such as (call-by-value) β-reduction, for equality in the target calculus.

Outline In Section 2.1, we review the blame calculus and the control operators shift/reset, explain why the standard cast does not work when they are naively combined, and briefly describe our solution. Section 2.2 formalizes our calculus with an operational semantics and a type system, and shows type soundness of the calculus. Section 2.3 shows a Blame Theorem in our calculus and Section 2.4 introduces a CPS transformation and shows its soundness.

2.1 Blame Calculus with Shift and Reset

2.1.1 Blame Calculus

The blame calculus of Wadler and Findler [120] is a kind of typed lambda calculus for studying integration of static and dynamic typing. It is designed as an intermediate language for gradually typed languages [101], where a program at an early stage is written in an untyped language and parts whose specifications are stable can be gradually rewritten in a typed language, resulting in a program with both typed and untyped parts. In blame calculi, untyped parts are represented as terms of the special, *dynamic type* [1, 51, 101] (denoted by $\star$), where any operation is statically allowed at the risk of causing run-time errors. Blame calculi support smooth interaction between typed and untyped parts—i.e., typed code can use an untyped component and vice versa—via a type-directed mechanism, *casts*.

A cast, taking the form $t : A \Rightarrow^p B$, checks that term t of source type A behaves as target type B at run time; p, called a *blame label*, is used to identify the cast that has failed at run time. For example, using integer type int, cast expression $1 : \mathsf{int} \Rightarrow^p \star$ injects integer 1 to the dynamic type; conversely, $t : \star \Rightarrow^p \mathsf{int}$ coerces untyped term t to int. A cast would fail if the coerced value cannot behave as the target type of the cast. For example, cast expression $(1 : \mathsf{int} \Rightarrow^{p_1} \star) : \star \Rightarrow^{p_2} \mathsf{bool}$, which coerces integer 1 to the dynamic type and then its result to Boolean type bool, causes blame $\mathsf{blame}\ p_2$ at run time since the coerced value 1 cannot behave as bool.

Using casts, in addition to fully typed and fully untyped programs, we can write a program where typed and untyped parts are mixed. For example, suppose that we first write an untyped program as follows:

$$\mathsf{let}\ \mathit{succ} = \lambda x.\, x + 1\ \mathsf{in}\ \mathit{succ}\, 1$$

where we color untyped parts gray.[1] If the successor function is statically typed, we rewrite the program so that it imports the typed successor function:

$$\mathsf{let}\ \mathit{succ} = (\lambda x.\, x + 1) : \mathsf{int} \to \mathsf{int} \Rightarrow^p \star\ \mathsf{in}\ \mathit{succ}\, 1$$

where we color typed parts white. When the source and target types in a cast are not important, as is often the case, we just surround a term by a frame to indicate the existence of some appropriate cast. So, the program above is presented as below:

$$\mathsf{let}\ \mathit{succ} = \boxed{\lambda x.\, x + 1}\ \mathsf{in}\ \mathit{succ}\, 1$$

Intuitively, a frame in programs in this style means that flows of values between the typed and untyped parts are monitored by casts. Conversely, the absence of a frame

[1] Precisely speaking, even untyped programs need casts to use values of the dynamic type as functions, integers, etc., but we omit them to avoid the clutter.

between the two parts indicates that the run-time system will overlook their communications.

What happens when a value is coerced to the dynamic type rests on the source type of the cast. If it is a first-order type such as int, the cast simply tags the value with its type. If it is a function type, by contrast, the cast generates a lambda abstraction that wraps the target function and then tags the wrapper. The wrapper, a function over values of the dynamic type, checks, by using a cast, that a given argument has the type expected by the wrapped function and coerces the return value of the wrapped function to the dynamic type, similarly to function contracts [35]. For example, cast expression $(\lambda x{:}\mathsf{int}.\, x+1)\ :\ \mathsf{int}\to\mathsf{int}\Rightarrow^{p}\ \star$ generates lambda abstraction $\lambda y{:}\star.\,(((\lambda x{:}\mathsf{int}.\, x+1)\,(y\ :\ \star\Rightarrow^{q}\ \mathsf{int}))\ :\ \mathsf{int}\Rightarrow^{p}\ \star)$. Here, blame label q is the negation of p, which we will discuss in detail below. Using the notation introduced above, it is easy to understand that all communications between typed and untyped parts are monitored because the program above reduces to:

$$\mathsf{let}\ succ = \lambda y.\ \boxed{(\lambda x.\, x+1)\ \boxed{y}}\ \mathsf{in}\ succ\, 1$$

As advocated by Findler and Felleisen [35], there are two kinds of blame—positive blame and negative blame, which indicate that, when a cast fails, its responsibility lies with the term contained in the cast and the context containing the cast, respectively. Following Wadler and Findler, we introduce an involutive operation $\bar{\cdot}$ of negation on blame labels: for any blame label p, $\bar{p}$ is its negation and $\bar{\bar{p}}$ is the same as p. For a cast with blame label p in a program, $\mathsf{blame}\, p$ and $\mathsf{blame}\, \bar{p}$ denotes positive blame and negative blame, respectively. A key observation, so-called the *Blame Theorem*, in work on blame calculi is that a cast failure is never caused by values from the more precisely typed side in the cast—i.e., if the side of a term contained in a cast with p is more precisely typed, a program including the cast never evaluates to $\mathsf{blame}\, p$, while if the side of a context containing the cast is, the program never evaluates to $\mathsf{blame}\, \bar{p}$.

2.1.2 Delimited-Control Operators: Shift and Reset

Shift and *reset* are delimited-control operators introduced by Danvy and Filinski [25]. Shift captures the current continuation, like another control operator call/cc, and reset delimits the continuation captured by shift. The captured continuation works as if it is a composable function, namely, unlike call/cc, control is returned to a caller when the call to the captured continuation finishes.

As an example with shift and reset, let us consider the following program:

$$\langle 5 + \mathcal{S}k.\,((k\,1 + k\,2) = 13)\rangle$$

Here, the shift operator is invoked by the subterm $\mathcal{S}k.\,((k\,1 + k\,2) = 13)$ and the reset operator $\langle ... \rangle$ encloses the whole term. To evaluate a reset operator, we evaluate its body. Evaluation of the shift operator $\mathcal{S}k.\,((k\,1 + k\,2) = 13)$ proceeds as follows. First, it captures the continuation up to the closest reset as a function. Since the delimited continuation in this program is $5 + [\,]$ (here, $[\,]$ means a hole of the context), the captured continuation takes the form $\lambda x.\,\langle 5 + x\rangle$ (note that the body of the function is enclosed by reset). Next, variable k is bound to the captured continuation. Finally, the body of the closest reset operator is replaced with the body of the shift operator. Thus, the example

program reduces to:

$$\langle(((\lambda x.\,\langle 5 + x\rangle)\,1) + ((\lambda x.\,\langle 5 + x\rangle)\,2)) = 13\rangle.$$

Since reset returns the result of its body, it evaluates to true.

Let us consider a more interesting example of function $choice$, a user of which passes a tuple of integers and expects to return one of them. The caller tests the returned integer by some Boolean expression and surrounds it by reset. Then, the whole reset expression evaluates to the index (tagged with Some) to indicate which integer satisfied the test, or None to indicate none of them satisfied. For example, $\langle \texttt{prime?}\,(choice\,(141, 197))\rangle$ will evaluate to Some 2 because the second argument 197 is a prime number. Using shift/reset, such a (two-argument version of) $choice$ function can be defined as follows:

$$choice = \lambda(x, y){:}\mathsf{int} \times \mathsf{int}.\,\mathcal{S}k.\,\mathsf{if}\ k\ x\ \mathsf{then\ Some}\ 1\ \mathsf{else\ if}\ k\ y\ \mathsf{then\ Some}\ 2\ \mathsf{else\ None}$$

It is important to observe k is bound to the predicate (in this case, $\lambda z.\,\langle \texttt{prime?}\ z\rangle$).

Since blame calculi support type-directed casts, it is crucial to consider type discipline in the presence of shift/reset. This work adopts the type system proposed by Danvy and Filinski [24]. Their type system introduces types, called *answer types*, of contexts up to the closest reset to track modification of the body of a reset operator—we have seen above that the body of a reset operator can be modified to the body of a shift operator at run time. In the type system, using metavariables α and β for types, function types take the form $A/\alpha \to B/\beta$, which means that a function of this type is one from A to B and, when applied, it modifies the answer type α to β. For example, using a function of type $(\mathsf{int} \times \mathsf{int})/\mathsf{bool} \to \mathsf{int}/(\mathsf{int\ option})$ (int option means integers tagged with Some and None), its user, when passing a pair of integers, expects to return an integer value and to modify the answer type bool to int option. Conversely, to see how functions are given such a function type, let us consider $choice$, which is typed at $(\mathsf{int} \times \mathsf{int})/\mathsf{bool} \to \mathsf{int}/(\mathsf{int\ option})$. It can be found from the type annotation that it takes pairs of integers. The body captures a continuation and calls it with the first and second components of the argument pair. Since a caller of $choice$ obtains a value passed to the continuation k, the return type is int. $choice$ demands the answer type of a context be bool because the captured continuation is required to return a Boolean value in conditional expressions; and the shift operator modifies the answer type to int option because the if-expression returns an int option value.

2.1.3 Blame Calculus with Shift and Reset

We extend the blame calculus with shift/reset so that all value flows between typed and untyped parts are monitored, following the type discipline discussed above. The main question here is how we should give the semantics of casts for function types, which now include answer type information. The standard semantics discussed above does not suffice because it is ignorant of answer types. In fact, it would fail to monitor value flows that occur due to manipulation of delimited continuations, as we see below. For example, let us consider the situation that untyped code imports typed function $choice$ via a cast (represented by a frame):

$$\mathsf{let}\ f = \boxed{choice}\ \mathsf{in}\ 5 + \langle succ\,(f\,(141, 197))\rangle$$

This program contains two errors: first, subterm $succ\,(f\,(141, 197))$ within the reset operator returns an integer, though the shift operator in $choice$ expects it to return a Boolean value since the continuation captured by the shift operator is used in conditional expressions; second, as found in subterm $5 + \langle \ldots \rangle$, the computation result of the reset operator is expected to be an integer, though it should be an int option value coming from the body of the shift operator in $choice$. However, if the cast on $choice$ behaved as a standard function cast we discussed in Section 2.1.1, these errors would not be detected at run time on borders between typed and untyped parts. To see the reason, let us reduce the program. First, since the choice is coerced to the dynamic type, a wrapper that checks an argument and the return value is generated and then is applied to $(141, 197)$:

$$\mathsf{let}\ f = \boxed{choice}\ \mathsf{in}\ 5 + \langle succ\,(f\,(141, 197))\rangle \longmapsto^* 5 + \langle succ\,(\boxed{choice\ \boxed{(141, 197)}})\rangle$$

The check for $(141, 197)$ succeeds and so $choice$ is applied to $(141, 197)$, and then the shift operator in $choice$ is invoked.

$$\begin{aligned} \cdots &\longmapsto^* \quad 5 + \langle succ\ \boxed{\mathcal{S}k.\ \mathsf{if}\ k\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else\ if}\ k\ 197\ \mathsf{then\ Some}\ 2\ \mathsf{else\ None}}\rangle \\ &\longmapsto^* \quad 5 + \langle\ \mathsf{if}\ (\lambda x.\ \langle succ\ \boxed{x}\rangle)\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else\ if}\ \ldots\ \mathsf{then\ Some}\ 2\ \mathsf{else\ None}\ \rangle \end{aligned}$$

Here, there are one gray area and one white area, both without surrounding frames. The former means that the value flow from the captured continuation $\lambda x.\ \langle succ\ \boxed{x}\rangle$ to typed code will not be monitored, when it should be by the cast from the dynamic type to bool. Similarly, the latter means that the value flow from the result of the (typed) if-expression to untyped code will not be monitored, either, when it should be by the cast from int option to the dynamic type. The problem is that the standard function casts can monitor calls of functions but does not capture and calls of delimited continuations.

Our cast mechanism can monitor such capture and calls of delimited continuations. A wrapper generated by a cast from $A/\alpha \to B/\beta$ to the dynamic type, when applied, ensures that the reset expression enclosing the application returns a value of the dynamic type by inserting injection from β and that the continuation captured during the call to the wrapped function returns a value of α by the cast to α. In the above example of $choice$, our cast mechanism reduces the original program to a term like:

$$5 + \langle\ \boxed{\mathsf{if}\ (\lambda x.\ \boxed{\langle succ\ \boxed{x}\rangle})\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else\ if}\ \ldots\ \mathsf{then\ Some}\ 2\ \mathsf{else\ None}}\ \rangle$$

where two casts are added: one to check that the return value of the continuation has bool and the other to inject the result of the if-expression to the dynamic type.

2.2 Language

In this section, we formally define a call-by-value blame calculus with delimited-control operators shift and reset and show its type soundness. Our calculus is a variant of the blame calculus by Ahmed et al. [7].

variables	x, y, z, k		**blame labels**	p, q
constants	c	$::=$ true | false | ...	**base types**	$\iota ::=$ bool | ...
ground types	G, H	$::=$	$\iota \mid \star/\star \to \star/\star$	
types	$A, B, \alpha, \beta, \gamma, \delta$	$::=$	$\iota \mid \star \mid A/\alpha \to B/\beta$	
values	v	$::=$	$x \mid c \mid \lambda x.\, t \mid v : G \Rightarrow \star$	
terms	s, t, u	$::=$	$x \mid c \mid op(\overline{t_i}^i) \mid \lambda x.\, t \mid s\ t \mid$ $s : A \Rightarrow^p B \mid s \text{ is } G \mid s : G \Rightarrow \star \mid \text{blame } p \mid$ $\langle s \rangle \mid \mathcal{S}k.\, s$	

FIGURE 2.1: Syntax.

2.2.1 Syntax

Figure 2.1 presents the syntax, which is parameterized over base types, denoted by ι, constants, denoted by c, and primitive operations, denoted by op, over constants. We assume that at least Booleans are available in our calculus.

Types consist of base types, the dynamic type, and function types with answer types. Unlike the blame calculus of Wadler and Findler, our calculus does not include refinement types for simplicity; we believe that it is not hard to add refinement types if refinements are restricted to be pure [8]. Ground types, denoted by G and H, classify kinds of values. If the ground type is a base type, the values are constants of the base type, and if it is a function type (constituted only of the dynamic type), the values are lambda abstractions.

Values, denoted by v, consist of variables, constants, lambda abstractions, and ground values. A lambda abstraction $\lambda x.\, t$ is standard; variable x is bound in the body t. A ground value $v : G \Rightarrow \star$ is a value of the dynamic type; the kind of v follows ground type G.

Terms, denoted by s and t, extend those in the simply typed blame calculus with two forms, reset expressions and shift expressions. Using the notation $\overline{t_i}^i$ to denote a sequence $t_1, ..., t_n$ of terms, we allow primitive operators to take tuples of terms. A type test s is G investigates a kind of the result of term s of the dynamic type at run time. If the value of s matches with G, then it returns true; otherwise, it returns false. A reset expression is written as $\langle s \rangle$ and a shift expression is as $\mathcal{S}k.\, s$ where k is bound in the body s. The syntax includes blame as a primitive construct despite the fact that exceptions can be implemented by shift and reset because blame is an *uncatchable* exception in a blame calculus. Note that ground values, ground terms ($s : G \Rightarrow \star$), and blame are supposed to be "run-time" citizens that appear only during reduction and not in a source program.

In what follows, as usual, we write $s\,[x := v]$ for capture-avoiding substitution of v for variable x in s. As shorthand, we write $s : G \Rightarrow \star \Rightarrow^p A$ and $s : A \Rightarrow^p G \Rightarrow \star$ for $(s : G \Rightarrow \star) : \star \Rightarrow^p A$ and $(s : A \Rightarrow^p G) : G \Rightarrow \star$, respectively.

2.2.2 Semantics

The semantics of our calculus is given in a small-step style by using two relations over terms: reduction relation $\longrightarrow$, which represents basic computation such as β-reduction, and evaluation relation $\longmapsto$, in which subterms are reduced.

$\boxed{s \longrightarrow t}$ **Reduction rules**

$$
\begin{array}{lcll}
op(\overline{v_i}^{\,i}) & \longrightarrow & \zeta\,(op, \overline{v_i}^{\,i}) & \text{R\_Op} \\
(\lambda x.\, s)\; v & \longrightarrow & s\,[x := v] & \text{R\_Beta} \\
\langle v \rangle & \longrightarrow & v & \text{R\_Reset} \\
\langle F[\mathcal{S}k.\, s] \rangle & \longrightarrow & \langle s\,[k := \lambda x.\, \langle F[x] \rangle] \rangle \quad \text{where } x \notin fv\,(F) & \text{R\_Shift} \\
v \;:\; \iota \Rightarrow^{p} \iota & \longrightarrow & v & \text{R\_Base} \\
v \;:\; \star \Rightarrow^{p} \star & \longrightarrow & v & \text{R\_Dyn} \\
v \;:\; A/\alpha \to B/\beta \Rightarrow^{p} A'/\alpha' \to B'/\beta' & \longrightarrow & & \\
\qquad \lambda x.\, \mathcal{S}k.\, (\langle (k\,((v\,(x \;:\; A' \Rightarrow^{\bar{p}} A)) \;:\; B \Rightarrow^{p} B')) \;:\; \alpha' \Rightarrow^{\bar{p}} \alpha \rangle \;:\; \beta \Rightarrow^{p} \beta') & & & \text{R\_Wrap} \\
v \;:\; A \Rightarrow^{p} \star & \longrightarrow & v \;:\; A \Rightarrow^{p} G \Rightarrow \star \quad \text{if } A \sim G \text{ and } A \neq \star & \text{R\_Ground} \\
v \;:\; G \Rightarrow \star \Rightarrow^{p} A & \longrightarrow & v \;:\; G \Rightarrow^{p} A \quad \text{if } G \sim A \text{ and } A \neq \star & \text{R\_Collapse} \\
v \;:\; G \Rightarrow \star \Rightarrow^{p} A & \longrightarrow & \mathsf{blame}\; p \quad \text{if } G \not\sim A & \text{R\_Conflict} \\
(v \;:\; G \Rightarrow \star)\; \mathsf{is}\; G & \longrightarrow & \mathsf{true} & \text{R\_IsTrue} \\
(v \;:\; H \Rightarrow \star)\; \mathsf{is}\; G & \longrightarrow & \mathsf{false} \quad \text{if } H \neq G & \text{R\_IsFalse}
\end{array}
$$

$\boxed{s \longmapsto t}$ **Evaluation rules**

$$\frac{s \longrightarrow t}{E[s] \longmapsto E[t]}\ \text{E\_Step} \qquad \frac{E \neq [\,]}{E[\mathsf{blame}\; p] \longmapsto \mathsf{blame}\; p}\ \text{E\_Abort}$$

FIGURE 2.2: Reduction and evaluation.

The reduction rules, shown at the top of Figure 2.2, are standard or similar to the previous calculi except (R_Wrap), which is the key of our work. In (R_Op), to reduce a call to a primitive operator, we assume that there is a function ζ which returns an appropriate value when taking an operator name and arguments to it. The rule (R_Shift) presents that the shift operator captures the continuation up to the closest reset operator. In the rule, the captured continuation is represented by pure evaluation contexts, denoted by F, which are evaluation contexts [33] where the hole does not occur in bodies of reset operators. Pure evaluation contexts are defined as follows:

$$F ::= [\,] \mid op(\overline{v_i}^{\,i}, F, \overline{t_j}^{\,j}) \mid F\; s \mid v\; F \mid F \;:\; A \Rightarrow^{p} B \mid F \;:\; G \Rightarrow \star \mid F\; \mathsf{is}\; G$$

As mentioned earlier, the body of the function representing the captured continuation is enclosed by reset. A type test succeeds and returns true if the kind of a examined value matches with the specified ground type (by (R_IsTrue)); otherwise, it returns false (by (R_IsFalse)).

There are six reduction rules for cast expressions. The rules (R_Base) and (R_Dyn) mean that casts between the same base type and between the dynamic type perform no checks. We find (R_Dyn), which does not appear in Ahmed et al. [7], matches well with CPS transformation; we will discuss it in Section 2.4. The rule (R_Ground), applied when the target type is the dynamic type but the source type is not, turns a cast expression to a ground term by inserting a cast to the ground type G that represents the kind of the value v. The relation $\sim$, called compatibility, over two types is defined by using rules in Figure 2.3. It intuitively means that a cast from A to B (and vice

$\boxed{A \sim B}$ **Compatibility rules**

$$\frac{}{A \sim \star}\ \text{C\_DynTo} \qquad \frac{}{\star \sim B}\ \text{C\_DynFrom} \qquad \frac{}{\iota \sim \iota}\ \text{C\_Base}$$

$$\frac{A' \sim A \quad B \sim B' \quad \alpha' \sim \alpha \quad \beta \sim \beta'}{A/\alpha \to B/\beta \sim A'/\alpha' \to B'/\beta'}\ \text{C\_Fun}$$

FIGURE 2.3: Compatibility rules.

versa) can succeed; in other words, $A \not\sim B$ means that a cast will fail. One interesting fact about compatibility is that, for any nondynamic type A, we can find exactly one ground type that is compatible with A: If A is a base type, then G is equal to A and, if A is a function type, then G is $\star/\star \to \star/\star$. As a result, G in (R_Ground) is uniquely determined. The rules (R_Collapse) and (R_Conflict) are applied when a target value is a ground value. When the kind G of the underlying value v is not compatible with the target type of the cast, the cast is blamed with blame label p by (R_Conflict). Otherwise, the underlying value is coerced from the ground type of the ground value to the target type of the cast by (R_Collapse).

The reduction rule (R_Wrap), applied to casts between function types, is the most involved. The rule means that the cast expression reduces to a lambda abstraction that wraps the target value v. Since the wrapper function works as a value of type $A'/\alpha' \to B'/\beta'$, it takes a value of A'. Like function contracts [35], in the wrapper, the argument denoted by x is coerced to argument type A of the source type to apply v to it and the return value of v is coerced to return type B' of the target type. Furthermore, to call the target function in a context of answer type α, the wrapper captures the continuation in which the wrapper is applied by using shift, applies the captured continuation to the result of the target function, and then coerces the result of the captured continuation to α. Since the wrapper is applied in a context of answer type α', the captured continuation returns a value of α'. By enclosing the cast to α with reset, a continuation captured during the call to v returns a value of α. Finally, the wrapper coerces the result of the reset operator from β to β' because the call to the target function modifies the answer type of the context to β, and so the reset expression returns a value of β, and the wrapper is expected to modify the answer type to β'. The rule (R_Wrap) reverses blame labels for casts from A' to A and from α' to α because target values for those casts originate from the context side.

We illustrate how (R_Wrap) makes monitoring of capture and calls of continuations possible, using *choice* in Section 2.1.3. By (R_Ground), the cast from $(\mathsf{int} \times \mathsf{int})/\mathsf{bool} \to \mathsf{int}/(\mathsf{int\ option})$ to the dynamic type reduces to that to $\star/\star \to \star/\star$. By (R_Wrap), the cast generates a wrapper.

$$\begin{array}{ll} & \mathsf{let}\ f = \boxed{\mathit{choice}}\ \mathsf{in}\ 5 + \langle \mathit{succ}\,(f\,(141, 197))\rangle \\ \longmapsto & \mathsf{let}\ f = \lambda x.\,\mathcal{S}k'.\ \Big\langle \boxed{k'\ \boxed{(\mathit{choice}\ \boxed{x})}} \Big\rangle\ \mathsf{in}\ 5 + \langle \mathit{succ}\,(f\,(141, 197))\rangle \end{array}$$

The wrapper is applied to $(141, 197)$, so the evaluation proceeds as follows:

$$
\begin{array}{lll}
\cdots & \longmapsto^* & 5 + \langle succ\,(\mathcal{S}k'.\,\langle\boxed{k'\,\boxed{(choice\,\boxed{(141,197)})}}\rangle)\rangle \\
& \longmapsto & 5 + \langle\langle\boxed{(\lambda x.\,\langle succ\,x\rangle)\,\boxed{(choice\,\boxed{(141,197)})}}\rangle\rangle \\
& \longmapsto^* & 5 + \langle\langle\boxed{(\lambda x.\,\langle succ\,x\rangle)\,\boxed{(\mathcal{S}k.\,\mathsf{if}\ k\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else}\ \ldots)}}\rangle\rangle \\
& \longmapsto & 5 + \langle\boxed{\langle\mathsf{if}\ v\ 141\ \mathsf{then\ Some}\ 1\ \mathsf{else\ if}\ v\ 197\ \mathsf{then\ Some}\ 2\ \mathsf{else\ None}\rangle}\rangle
\end{array}
$$

where $v = \lambda y.\,\langle\boxed{(\lambda x.\,\langle succ\,x\rangle)\,\boxed{y}}\rangle$. We can observe that all borders in the last term are monitored by casts.

Evaluation rules, presented at the bottom of Figure 2.2, are standard: (E_STEP) reduces a subterm that is a redex in a program and (E_ABORT) halts evaluation of a program at blame when cast failure happens. To determine a redex in a program, we use evaluation contexts [33], which are defined as follows.

$$E ::= [\,] \mid op(\overline{v_i}^i, E, \overline{t_j}^j) \mid E\ s \mid v\ E \mid \langle E\rangle \mid E : A \Rightarrow^p B \mid E : G \Rightarrow \star \mid E\ \mathsf{is}\ G$$

This definition means that terms are evaluated from left to right. Unlike pure evaluation contexts, evaluation contexts include a context where the hole is put in the body of a reset operator.

2.2.3 Type System

This section presents a type system of our calculus. It is defined as a combination of that of Danvy and Filinski and that of Wadler and Findler. As usual, we use typing contexts, denoted by Γ, to denote a mapping of variables to types:

$$\Gamma ::= \emptyset \mid \Gamma, x{:}A$$

Typing judgments in our type system take the form $\Gamma; \alpha \vdash s : A; \beta$, which means that term s is typed at type A under typing context Γ and it modifies answer type α to β when evaluated. Perhaps, it may be easier to understand what the typing judgment means when its CPS transformation is considered. When we write $[\![\cdot]\!]$ for the CPS transformation, the typing judgment $\Gamma; \alpha \vdash s : A; \beta$ is translated into the form $[\![\Gamma]\!] \vdash [\![s]\!] : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$ in the simply typed blame calculus (without shift/reset). That is, type A of term s and type α are the argument type and the return type of a continuation, respectively, and type β is the type of the whole computation result when the continuation is passed.

Figure 2.4 shows typing rules for deriving typing judgments. Typing rules for shift operators, reset operators, and terms from the lambda calculus are the same as Danvy and Filinski's type system. In (T_OP), we use function ty from primitive operator names to their (first-order) types. Typing rules for terms from the blame calculus are changed to follow Danvy and Filinski's type system. In (T_CAST), following previous work on the blame calculus, we restrict casts in well typed programs to be ones between compatible types. In other words, (T_CAST) rules out casts that will always fail.

$\boxed{\Gamma; \alpha \vdash t \,:\, A; \beta}$ **Typing rules**

$$\frac{}{\Gamma; \alpha \vdash c \,:\, ty\,(c); \alpha}\ \text{T\_Const} \qquad \frac{ty\,(op) \,=\, \overline{\iota_i}^{\,i} \to \iota \quad \overline{\Gamma; \alpha_i \vdash t_i \,:\, \iota_i; \alpha_{i-1}}^{\,i}}{\Gamma; \alpha_n \vdash op(\overline{t_i}^{\,i}) \,:\, \iota; \alpha_0}\ \text{T\_Op}$$

$$\frac{}{\Gamma; \alpha \vdash \mathsf{blame}\, p \,:\, A; \beta}\ \text{T\_Blame} \qquad \frac{\Gamma, x{:}A; \beta \vdash t \,:\, B; \gamma}{\Gamma; \alpha \vdash \lambda x.\, t \,:\, A/\beta \to B/\gamma; \alpha}\ \text{T\_Abs}$$

$$\frac{x{:}A \,\in\, \Gamma}{\Gamma; \alpha \vdash x \,:\, A; \alpha}\ \text{T\_Var} \qquad \frac{\Gamma; \gamma \vdash t \,:\, A/\alpha \to B/\beta; \delta \quad \Gamma; \beta \vdash s \,:\, A; \gamma}{\Gamma; \alpha \vdash t\, s \,:\, B; \delta}\ \text{T\_App}$$

$$\frac{\Gamma; \alpha \vdash s \,:\, A; \beta \quad A \,\sim\, B}{\Gamma; \alpha \vdash (s \,:\, A \Rightarrow^{p}\, B) \,:\, B; \beta}\ \text{T\_Cast} \qquad \frac{\Gamma; \alpha \vdash s \,:\, G; \beta}{\Gamma; \alpha \vdash (s \,:\, G \Rightarrow \star) \,:\, \star; \beta}\ \text{T\_Ground}$$

$$\frac{\Gamma; \alpha \vdash s \,:\, \star; \beta}{\Gamma; \alpha \vdash s\ \mathsf{is}\ G \,:\, \mathsf{bool}; \beta}\ \text{T\_Is} \qquad \frac{\Gamma, k{:}A/\gamma \to \alpha/\gamma; \delta \vdash s \,:\, \delta; \beta}{\Gamma; \alpha \vdash \mathcal{S}k.\, s \,:\, A; \beta}\ \text{T\_Shift}$$

$$\frac{\Gamma; \beta \vdash s \,:\, \beta; A}{\Gamma; \alpha \vdash \langle s \rangle \,:\, A; \alpha}\ \text{T\_Reset}$$

FIGURE 2.4: Typing rules.

The typing rule (T_BLAME) seems to allow blame to modify answer types to any type though blame does not invoke shift operator; this causes no problems (and is necessary for type soundness) because blame halts a program.

2.2.4 Type Soundness

We show type soundness of our calculus in the standard way: Progress and Preservation [121]. In the presence of the dynamic type, we can write a divergent term easily, and blame is a legitimate state of program evaluation. Thus, type soundness for our calculus means that any well typed program (a closed term enclosed by reset) evaluates to a well typed value, diverges, or raises blame. In what follows, we write $\longmapsto^*$ for the reflexive and transitive closure of $\longmapsto$.

Theorem 1 (Type Soundness). *If $\emptyset; \alpha \vdash \langle s \rangle \,:\, A; \alpha$, then one of the followings holds:*

- *there is an infinite evaluation sequence from $\langle s \rangle$;*
- *$\langle s \rangle \longmapsto^* \mathsf{blame}\, p$ for some p; or*
- *$\langle s \rangle \longmapsto^* v$ for some v such that $\emptyset; \alpha \vdash v \,:\, A; \alpha$.*

The outermost reset is assumed to exclude terms stuck at a shift operator without a surrounding reset. The statement of Progress shown before Preservation, however, has to take into account such a possibility for proof by induction to work.

Lemma 1 (Progress). *If $\emptyset; \alpha \vdash s \,:\, A; \beta$, then one of the followings holds:*

- *$s \longmapsto s'$ for some s';*
- *s is a value;*
- *$s \,=\, \mathsf{blame}\, p$ for some p; or*

- $s = F[\mathcal{S}k.\, t]$ *for some* F, k *and* t.

Proof. Straightforward by induction on the typing derivation. □

Lemma 2 (Preservation). *Suppose that* $\emptyset; \alpha \vdash s : A; \beta$.

(1) If $s \longrightarrow t$, *then* $\emptyset; \alpha \vdash t : A; \beta$.

(2) If $s \longmapsto t$, *then* $\emptyset; \alpha \vdash t : A; \beta$.

Proof. By induction on the typing derivation with case analysis on the reduction/evaluation rule applied to s. In the case for (R_SHIFT), we follow the proof in the previous work on shift/reset [9]. □

Proof of Theorem 1. By Progress and Preservation. Note that the evaluation from $\langle s \rangle$ to $F[\mathcal{S}k.\, t]$ as stated in Progress does not happen since s is enclosed by reset and reset does not appear in F. □

2.3 Blame Theorem

Blame Theorem intuitively states that values from the typed code will never be sources of cast failure at run time and, more specifically, clarifies conditions under which some blame never happens. Following the original work [120], we formalize such conditions using a few, different subtyping relations. Our proof is based on that in Ahmed et al.'s work [7], which defined a safety relation for terms and showed Blame Progress and Blame Preservation like progress and preservation for type soundness.

2.3.1 Subtyping

To state a Blame Theorem, we introduce naive subtyping $<:_n$, which formalizes the notion of being "more precisely typed." Roughly speaking, type A is a naive subtype of B when A is obtained by substituting some types for occurrences of the dynamic type in B. For example, $\mathsf{int} <:_n \star$ and $\mathsf{int}/\mathsf{int} \to \mathsf{int}/\mathsf{int} <:_n \star/\mathsf{int} \to \mathsf{int}/\star$. Note that argument types are covariant here. The Blame Theorem states that if type A is a naive subtype of type B, then the side of A is never blamed, that is, a cast $s : A \Rightarrow^p B$ does not cause blame p and $s : B \Rightarrow^p A$ does not blame $\bar{p}$.

To prove the Blame Theorem, we introduce positive and negative subtyping. Intuitively, that type A is a positive (resp. negative) subtype of B expresses that positive (resp. negative) blame never happens for a cast from A to B. It turns out that naive subtyping can be expressed in terms of positive and negative subtyping, from which the Blame Theorem easily follows. In addition, a cast from an ordinary subtype—where argument types of function types are contravariant—to a supertype is shown not to raise blame.

Subtyping relations—ordinary subtyping $<:$, naive subtyping $<:_n$, positive subtyping $<:^+$, and negative subtyping $<:^-$—are reflexive relations satisfying subtyping rules presented in Figure 2.5. The idea shared across all subtyping rules for function types is that function type $A/\alpha \to B/\beta$ is interpreted as if it takes the CPS-transformation form $A \to (B \to \alpha) \to \beta$. In this form, A and α occur at negative positions while B and β occur at positive positions.

We write $A <: B$ to denote that A is a subtype of B. The rule (S_DYN) means that any (nondynamic) type is a subtype of the dynamic type if it is a subtype of the (unique) ground type compatible to it. The premise is needed for cases that the subtype

$\boxed{A <: B}$ **Subtype**

$$\frac{}{A <: A}\ \text{S\_Refl} \qquad \frac{A <: G}{A <: \star}\ \text{S\_Dyn} \qquad \frac{A' <: A \quad B <: B' \quad \alpha' <: \alpha \quad \beta <: \beta'}{A/\alpha \to B/\beta <: A'/\alpha' \to B'/\beta'}\ \text{S\_Fun}$$

$\boxed{A <:_n B}$ **Naive Subtype**

$$\frac{}{A <:_n A}\ \text{SN\_Refl} \qquad \frac{}{A <:_n \star}\ \text{SN\_Dyn}$$

$$\frac{A <:_n A' \quad B <:_n B' \quad \alpha <:_n \alpha' \quad \beta <:_n \beta'}{A/\alpha \to B/\beta <:_n A'/\alpha' \to B'/\beta'}\ \text{SN\_Fun}$$

$\boxed{A <:^+ B}$ **Positive Subtype**

$$\frac{}{A <:^+ A}\ \text{S}^+\text{\_Refl} \qquad \frac{}{A <:^+ \star}\ \text{S}^+\text{\_Dyn}$$

$$\frac{A' <:^- A \quad B <:^+ B' \quad \alpha' <:^- \alpha \quad \beta <:^+ \beta'}{A/\alpha \to B/\beta <:^+ A'/\alpha' \to B'/\beta'}\ \text{S}^+\text{\_Fun}$$

$\boxed{A <:^- B}$ **Negative Subtype**

$$\frac{}{A <:^- A}\ \text{S}^-\text{\_Refl} \qquad \frac{}{\star <:^- A}\ \text{S}^-\text{\_Dyn} \qquad \frac{A <:^- G}{A <:^- B}\ \text{S}^-\text{\_Any}$$

$$\frac{A' <:^+ A \quad B <:^- B' \quad \alpha' <:^+ \alpha \quad \beta <:^- \beta'}{A/\alpha \to B/\beta <:^- A'/\alpha' \to B'/\beta'}\ \text{S}^-\text{\_Fun}$$

FIGURE 2.5: Subtyping rules.

is higher order. Function types are covariant at positive positions and contravariant at negative positions as usual.

As mentioned before, type A is a naive subtype of B when A is obtained by putting some types in occurrences of the dynamic type in B. The rule (SN_Dyn) means that the dynamic type is least precise. In the rule (SN_Fun), function types for naive subtyping are covariant in both positive and negative positions.

The definitions of positive and negative subtyping are mutually recursive. The rule (S^+_Dyn) means that positive blame never happens when any value is coerced to the dynamic type. Similarly to ordinary subtyping, in (S^+_Fun), function types are covariant at positive positions and contravariant at negative positions. Negative subtyping is a reversed version of positive subtyping except for addition of (S^-_Any), which is a combination of (S^-_Dyn) and the fact that a cast from type A to the dynamic type never gives rise to negative blame when A is a negative subtype of its ground type. The rule (S^-_Any) follows from Ahmed et al.'s work [7] and represents a relaxed form of the system of Wadler and Findler [120]. Notice that polarity of subtyping is reversed at negative positions.

As mentioned above, we show that naive subtyping (and ordinary subtyping) can be expressed in terms of positive and negative subtyping.

Lemma 3. *If $A/\alpha \to B/\beta <:^- G$, then $A = \alpha = \star$ and $B <:^- \gamma$ and $\beta <:^- \gamma$ for any γ.*

Lemma 4. *$A <:_n B$ iff $A <:^+ B$ and $B <:^- A$.*

$$\frac{s\ \mathsf{sf}\ p \quad A <:^{+} B}{s\ :\ A \Rightarrow^{p} B\ \mathsf{sf}\ p}\ \text{SF\_POS} \qquad \frac{s\ \mathsf{sf}\ p \quad A <:^{-} B}{s\ :\ A \Rightarrow^{\bar{p}} B\ \mathsf{sf}\ p}\ \text{SF\_NEG} \qquad \frac{}{c\ \mathsf{sf}\ p}\ \text{SF\_CONST}$$

$$\frac{\forall i.\ t_i\ \mathsf{sf}\ p}{op(\overline{t_i}^{\,i})\ \mathsf{sf}\ p}\ \text{SF\_OP} \qquad \frac{}{x\ \mathsf{sf}\ p}\ \text{SF\_VAR} \qquad \frac{s\ \mathsf{sf}\ p}{\lambda x.\, s\ \mathsf{sf}\ p}\ \text{SF\_ABS} \qquad \frac{s\ \mathsf{sf}\ p \quad t\ \mathsf{sf}\ p}{s\ t\ \mathsf{sf}\ p}\ \text{SF\_APP}$$

$$\frac{q \neq p \quad q \neq \bar{p} \quad s\ \mathsf{sf}\ p}{s\ :\ A \Rightarrow^{q} B\ \mathsf{sf}\ p}\ \text{SF\_CAST} \qquad \frac{s\ \mathsf{sf}\ p}{s\ :\ G \Rightarrow \star\ \mathsf{sf}\ p}\ \text{SF\_GROUND} \qquad \frac{s\ \mathsf{sf}\ p}{s\ \mathsf{is}\ G\ \mathsf{sf}\ p}\ \text{SF\_IS}$$

$$\frac{q \neq p}{\mathsf{blame}\ q\ \mathsf{sf}\ p}\ \text{SF\_BLAME} \qquad \frac{s\ \mathsf{sf}\ p}{\mathcal{S}k.\, s\ \mathsf{sf}\ p}\ \text{SF\_SHIFT} \qquad \frac{s\ \mathsf{sf}\ p}{\langle s \rangle\ \mathsf{sf}\ p}\ \text{SF\_RESET}$$

FIGURE 2.6: Safety rules.

Lemma 5. *$A <: B$ iff $A <:^{+} B$ and $A <:^{-} B$.*

The proofs of the direction from left to right are straightforward by induction on the derivations of $A <:_n B$ and $A <: B$. The other direction is shown by structural induction on A with Lemma 3.

2.3.2 Blame Theorem

The proof of the Blame Theorem is similar to progress and preservation for type soundness. Instead of a type system, we introduce a safety relation using positive and negative subtyping and show Blame Progress, which states that a safe term does not give rise to blame, and Blame Preservation, which states safety is preserved by evaluation. In this section, we focus only on whether a term gives rise to blame or not and not on whether a term gets stuck or not.

A term s is *safe for blame label* p, written as $s\ \mathsf{sf}\ p$, if every cast with blame label p in s is from a type to its positive supertype and every cast with $\bar{p}$ is from a type to its negative supertype. We present inference rules for the safety relation in Figure 2.6. From the definition, it is observed that a term safe for p does not contain blame with p; this does not restrict a source program written by a programmer because it should not contain any blame.

Blame Progress and Blame Preservation show that, if $s\ \mathsf{sf}\ p$, term s never gives rise to blame with label p. We write $s \mapsto\!\!\!\!\!\!\rightarrow\ t$ and $s \mapsto\!\!\!\!\!\!\rightarrow^{*}\ t$ to denote that term s does not reduce to term t in a single step and in multiple steps, respectively.

Lemma 6 (Blame Progress)**.** *If $s\ \mathsf{sf}\ p$, then $s \mapsto\!\!\!\!\!\!\rightarrow\ \mathsf{blame}\ p$.*

Proof. Straightforward by induction on the derivation of $s\ \mathsf{sf}\ p$. □

Lemma 7 (Blame Preservation)**.** *(1) If $s\ \mathsf{sf}\ p$ and $s \longrightarrow t$, then $t\ \mathsf{sf}\ p$.*

(2) If $s\ \mathsf{sf}\ p$ and $s \longmapsto t$, then $t\ \mathsf{sf}\ p$.

Proof. By induction on the derivation of $s\ \mathsf{sf}\ p$ with case analysis on the reduction/evaluation rule applied to s. In the case for (R_FUN), we use Lemma 3 for (S^{-}_ANY). □

Finally, we show the Blame Theorem—values that flow from the more precisely typed side never cause blame—and, furthermore, that casts from one type to its supertype never give rise to blame.

Theorem 2 (Blame Theorem and Subtype Theorem). *Let s be a term with a subterm $t : A \Rightarrow^p B$ where cast is labeled by the only occurrence of p in s. Moreover, suppose that $\bar{p}$ does not appear in s.*

(1) If $A <:^+ B$, then $s \mapsto\!\!\!\!\!\rightarrow^* \mathsf{blame}\, p$.

(2) If $A <:^- B$, then $s \mapsto\!\!\!\!\!\rightarrow^* \mathsf{blame}\, \bar{p}$.

(3) If $A <:_n B$, then $s \mapsto\!\!\!\!\!\rightarrow^* \mathsf{blame}\, p$; if $B <:_n A$, then $s \mapsto\!\!\!\!\!\rightarrow^* \mathsf{blame}\, \bar{p}$.

(4) If $A <: B$, then $s \mapsto\!\!\!\!\!\rightarrow^* \mathsf{blame}\, p$ and $s \mapsto\!\!\!\!\!\rightarrow^* \mathsf{blame}\, \bar{p}$.

Proof. The first and second cases are shown by Blame Progress and Blame Preservation because $s\ \mathsf{sf}\ p$ in the first case and $s\ \mathsf{sf}\ \bar{p}$ in the second case. The third case (resp. the fourth case) follows from the first and second cases and Lemma 4 (resp. Lemma 5). □

2.4 CPS Transformation

The semantics of programming languages with control operators has often been established by transformation of programs with control operators to continuation passing style (CPS), a programming style where continuations appear in a program as arguments of functions. For example, programs with Reynolds's escape operator [88], call/cc in Scheme, shift/reset [25], and so on can be transformed to CPS form.

As a proof of correctness of our approach, we define a CPS transformation from terms in our calculus to those in the simply typed blame calculus of Ahmed et al. [7] and show that a well typed source term is transformed to a well typed target term and, for any source terms such that one reduces to the other, their CPS-transformation results are equivalent in the target calculus. The equational system is based on call-by-value axioms [93] due to blame, which is effectful.

Before giving the CPS transformation, we modify the syntax and the reduction rule (R_GROUND) of our calculus slightly in order to transform a ground value of the form $v : \star/\star \to \star/\star \Rightarrow \star$ to a value with a cast (the reason is detailed later). To assign a blame label to the cast, the syntax is changed as follows:

$$v ::= \ldots \mid \quad v : G \Rightarrow_p \star \qquad s ::= \ldots \mid \quad s : G \Rightarrow_p \star$$

Blame labels in ground terms and values are given as subscripts for ease of distinction from casts. The reduction rule (R_GROUND) takes the following form:

$$v : A \Rightarrow^p \star \longrightarrow (v : A \Rightarrow^p G) : G \Rightarrow_p \star \ (\text{if } A \sim G \text{ and } A \neq \star) \quad \text{R\_GROUND}$$

Our CPS transformation, which mostly follows Danvy and Filinski [25], is shown in Figure 2.7 in three parts: transformation for types, values, and terms. We use variable κ to denote continuations. The CPS transformation for types is standard. A function of type $A/\alpha \to B/\beta$ takes an argument of A, would pass a value of B to a continuation that returns α, and results in a value of β as the computation result. The CPS transformation for values maps values in our calculus to those in the blame calculus without shift/reset. The definition shown in Figure 2.7 is easy to understand except for ground values where the ground type is a function type. We might expect that the CPS-transformation result of ground value $v : G \Rightarrow_p \star$ can be defined as $v^* : [\![G]\!] \Rightarrow \star$. However, that form would not be a valid term in the target calculus if the ground type G is a function type, because the ground function type in the target calculus takes only

$\boxed{[\![A]\!]}$ **CPS Transformation (Types)**

$$[\![\iota]\!] = \iota \qquad [\![\star]\!] = \star \qquad [\![A/\alpha \to B/\beta]\!] = [\![A]\!] \to ([\![B]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$$

$\boxed{v^*}$ **CPS Transformation (Values)**

$$x^* = x \qquad c^* = c \qquad (\lambda x.\, s)^* = \lambda x.\, [\![s]\!] \qquad (v : \iota \Rightarrow \star)^* = v^* : \iota \Rightarrow \star$$
$$(v : \star/\star \to \star/\star \Rightarrow_p \star)^* = (\lambda x.\, (v^*\, x) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star$$

$\boxed{[\![s]\!]}$ **CPS Transformation (Terms)**

$$\begin{array}{lcl}
[\![v]\!] & = & \lambda\kappa.\, \kappa\, v^* \\
[\![op(\overline{t_i}^{\,i})]\!] & = & \lambda\kappa.\, [\![t_1]\!]\, (\lambda x_1.\, \ldots\, [\![t_n]\!]\, (\lambda x_n.\, \kappa\, op(\overline{x_i}^{\,i}))\, \ldots) \\
[\![s\ t]\!] & = & \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, [\![t]\!]\, (\lambda y.\, x\ y\ \kappa)) \\
[\![\langle s \rangle]\!] & = & \lambda\kappa.\, \kappa\, ([\![s]\!]\, (\lambda x.\, x)) \\
[\![\mathcal{S}k.\, s]\!] & = & \lambda\kappa.\, ([\![s]\!]\, (\lambda x.\, x))\, [k := \lambda x.\, \lambda\kappa'.\, \kappa'\, (\kappa\, x)] \\
[\![s : A \Rightarrow^p B]\!] & = & \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, \kappa\, (x : [\![A]\!] \Rightarrow^p [\![B]\!])) \\
[\![s : G \Rightarrow \star]\!] & = & \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, \kappa\, (x : G \Rightarrow \star)^*) \\
[\![s \,\mathsf{is}\, \iota]\!] & = & \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, \kappa\, (x \,\mathsf{is}\, \iota)) \\
[\![s \,\mathsf{is}\, (\star/\star \to \star/\star)]\!] & = & \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, \kappa\, (x \,\mathsf{is}\, (\star \to \star))) \\
[\![\mathsf{blame}\, p]\!] & = & \lambda\kappa.\, \mathsf{blame}\, p
\end{array}$$

FIGURE 2.7: CPS transformation.

the form $\star \to \star$ but $[\![\star/\star \to \star/\star]\!] = \star \to (\star \to \star) \to \star$. Expecting a value will be translated to a value in the target calculus, we set a ground value $v : \star/\star \to \star/\star \Rightarrow_p \star$ to be mapped to *a value to which* $v^* : [\![G]\!] \Rightarrow^p \star$ *reduces*, instead. (Notice the superscript on $\Rightarrow$. A term $v^* : [\![G]\!] \Rightarrow^p \star$ is a cast and always valid.) In the result, we omit the trivial cast $x : \star \Rightarrow^{\bar{p}} \star$. The CPS transformation for terms is self-explanatory. It is worth noting that, for type tests, there are two cases on tested types. The case that a tested type is a base type is trivial. The other case translates function type $\star/\star \to \star/\star$ with answer types to the function type $\star \to \star$, where the type of continuations is not presented, because the simply typed blame calculus does not support type tests with higher-order types and, more unfortunately, we cannot investigate that a value of the dynamic type would take functions as an argument (recall $[\![\star/\star \to \star/\star]\!] = \star \to (\star \to \star) \to \star$) in general. Although this treatment of type tests with function types causes no problems in this work, it would be problematic when we consider inverse of the CPS transformation as in completeness of axiomatization [57].

It is straightforward to show that well typed source terms are transformed to well typed target terms. For any typing context Γ, we write $[\![\Gamma]\!]$ for the typing context obtained by applying the CPS transformation to types mapped by Γ.

Theorem 3 (Preservation of Type)**.** *If* $\Gamma; \alpha \vdash s : A; \beta$*, then* $[\![\Gamma]\!] \vdash [\![s]\!] : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$.

Next, we define an equational system in the target calculus. The system consists of axioms about casts as well as usual call-by-value axioms [93]. In what follows, we use metavariables $\mathbb{e}$, $\mathbb{v}$, $\mathbb{E}$, and $\mathbb{A}$ (and $\mathbb{B}$) to denote terms, values, evaluation contexts, and

types in the target calculus, respectively, and write $fv\,(\mathbb{v})$ and $fv\,(\mathbb{E})$ for the sets of free variables in $\mathbb{v}$ and $\mathbb{E}$, respectively. In addition, let the relation $\Longrightarrow$ be the evaluation relation in the target calculus.

Definition 1 (Term Equality)**.** *The relation $\approx$ is the least congruence that contains the following axioms:*

$$\frac{\mathbb{e}_1 \Longrightarrow \mathbb{e}_2}{\mathbb{e}_1 \approx \mathbb{e}_2} \qquad \frac{x \notin fv\,(\mathbb{v})}{\lambda x.\,\mathbb{v}\; x \approx \mathbb{v}} \qquad \frac{x \notin fv\,(\mathbb{E})}{(\lambda x.\,\mathbb{E}[x])\,\mathbb{e} \approx \mathbb{E}[\mathbb{e}]}$$

$$\mathbb{e} : \star \Rightarrow^p \star \approx \mathbb{e} \qquad \mathbb{e} : \star \Rightarrow^p \star \to \star \Rightarrow^p \mathbb{A} \to \mathbb{B} \approx \mathbb{e} : \star \Rightarrow^p \mathbb{A} \to \mathbb{B}$$

We think that the last two axioms about casts are reasonable. The former, which skips the trivial cast, is found in another blame calculus [103]. This axiom is introduced mainly to ignore redundant casts that often happen in CPS-transformation results. The latter axiom, which collapses two casts into one, is used to show terms reduced by (R_COLLAPSE) are equivalent after CPS transformation. The latter might be unnecessary if our calculus was able to investigate structures of values of the dynamic type as Abadi et al. [1], but we leave it as future work.

Now, we show that the relationship between our semantics in direct-style and the CPS transformation.

Theorem 4 (Preservation of Equality)**.** *If $s \longmapsto t$, then $[\![s]\!] \approx [\![t]\!]$.*

Finally, we remark on (R_DYN). In fact, although we first had tried to show Theorem 4 without (R_DYN), we could not. Without (R_DYN), we have to show that the transformation results of $v : G \Rightarrow_q \star \Rightarrow^p \star$ and $v : G \Rightarrow^p \star$ are equivalent because the side condition $A \neq \star$ in (R_COLLAPSE) is not needed [7] and then the former would reduce to the latter. Unfortunately, the results are not equivalent in our equational system because the former refers to label q but the latter does not. We consider that there is room for improvement of the CPS transformation, the equational system, and the proof of soundness of the transformation in this thesis; it is left as future work.

Chapter 3

Manifest Contracts with Parametric Polymorphism

This chapter studies and establishes foundations of manifest contracts with type abstraction based on *parametric polymorphism*. Parametric polymorphism, which was introduced by Girard [41] and Reynolds [89, 90] independently, is a key concept to reuse program components and a basis of abstract types [78] and program reasoning [119]. We start with seeing that manifest contracts are suited with type-based abstraction mechanisms based on parametric polymorphism through an example with abstract datatypes (ADTs) and then describe our contributions.

A motivating example To motivate the combination of contracts and ADTs, consider the interface of an ADT modeling the natural numbers, written in an ML-like language:

```
module type NAT =
sig
   type t
   val zero : t
   val succ : t -> t
   val isZ  : t -> bool
   val pred : t -> t
end
```

It is an *abstract* datatype because the actual representation of `t` is hidden: users of `NAT` interact with it through the constructors and operations provided. The `zero` constructor represents 0; the `succ` constructor takes a natural and produces its successor. The predicate `isZ` determines whether a given natural is zero. The `pred` operation takes a natural number and returns its predecessor.

This interface, however, is not fine-grained enough to prevent misuse of partial operations. For example, `pred` can be applied to `zero`, whereas the mathematical natural-number predecessor operation is not defined for zero.

Using (refinement types with) contracts, we can explicitly specify the constraint that an argument to `pred` is not zero:

```
module type NAT =
sig
   type t
   val zero : t
   val succ : t -> t
   val isZ  : t -> bool
   val pred : {x:t | not (isZ x)} -> t
end
```

The type `{x:t | not (isZ x)}` is a refinement that denotes the set of values `x` such that `not (isZ x)` evaluates to true. So, this new interface does not allow `pred` to be applied to zero.

Polymorphic manifest contract calculus In fact, our work is not the first to combine manifest contracts and parametric polymorphism. Gronski et al. have studied manifest contracts in the presence of polymorphism by developing SAGE language [46], which supports manifest contracts and polymorphism, in addition to the dynamic type [1, 51, 101] and even the so-called "Type:Type" discipline [19]. However, consequences of combining these features, in particular, interactions between manifest contracts and type abstraction (provided by parametric polymorphism), are not studied in depth in Gronski et al. [46].

To study type abstraction for manifest contracts rigorously, Belo et al. [14] developed a polymorphic manifest contract calculus F_{H}, an extension of System F with manifest contracts, and investigated its properties, including type soundness and (syntactic) parametricity. For F_{H} to scale up to polymorphism, they made two technical contributions to diverge from earlier manifest calculi such as λ_{H} [37], a simply typed manifest contract calculus. First, F_{H} gives the semantics of casts in the presence of so-called "general refinements," where the underlying type T in a refinement type $\{x{:}T \mid e\}$ can be an arbitrary type (not only base types like bool and int but also function, forall, and even refinement types), when earlier manifest calculi restrict refinements to base types. Support for general refinements is important because it means that an abstract datatype can be implemented by any type. SAGE also allows arbitrary types to be refined but the semantics of casts relies on the dynamic type, which is problematic for parametricity [69]. Second, Belo et al. have proposed a new, two-step, syntactic approach to formalizing manifest calculi. The first step is to establish fundamental properties such as type soundness for a calculus *without subsumption* (and subtyping), while earlier calculi [64, 44] rest on subtyping and denotational semantics of types in their construction. Technically, they replaced subtyping with a syntactic type conversion relation, which is required to show preservation in the presence of dependent function types. The lack of subsumption allows for an entirely syntactic metatheory but it also amounts to the lack of static contract checking. The second step is to give static analysis to remove casts that never fail in order to compensate the lack of static contract checking. In fact, Belo et al. give "post facto" subtyping and examine a property called Upcast Lemma, which says an upcast—a cast from one type to a supertype—is logically related (thus equivalent in a certain sense) to an identity function, as a correctness property of static contract checking.

Unfortunately, however, the proofs of type soundness and parametricity of F_{H} turn out to be flawed and, worse, the properties themselves are later found to be false. In fact, the type conversion makes an inconsistent contract system; if a cast-free closed expression is well typed, then its type can be refined arbitrarily—e.g., integer 0 can be given type $\{x{:}\mathsf{int} \mid x = 42\}$! These anomalies are first recognized as a false lemma about the type conversion relation. Greenberg [43] fixed the false lemma by changing the conversion relation. Another key property, called *cotermination* and left as a conjecture in both Belo et al. [14] and Greenberg [43], also turns out to be wrong.[1] Inconsistency and failure of type soundness and parametricity follow from counterexamples to these

[1] In the end of Section 4 of Belo et al. [14], the authors write "our proof of type soundness in Section 3 relies on much simpler properties of parallel reduction, which we *have* proved." as if the type soundness proof did not depend on cotermination, but this claim also turns out to be false.

	Belo et al. [14] (F_H)	Greenberg [43]	Our work (F^{σ}_H)
Lemma w.r.t. convertibility	✗	✓	✓
Cotermination	✗ (conjecture)	✗ (conjecture)	✓
Progress	✗	✗	✓
Preservation	$\checkmark^?$	$\checkmark^?$	✓
Parametricity	✗	✗	✓
Upcast Lemma	$\checkmark^?$	$\checkmark^?$	?

✓ ··· proved $\checkmark^?$ ··· proved with flawed premises ✗ ··· flawed ? ··· unknown

TABLE 3.1: The status of properties of polymorphic manifest calculi.

properties. As we will discuss in detail, the root cause of the problem can be attributed to the fact that substitution can badly affect how casts behave.

Contributions In this chapter, we introduce a new polymorphic manifest contract calculus F^{σ}_H that resolves the technical flaws in F_H. We call our calculus F^{σ}_H because it takes the F_H from Belo et al. [14] and Greenberg [43] and introduces a new substitution semantics using *delayed substitutions*, which we write σ. Delayed substitutions are close to explicit substitutions [2] but only substitutions on casts are explicit (and delayed) in F^{σ}_H. Although, in some work [47, 7], delayed substitutions, also called explicit bindings, have been used to represent syntactic "barriers" for type abstractions, we rather use them to determine how casts reduce statically. Thanks to delayed substitution, the semantics of F^{σ}_H can choose cast reduction rules independently of substitution; this property is crucial when we prove cotermination. We can finally show that type soundness and parametricity all hold in F^{σ}_H—*without leaving any conjectures*. Consistency of the contract system of F^{σ}_H is derived immediately from type soundness.

Table 3.1 summarizes the status of properties of polymorphic manifest calculi; the columns and rows represent properties and work on polymorphic manifest contracts, respectively. We wrote ✓ for properties that are proved, $\checkmark^?$ for properties with proofs that are based on false premises, ✗ for properties that are flawed, and ? for properties we are unsure of. We have not investigated the Upcast Lemma in F^{σ}_H because the first step of Belo et al.'s approach—namely, establishing fundamental properties for a manifest calculus without subsumption (hence static contract checking)—has turned out to be trickier than we initially thought and is worth independent treatment. The value of the Upcast Lemmas in Belo et al. [14] and Greenberg [43] is questionable, due to the flaws on which the proofs of type soundness and parametricity rest, though the properties themselves may still hold.

Outline The rest of this chapter is organized as follows. We start Section 3.1 with a brief history of manifest contract calculi (both monomorphic and polymorphic) and discuss their technical issues and our solutions. Section 3.2 defines F^{σ}_H. We prove type soundness in Section 3.3, fixing Belo et al. [14] with common-subexpression reduction from Greenberg [43] and our novel use of delayed substitutions. We prove parametricity in Section 3.4; along with the proofs of cotermination and type soundness in the prior section, this constitutes the first conjecture-free metatheory for the combination of System F and manifest contracts, resolving issues in prior versions of F_H. Section 3.5

compares F^{σ}_{H} with two variants of polymorphic manifest contracts [14, 43] and presents counterexamples to broken properties in these earlier calculi.

3.1 Overview

This section first reviews manifest contract calculi [37, 44, 64]—proposed as foundations of *hybrid type checking*, a synthesis of static and dynamic specification checking—and earlier polymorphic extensions [14, 43] with their technical challenges; then we describe problems in the earlier polymorphic calculi and our solutions.

3.1.1 Manifest Contract Calculus for Hybrid Type Checking

Flanagan [37] proposed hybrid type checking, a framework to combine static and dynamic verification techniques for modularly checking implementations against contract-based precise interface specifications, and formalized λ_{H} as a theoretical foundation to study hybrid type checking. Later work revised and refined those early ideas [64, 44], named the core dynamic checking framework a 'manifest contract calculus' (or simply, manifest calculus) [44].

Hybrid type checking reduces program verification to subtype checking problems, solving them statically as much as possible and deferring checking to run time if a problem instance is not solved statically. We describe how these ideas are formalized in λ_{H} below; briefly, characteristic features of manifest contract calculi (in particular, early ones such as slightly different versions of λ_{H}) could be summarized as:

- *Type-based specifications:* refinement types (and dependent function types) to represent specifications;
- *Static checking:* subtyping to model static verification; and
- *Dynamic checking:* casts to model dynamic verification.

Type-based specifications In λ_{H}, specifications are expressed in terms of types, more concretely, *refinement types* and *dependent function types*. A refinement type $\{x{:}B \mid e\}$ intuitively denotes the set of values v of base type B (e.g., int, bool, and so on) such that $[v/x]e$ reduces to true. In that type, e, also called a contract or a refinement, can be an *arbitrary* Boolean expression, so refinement types can represent any subset of the base-type constants as long as a constraint to specify the subset can be written as a program expression. For example, prime numbers can be represented as $\{x{:}\mathsf{int} \mid \mathsf{prime?}\ x\}$, using a primality test function $\mathsf{prime?}$. A dependent function type $x{:}T_1 \to T_2$ denotes functions taking arguments v of domain type T_1 and returning values of codomain type $[v/x]T_2$. Dependent functions cleanly express the relation between inputs and outputs of a function. For example, $x{:}\mathsf{int} \to \{y{:}\mathsf{int} \mid y > x\}$ denotes functions that return an integer larger than the argument.

Manifest calculi need not have *arbitrary* Boolean expressions and dependent function types. For example, Ou et al. [82] restrict predicates to be pure expressions and the blame calculus by Wadler and Findler [120] supports only non-dependent function types. As we will discuss below, having arbitrary predicates and dependent functions significantly complicates metatheory. We will call a manifest calculus with both of these optional features a *full* manifest calculus.

Static checking With these expressive types, program verification amounts to type checking, in particular, checking subtyping between refinement types. For example, to see if a prime number (of type $\{x{:}\mathsf{int} \mid \mathsf{prime?}\, x\}$) can be safely passed to a function expecting positive numbers (of type $\{x{:}\mathsf{int} \mid x > 0\}$) is to see if the former type is a subtype of the latter. Informally, a refinement type $\{x{:}B \mid e_1\}$ is a subtype of $\{x{:}B \mid e_2\}$ when e_2 holds for any value of B satisfying e_1. Formally, supposing that we use σ to denote substitutions and write $\Gamma, x{:}\{x{:}B \mid \mathsf{true}\} \vdash \sigma$ to mean that σ is a closing substitution respecting $(\Gamma, x{:}\{x{:}B \mid \mathsf{true}\})$, Flanagan gives a subtyping rule for refinement types like:[2]

$$\frac{\forall\sigma.\ (\Gamma, x{:}\{x{:}B \mid \mathsf{true}\} \vdash \sigma \wedge \sigma(e_1) \longrightarrow^* \mathsf{true})\ \mathsf{implies}\ \sigma(e_2) \longrightarrow^* \mathsf{true}}{\Gamma \vdash \{x{:}B \mid e_1\} <: \{x{:}B \mid e_2\}}$$

This formalization allows language designers to choose their favorite static checking methods because it states *what* static checking verifies, rather than how a specific static checking method works.

Dynamic checking Unlike previous work on refinement types [40, 123, 67, 82], however, the predicate language is very expressive—in fact, too expressive to be decidable. Flanagan's approach to undecided subtyping is to defer subtyping check at run time by inserting casts to where subtyping cannot be decided, rather than reject a program. More concretely, if static checking cannot decide whether the type T_1 of a given expression e is a subtype of T_2, then the compiler inserts a cast—written $\langle T_1 \Rightarrow T_2 \rangle^l$—from T_1 (called source type) to T_2 (called target type) and yields $\langle T_1 \Rightarrow T_2 \rangle^l\, e$. At run time, it is checked whether (the value of) e can behave as T_2. The superscript l is called a *blame label*, an abstract source location used to differentiate between different casts and identify the source of failures; unlike the blame calculus in Chapter 2, we do not distinguish positive and negative blame in this chapter because we are not interested in a theory of blame here.

We briefly explain how casts work in simple cases. At refinement types, casts either return the value they are applied to, or abort program execution by raising blame, which indicates that the supposed subtyping turns out to be false. For example, consider a cast from positive integers $\{x{:}\mathsf{int} \mid x > 0\}$ to odd integers $\{x{:}\mathsf{int} \mid \mathsf{odd}\, x\}$. If we apply cast $\langle \{x{:}\mathsf{int} \mid x > 0\} \Rightarrow \{x{:}\mathsf{int} \mid \mathsf{odd}\, x\} \rangle^l$ to 5, we expect to get 5 back, since 5 is an odd integer (that is, $\mathsf{odd}\, 5$ evaluates to true). So,

$$\langle \{x{:}\mathsf{int} \mid x > 0\} \Rightarrow \{x{:}\mathsf{int} \mid \mathsf{odd}\, x\} \rangle^l\, 5 \longrightarrow^* 5.$$

Then, 5 can be typed at $\{x{:}\mathsf{int} \mid \mathsf{odd}\, x\}$. On the other hand, suppose we apply the same cast to 2. This cast fails, since 2 is even. When the cast fails, it will raise blame with its label:

$$\langle \{x{:}\mathsf{int} \mid x > 0\} \Rightarrow \{x{:}\mathsf{int} \mid \mathsf{odd}\, x\} \rangle^l\, 2 \longrightarrow^* \Uparrow l.$$

Casts between dependent function types are also made possible in λ_{H} by adapting higher-order contracts by Findler and Felleisen [35].

Type soundness of λ_{H} Proving syntactic type soundness of a full calculus (such as λ_{H}) via progress and preservation is tricky. We identify two main issues here.

[2] Readers familiar with the systems will recognize that we have folded the implication judgment into the relevant subtyping rule.

The first issue is how to allow values to be typed at refinements they satisfy. For example, the type system should be able to give integer 2 type $\{x{:}\mathsf{int} \mid \mathsf{true}\}$, $\{x{:}\mathsf{int} \mid \mathsf{even}\, x\}$, or $\{x{:}\mathsf{int} \mid \mathsf{prime?}\, x\}$. Subtyping resolves it with the help of "selfified" types [82], which are *most specific types* of constants—e.g., the selfified type of integer n is $\{x{:}\mathsf{int} \mid x = n\}$. For example, if $\langle\{x{:}\mathsf{int} \mid \mathsf{true}\} \Rightarrow \{x{:}\mathsf{int} \mid x > 0\}\rangle^l\, n \longrightarrow^* n$, then n can be given type $\{x{:}\mathsf{int} \mid x > 0\}$ by using the subtyping rule above because inhabitants of the selfified type of n are only n and the dynamic check has ensured that $n > 0$ holds.

The second issue is standard in a typed calculus with dependent function types: if e_1 evaluates to e_2, the type system must allow terms of $[e_1/x]\,T$ to be typed at $[e_2/x]\,T$, too, and vice versa to show preservation. Let us consider the case for a function application $v_1\, e_2 \longrightarrow v_1\, e_2'$. Since v_1 is at a function position, its type takes the form $x{:}T_1 \to T_2$. The return type of a function is dependent on an argument to the function, so types of $v_1\, e_2$ and $v_1\, e_2'$ would be $[e_2/x]\,T_2$ and $[e_2'/x]\,T_2$, respectively. Since preservation says that evaluation preserves types of well typed terms, $v_1\, e_2'$ has to be typed also at $[e_2/x]\,T_2$.

A typical solution found in dependent type theory [26, 13, 50] is to introduce a type equivalence relation, which is congruence closed under (β or sometimes $\beta\eta$) reduction. Ou et al. [82] address this issue with subtyping; they show that, for any pure expressions e_1 and e_2, if $e_1 \longrightarrow e_2$, then $[e_2/x]\,T$ is a subtype of $[e_1/x]\,T$. It is not clear, however, how $[e_1/x]\,T$ and $[e_2/x]\,T$ should be related in a full manifest calculus mainly due to the above-mentioned subtyping rule for refinement types and the fact that computation is effectful (recall that blame is an uncatchable exception). Unfortunately, earlier work is not fully satisfactory in this regard. In fact, both Flanagan [37] and Knowles and Flanagan [64] do not discuss this issue and Greenberg et al. [44] sidestep it by showing only *semantic type soundness* using a logical predicate technique, which is motivated by another reason—see Section 3.1.2. (Knowles and Flanagan [64] and Greenberg et al. [44] prove, though, a closely related property that, if $e_1 \longrightarrow e_2$, then $[e_1/x]\,T$ and $[e_2/x]\,T$ are *semantic* subtypes of each other.)

In short, there is no fully satisfactory proof of syntactic type soundness of a full manifest calculus. Semantic type soundness is fine but it will be hard to extend if more features are added to the calculus. Thus, a more syntactic proof is desirable. In fact, Belo et al. [14] have attacked this problem of proving type soundness in a more syntactic manner when they extend a manifest calculus to parametric polymorphism.

3.1.2 Polymorphic Manifest Contract Calculus $\mathrm{F_H}$

A full manifest calculus $\mathrm{F_H}$ [14] has been developed to study type abstraction provided by parametric polymorphism in manifest contracts. Parametric polymorphism is a cornerstone of reusability in functional programming. For example, polymorphism can encode existentials, which are crucial for defining abstract datatypes and expressing modularity. In this context, manifest contracts are also used to specify precise interfaces of modules by refining existentials, as we discussed in the beginning of this chapter. This section describes key ideas in that work, namely refinement types with arbitrary underlying types and subsumption-free formalization, and the next presents technical flaws in the metatheory of $\mathrm{F_H}$.

Polymorphism and general refinements Adding polymorphism to manifest contracts is not as simple as it might appear. The crux of the matter is this: we need to be able to write $\{x{:}\alpha \mid e\}$ for refinements to interact with abstract datatypes in a useful way. A question here is: What types can be instantiated for the type variable α? Earlier

manifest calculi restrict refinements to base types, forbidding refinements of function types like $\{f{:}(\mathsf{int}\to\mathsf{int}) \mid f\ 0 = 0\}$. However, this restriction is severe and limits expressiveness of types excessively. For example, let us consider implementing the abstract datatype for natural numbers in the beginning of this chapter by using the Church encoding. Since the natural number type is $\forall\alpha.\alpha\to(\alpha\to\alpha)\to\alpha$, predecessor function `pred` over naturals has to be implemented as a function of type

$$\{x{:}\forall\alpha.\alpha\to(\alpha\to\alpha)\to\alpha \mid \mathsf{not}\,(\mathsf{isZ}\ x)\}\to(\forall\alpha.\alpha\to(\alpha\to\alpha)\to\alpha),$$

in which, to restrict arguments to be nonzero, the domain type refines the Church natural number type $\forall\alpha.\alpha\to(\alpha\to\alpha)\to\alpha$ by substituting it for the abstract type but this type is *ill-formed* because the underlying type is not a base type.

$\mathrm{F_H}$ supports *general refinements*, which allow type variables α to be instantiated with *any* type, that is, not only base types like bool and int but also function, forall, and even refinement types. Introducing general refinements calls for a new semantics for casts: how do casts evaluate? A cast $\langle T_1 \Rightarrow T_2\rangle^l$ evaluates in several steps (we describe it in detail in Section 3.2). Roughly speaking, the semantics forgets refinements in T_1 and then starts checking refinements in T_2 from the inside out. The cast semantics of $\mathrm{F_H}$ skips some refinement checks that appear to be unnecessary. For example, reflexive casts of the form $\langle T \Rightarrow T\rangle^l$ just disappear—this is motivated by parametricity: $\langle\alpha \Rightarrow \alpha\rangle^l$ should behave the same whatever the type variable α is bound to and the only reasonable behavior seems to disappear like the identity function.

As we mentioned in the beginning of this chapter, SAGE also allows any type to be refined; however, in SAGE, the source type in a cast is always the dynamic type. While this makes the cast semantics much simpler, parametricity in the presence of the dynamic type would not be straightforward [69].

Subsumption-free formulation Although subtyping plays a crucial role in manifest calculi, it also brings a metatheoretic issue, as described by Knowles and Flanagan [64] and Greenberg et al. [44]. The issue is that rules of the type system are not monotonic—in particular, the subtyping rule for refinement types refers to well typedness in a negative position for well formed closing substitutions—and so it is not clear that the type system is even well defined. Knowles and Flanagan [64] and Greenberg et al. [44] have avoided it by giving denotational semantics (namely, logical predicates) of types and changing the problematic subtyping rule so that it refers to the denotations instead of well typedness. One (philosophical) problem is that soundness of the type system with respect to the denotational semantics has to be shown *before* soundness with respect to the operational semantics. Another, perhaps more serious problem is that the denotational approach is expected to be hard to scale than standard syntactic methods (i.e., progress and preservation), when we consider other features such as polymorphism. We discuss it in more detail in Section 5.2.

$\mathrm{F_H}$ addresses this issue by dropping subsumption (and hence subtyping) from the type system. Since subtyping is removed, it is easy to see that the type system is well defined. However, removing subtyping raises the two issues for type soundness again and, additionally, another issue about how to deal with static verification, which is based on subtyping in the original hybrid checking framework.

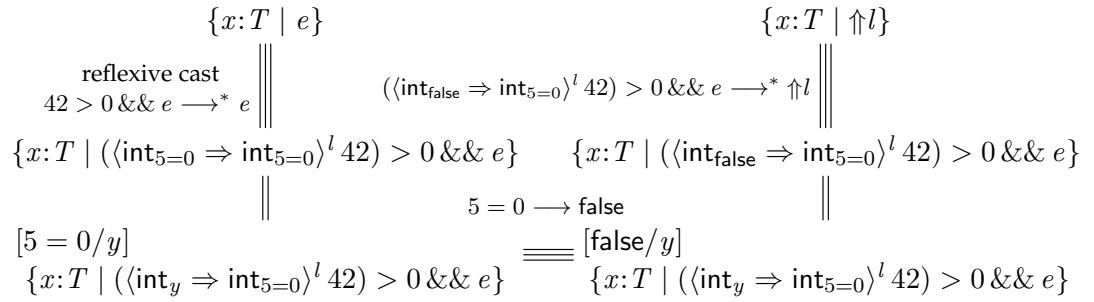

FIGURE 3.1: An inconsistent derivation of $\mathrm{F_H}$'s type conversion relation.

For the type soundness issues, Belo et al. introduce a special typing rule to give values any refinement they satisfy and a type conversion relation, which is based on (call-by-value) parallel reduction.[3] With the type conversion relation, $[e_1/x]T$ and $[e_2/x]T$ are convertible if $e_1 \longrightarrow e_2$ and a typing rule that allows terms to be retyped at convertible types is substituted for the subsumption rule. Using such a type system, they claim to have "proved" type soundness in an entirely syntactic manner—via progress and preservation—and also parametricity based on syntactic logical relations.

Although the resulting system can be formalized without resting on denotational semantics, the lack of subsumption means that all refinements in a well typed program will be checked at run time. As we have already mentioned in the beginning of this chapter, Belo et al. recover static verification by introducing subtyping *post facto* and examining sufficient conditions to eliminate casts.

3.1.3 Flaws in $\mathrm{F_H}$—and How We Solve Them

Unfortunately, as mentioned in the beginning of this chapter, a few properties required to show type soundness and parametricity turn out to be false. We will discuss the flawed properties with their counterexamples in detail in Section 3.5 but, in essence, the source of anomaly is that substitutions, which affect how casts behave, badly interact with the type conversion. As we discussed above, for preservation, two types $[e_1/x]T$ and $[e_2/x]T$ should be convertible if $e_1 \longrightarrow e_2$. Naively allowing this, however, will cause two refinement types $\{x{:}T \mid e_1\}$ and $\{x{:}T \mid e_2\}$ to be convertible (via $\{x{:}T \mid \Uparrow l\}$) for *any* Boolean terms e_1 and e_2. So, $\mathrm{F_H}$'s (static) contract system is inconsistent in the sense that a well typed cast-free term can be given any refinement.

Figure 3.1 shows such a derivation (here, given (closed) expression e, we write int_e for $\{z{:}\mathsf{int} \mid e\}$ and $\&\&$ stands for Boolean conjunction; the relation $\equiv$ denotes the type conversion relation). The crux of this example is that substitution of $5 = 0$ for y yields a reflexive cast, while that of false for y yields a failing cast. Actually, the two intermediate types are *ill-formed*, because 42 cannot be given type $\mathsf{int}_{5=0}$ or $\mathsf{int}_{\mathsf{false}}$—the source types of the casts. Nevertheless, we cannot exclude such nonsense terms and have to examine properties of a type conversion relation in the untyped setting *until we prove type soundness*.

$\mathrm{F}^{\sigma}_{\mathrm{H}}$ corrects this anomaly; in $\mathrm{F}^{\sigma}_{\mathrm{H}}$,

$$\{x{:}T \mid (\langle \mathsf{int}_{5=0} \Rightarrow \mathsf{int}_{5=0} \rangle^l 42) > 0 \,\&\&\, e\} \not\equiv \{x{:}T \mid (\langle \mathsf{int}_{\mathsf{false}} \Rightarrow \mathsf{int}_{5=0} \rangle^l 42) > 0 \,\&\&\, e\},$$

[3] Belo et al. [14] do not really show a formal definition of type conversion; it appears in Greenberg [43] and will be presented in Section 3.5.

avoiding $\{x{:}T \mid e\} \equiv \{x{:}T \mid \Uparrow l\}$, whereas

$$\begin{array}{c} [5=0/y] \\ \{x{:}T \mid (\langle \mathsf{int}_y \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\} \end{array} \equiv \begin{array}{c} [\mathsf{false}/y] \\ \{x{:}T \mid (\langle \mathsf{int}_y \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\} \end{array}$$

does hold. At first, these (in)equations seem contradictory because the first type $\{x{:}T \mid (\langle \mathsf{int}_{5=0} \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\}$ and the third $[5=0/y]\{x{:}T \mid (\langle \mathsf{int}_y \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\}$ are usually syntactically equal and so are the second and fourth. In fact, $\mathrm{F}^{\sigma}_{\mathrm{H}}$ distinguishes both pairs syntactically and obtains desirable type conversion, as illustrated below.

$$\begin{array}{ccc} \{x{:}T \mid e\} & & \{x{:}T \mid \Uparrow l\} \\ \Vert\!\Vert & & \Vert\!\Vert \\ \{x{:}T \mid (\langle \mathsf{int}_{5=0} \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\} & & \{x{:}T \mid (\langle \mathsf{int}_{\mathsf{false}} \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\} \\ \nparallel & & \nparallel \\ [5=0/y] & & [\mathsf{false}/y] \\ \{x{:}T \mid (\langle \mathsf{int}_y \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\} & \equiv & \{x{:}T \mid (\langle \mathsf{int}_y \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\} \end{array}$$

This is achieved by (1) changing the syntax and semantics of casts so that substitution does not affect how casts behave and (2) devising type conversion based on the notion we call "common subexpression reduction" (or CSR).

Delayed substitutions for casts To distinguish $[5=0/y]\langle \mathsf{int}_y \Rightarrow \mathsf{int}_{5=0}\rangle^l$ and $\langle \mathsf{int}_{5=0} \Rightarrow \mathsf{int}_{5=0}\rangle^l$, $\mathrm{F}^{\sigma}_{\mathrm{H}}$ uses *delayed substitutions* σ, which are also used to ensure that substitution does not interfere with how casts evaluate. First, cast expressions are augmented with delayed substitutions and take the form $\langle T_1 \Rightarrow T_2\rangle^l_{\sigma}$. (We often omit σ when it is empty.) Second, a substitution applied to casts is *not* forwarded to their target and source types immediately and instead stored as delayed substitutions—this is the reason why σ is called "delayed." For example, when term $5=0$ is substituted for y in $\langle \mathsf{int}_y \Rightarrow \mathsf{int}_{5=0}\rangle^l$, the result is $\langle \mathsf{int}_y \Rightarrow \mathsf{int}_{5=0}\rangle^l_{\sigma}$ where σ maps y to $5=0$. Delayed substitutions attached to casts are ignored when deciding what steps to take to check values. Thus, $\langle \mathsf{int}_y \Rightarrow \mathsf{int}_{5=0}\rangle^l_{\sigma}$ does not disappear, even when $[5=0/y]\mathsf{int}_y$ and $\mathsf{int}_{5=0}$ are syntactically equal; instead, a check to see if $5=0$ evaluates to true will run and the cast will raise blame eventually.

New type conversion, common subexpression reduction The motivation for type conversion was that we had to relate two types $[e_1/x]\,T$ and $[e_2/x]\,T$ if $e_1 \longrightarrow e_2$. Now that delayed substitutions make explicit what substitutions are applied, we can define type conversion so that it relates two types only if their differences are in substituted terms, not arbitrary subexpressions at the same position. Since the substituted terms are related by reduction, we call the new type conversion relation $\equiv$ *common subexpression reduction* (or CSR). Consequently, CSR $T_1 \equiv T_2$ is given as congruence closed under the following rule:

$$\frac{T_1 \equiv T_2 \qquad \forall i \in \{1,\ldots,n\}.e_i \longrightarrow^* e_i'}{\{x{:}T_1 \mid [e_1/x_1,\ldots,e_n/x_n]e\} \equiv \{x{:}T_2 \mid [e_1'/x_1,\ldots,e_n'/x_n]e\}}$$

Now the two types $\{x{:}T \mid (\langle \mathsf{int}_{5=0} \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\}$ and $\{x{:}T \mid (\langle \mathsf{int}_{\mathsf{false}} \Rightarrow \mathsf{int}_{5=0}\rangle^l\, 42) > 0 \,\&\&\, e\}$ are *not* convertible because it is not possible to "factor out" the difference of the two types in the form of substitution $[e/y]\,T$.

Terms, substitutions, and contexts

$$
\begin{array}{rcl}
\mathsf{Ty} \ni T & ::= & B \mid \alpha \mid x{:}T_1 \to T_2 \mid \forall\alpha.\, T \mid \{x{:}T \mid e\} \\
\sigma & \in & (\mathsf{TmVar} \xrightarrow{\text{fin}} \mathsf{Tm}) \times (\mathsf{TyVar} \xrightarrow{\text{fin}} \mathsf{Ty}) \\
\Gamma & ::= & \emptyset \mid \Gamma, x{:}T \mid \Gamma, \alpha
\end{array}
$$

Terms, values, results, and evaluation contexts

$$
\begin{array}{rcl}
\mathsf{Tm} \ni e & ::= & x \mid k \mid op\,(e_1, \ldots, e_n) \mid \lambda x{:}T.\ e \mid \Lambda\alpha.\ e \mid e_1\ e_2 \mid e\ T \mid \\
 & & \langle T_1 \Rightarrow T_2\rangle^l_\sigma \mid \Uparrow l \mid \langle\{x{:}T \mid e_1\}, e_2, v\rangle^l \\
v & ::= & k \mid \lambda x{:}T.\ e \mid \Lambda\alpha.\ e \mid \langle T_1 \Rightarrow T_2\rangle^l_\sigma \\
r & ::= & v \mid \Uparrow l \\
E & ::= & [\,]\ e_2 \mid v_1\ [\,] \mid [\,]\ T \mid \langle\{x{:}T \mid e\}, [\,], v\rangle^l \mid op(v_1, \ldots, v_{i-1}, [\,], e_{i+1}, \ldots, e_n)
\end{array}
$$

FIGURE 3.2: Syntax for $\mathrm{F}^\sigma_{\mathrm{H}}$

3.2 Defining $\mathrm{F}^\sigma_{\mathrm{H}}$

3.2.1 Syntax

The syntax of $\mathrm{F}^\sigma_{\mathrm{H}}$ is given in Figure 3.2. For unrefined types we have: base types B, which must include bool; type variables α; dependent function types $x{:}T_1 \to T_2$ where x is bound in T_2; and universal types $\forall\alpha.\, T$, where α is bound in T. Aside from dependency in function types, these are just the types of the standard polymorphic lambda calculus. For each B, we fix a set $\mathcal{K}_B$ of the constants in that type. We require the typing rules for constants and the typing and evaluation rules for operations to respect this set; we formally define requirements for constants and operations in Section 3.2.3. We also require that $\mathcal{K}_{\mathsf{bool}} = \{\mathsf{true}, \mathsf{false}\}$. We also have predicate contracts, or *refinement types*, written $\{x{:}T \mid e\}$. Conceptually, $\{x{:}T \mid e\}$ denotes values v of type T for which $[v/x]e$ reduces to true. As mentioned before, refinement types in $\mathrm{F}^\sigma_{\mathrm{H}}$ are more general than existing manifest calculi (except for SAGE [46]) in that *any* type (even a refinement type) can be refined, not just base types (as in [37, 44, 45, 64, 82]).

In the syntax of terms, the first line is standard for a call-by-value polymorphic language: variables, constants, several monomorphic first-order operations op (i.e., destructors of one or more base-type arguments), term and type abstractions, and term and type applications. Note that there is no value restriction on type abstractions—as in System F, we do not evaluate under type abstractions, so there is no issue with ordering of effects. The second line offers the standard constructs of a manifest contract calculus [37, 44, 64], with a few alterations, discussed below.

As we have already discussed in the last section, casts in $\mathrm{F}^\sigma_{\mathrm{H}}$ are of the form $\langle T_1 \Rightarrow T_2\rangle^l_\sigma$, where the delayed substitution σ is formally a pair of substitutions from term and type variables to terms and types, respectively. When a cast detects a problem, it raises blame, a label-indexed uncatchable exception written $\Uparrow l$. The label l allows us to trace blame back to a specific cast. (While labels here are drawn from an arbitrary set, in practice l will refer to a source-code location.) Finally, we use active checks $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^l$ to support a small-step semantics for checking casts into refinement types. In an active check, $\{x{:}T \mid e_1\}$ is the refinement being checked, e_2 is the current state of checking, and v is the value being checked. The type in the first position of an active check is not necessary for the operational semantics, but we keep it around as a technical aid to our syntactic proof of preservation. The value in the third position can be any value, not just a constant according to generalization of refinement types. If checking the refinement type succeeds, the check will return v; if checking fails, the

check will blame its label, raising $\Uparrow l$. Active checks and blame are not intended to occur in source programs—they are run-time devices. (In a real programming language based on this calculus, casts will probably not appear explicitly either, but will be inserted by an elaboration phase. The details of this process are beyond the present scope. Readers are referred to, e.g., Flanagan [37].)

The values in $\mathrm{F}^{\sigma}_{\mathrm{H}}$ are constants, term and type abstractions, and casts. We also define *results*, which are either values or blame. Type soundness, stated in Theorem 7, will show that evaluation produces a result, but not necessarily a value. We note that, as in Chapter 2, function cast applications $\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22} \rangle^{l}\ v$ are not seen as values, which simplifies our inversion lemmas, and instead casts between function types will η-expand and wrap with the casts on the domain and the codomain their argument. This makes the notion of "function proxy" explicit: the cast semantics adds many new closures.

To define semantics, we use evaluation contexts [33] (ranged over by E), a standard tool to introduce small-step operational semantics. The syntax of evaluation contexts shown in Figure 3.2 means that the semantics evaluates subterms from left to right in the call-by-value style.

As usual, we introduce some conventional notations. We write $\mathrm{FV}(e)$ (resp. $\mathrm{FV}(T)$) to denote free term variables in the term e (resp. the type T), which is defined as usual, except for casts:

$$\mathrm{FV}(\langle T_1 \Rightarrow T_2 \rangle^{l}_{\sigma}) = ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma)$$

where $\mathrm{dom}(\sigma)$ is the domain set of σ and $\mathrm{FV}(\sigma)$ is the set of free term variables in terms and types that appear in the range of σ. Similarly, we use $\mathrm{FTV}(e)$, $\mathrm{FTV}(T)$, and $\mathrm{FTV}(\sigma)$ for free type variables, and $\mathrm{AFV}(e)$, $\mathrm{AFV}(T)$, and $\mathrm{AFV}(\sigma)$ for all free variables, namely, both free term and type variables. We say that terms and types are closed when they have no free term and type variables.

We define application of substitutions, which is almost standard except the case for casts, below. To preserve standard properties of substitution, such as, "applying a substitution to a closed term yields the same term," we consider only terms without garbage bindings in delayed substitutions and assume that $\mathrm{dom}(\sigma) \subseteq \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)$ holds for every cast $\langle T_1 \Rightarrow T_2 \rangle^{l}_{\sigma}$. Before defining application of substitution, we introduce a few auxiliary notations. For a set S of variables, $\sigma|_S$ denotes the restriction of σ to S. Formally,

$$\sigma|_S = (\{x \mapsto \sigma(x) \mid x \in \mathrm{dom}(\sigma) \cap S\}, \{\alpha \mapsto \sigma(\alpha) \mid \alpha \in \mathrm{dom}(\sigma) \cap S\}).$$

We denote by $\sigma_1 \uplus \sigma_2$ a delayed substitution obtained by concatenating substitutions with disjoint domains elementwise.

Definition 2 (Substitution)**.** *Substitution in F^{σ}_{H} is the standard capture-avoiding substitution function with a single change, in the cast case:*

$$\sigma(\langle T_1 \Rightarrow T_2 \rangle^{l}_{\sigma_1}) = \langle T_1 \Rightarrow T_2 \rangle^{l}_{\sigma_2}$$

where $\sigma_2 = \sigma(\sigma_1) \uplus (\sigma|_{(\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1)})$. Here, $\sigma(\sigma_1)$ denotes the (pairwise) composition of σ and σ_1; formally,

$$\sigma(\sigma_1) = (\{x \mapsto \sigma(\sigma_1(x)) \mid x \in \mathrm{dom}(\sigma_1)\}, \{\alpha \mapsto \sigma(\sigma_1(\alpha)) \mid \alpha \in \mathrm{dom}(\sigma_1)\}).$$

Reduction rules $\boxed{e_1 \rightsquigarrow e_2}$

$$
\begin{array}{lcll}
op\,(v_1, \ldots, v_n) & \rightsquigarrow & [\![\mathrm{op}]\!]\,(v_1, \ldots, v_n) & \text{E\_OP} \\
(\lambda x{:}T_1.\ e_{12})\ v_2 & \rightsquigarrow & [v_2/x]e_{12} & \text{E\_BETA} \\
(\Lambda\alpha.\ e)\ T & \rightsquigarrow & [T/\alpha]e & \text{E\_TBETA} \\
\\
\langle T \Rightarrow T\rangle^l_\sigma\ v & \rightsquigarrow & v & \text{E\_REFL} \\
\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l_\sigma\ v & \rightsquigarrow & & \text{E\_FUN} \\
\multicolumn{4}{r}{\lambda x{:}\sigma(T_{21}).\ \mathsf{let}\ y : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\ x\ \mathsf{in}\ \langle [y/x]T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2}\ (v\ y)} \\
\multicolumn{4}{r}{\text{when } x{:}T_{11} \to T_{12} \neq x{:}T_{21} \to T_{22} \text{ and } x \notin \mathrm{dom}(\sigma) \text{ and}} \\
\multicolumn{4}{r}{y \text{ is fresh and, for } i \in \{1,2\},\ \sigma_i = \sigma|_{\mathrm{AFV}(T_1\,i) \cup \mathrm{AFV}(T_2\,i)}} \\
\langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l_\sigma\ v & \rightsquigarrow & \Lambda\alpha.\ (\langle [\alpha/\alpha]T_1 \Rightarrow T_2\rangle^l_\sigma\ (v\,\alpha)) & \text{E\_FORALL} \\
\multicolumn{4}{r}{\text{when } \forall\alpha.T_1 \neq \forall\alpha.T_2 \text{ and } \alpha \notin \mathrm{dom}(\sigma)} \\
\\
\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l_\sigma\ v & \rightsquigarrow & \langle T_1 \Rightarrow T_2\rangle^l_{\sigma'}\ v & \text{E\_FORGET} \\
\multicolumn{4}{r}{\text{when } T_2 \neq \{x{:}T_1 \mid e\} \text{ and } T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e_2\}} \\
\multicolumn{4}{r}{(\sigma' = \sigma|_{\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)})} \\
\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_\sigma\ v & \rightsquigarrow & & \text{E\_PRECHECK} \\
\multicolumn{4}{r}{\langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1}\ (\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\ v)} \\
\multicolumn{4}{r}{\text{when } T_1 \neq T_2 \text{ and } T_1 \neq \{x{:}T' \mid e'\}} \\
\multicolumn{4}{r}{(\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2|e_2\})} \text{ and } \sigma_2 = \sigma|_{\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)})} \\
\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l_\sigma\ v & \rightsquigarrow & \langle \sigma(\{x{:}T \mid e\}), \sigma([v/x]e), v\rangle^l & \text{E\_CHECK} \\
\\
\langle \{x{:}T \mid e\}, \mathsf{true}, v\rangle^l & \rightsquigarrow & v & \text{E\_OK} \\
\langle \{x{:}T \mid e\}, \mathsf{false}, v\rangle^l & \rightsquigarrow & \Uparrow l & \text{E\_FAIL}
\end{array}
$$

Evaluation rules $\boxed{e_1 \longrightarrow e_2}$

$$\frac{e_1 \rightsquigarrow e_2}{e_1 \longrightarrow e_2}\ \text{E\_REDUCE} \qquad \frac{e_1 \longrightarrow e_2}{E\,[e_1] \longrightarrow E\,[e_2]}\ \text{E\_COMPAT} \qquad \frac{}{E\,[\Uparrow l] \longrightarrow \Uparrow l}\ \text{E\_BLAME}$$

FIGURE 3.3: Operational semantics for $\mathrm{F}^{\sigma}_{\mathrm{H}}$

Notice that, in the definition of σ_2, the restriction on σ is required to remove garbage bindings. We show that many properties of substitution in lambda calculi hold for our substitution in Appendix.

Finally, we introduce several syntactic shorthands. We write $T_1 \to T_2$ for $x{:}T_1 \to T_2$ when x does not appear free in T_2 and $\langle T_1 \Rightarrow T_2\rangle^l$ for $\langle T_1 \Rightarrow T_2\rangle^l_\sigma$ if the domain of σ is empty. A let expression $\mathsf{let}\ x : T = e_1\ \mathsf{in}\ e_2$ denotes an application term of the form $(\lambda x{:}T.\ e_2)\,e_1$. We may omit the type if it is clear from the context. If $\sigma = (\{x \mapsto e\}, \emptyset)$, then we write $[e/x]e'$, $[e/x]T'$, and $[e/x]\sigma'$ for $\sigma(e')$, $\sigma(T')$, and $\sigma(\sigma')$, respectively. Similarly, we write $[T/\alpha]e'$, $[T/\alpha]T'$, and $[T/\alpha]\sigma'$ for $\sigma(e')$, $\sigma(T')$, and $\sigma(\sigma')$, respectively, if $\sigma = (\emptyset, \{\alpha \mapsto T\})$.

3.2.2 Operational Semantics

The call-by-value operational semantics in Figure 3.3 is given as a small-step relation, split into two sub-relations: one for reductions ($\rightsquigarrow$) and one for subterm reductions and blame lifting ($\longrightarrow$). We define these relations as over *closed* terms.

The latter relation is standard. The (E_REDUCE) rule lifts $\rightsquigarrow$ reductions into $\longrightarrow$; the (E_COMPAT) rule reduces subterms put in evaluation contexts; and the (E_BLAME) rule lifts blame, treating it as an uncatchable exception. The reduction relation $\rightsquigarrow$ is more interesting. There are four different kinds of reductions: the standard lambda calculus reductions, structural cast reductions, cast staging reductions, and checking reductions.

The (E_BETA) and (E_TBETA) rules should need no explanation—these are the standard call-by-value polymorphic lambda calculus reductions. The (E_OP) rule uses a denotation function $[\![-]\!]$ to give meaning to the first-order operations. In Section 3.2.3, we describe a property of $[\![-]\!]$ to be required for showing type soundness.

The (E_REFL), (E_FUN), and (E_FORALL) rules reduce casts structurally. (E_REFL) eliminates a cast from a type to itself; intuitively, such a cast should always succeed anyway. (We discuss this rule more in Section 3.4.) When a cast between function types is applied to a value v, the (E_FUN) rule produces a new lambda, wrapping v with a contravariant cast on the domain and a covariant cast on the codomain. The extra substitution in the left-hand side of the codomain cast may seem suspicious, but in fact the rule must be this way for type preservation to hold (see Greenberg et al. [44] for an explanation). Just like substitution (Definition 2), (E_FUN) and other cast rules restrict the domain of each delayed substitution in the right-hand side of reduction to free variables in the source and the target types of the corresponding cast. Note that (E_FUN) uses a let expression—syntactic sugar for immediate application of a lambda—for the domain check. This is a nicer evaluation semantics than one in the previous calculi where the domain check can be duplicated by substitution. Avoiding this duplication is more efficient and simplifies some of our proofs of parametricity—in particular, we not need to show that our logical relation is closed under *term* substitution, i.e., two open, logically related terms are related after replacing variables in them with logically related terms. The (E_FORALL) rule is similar to (E_FUN), generating a type abstraction with the necessary covariant cast. A seemingly trivial substitution $[\alpha/\alpha]$ is necessary for showing preservation: the value v in this rule is expected to have $\forall\alpha.\,T_1$ and then $v\,\alpha$ is given type $[\alpha/\alpha]\,T_1$, which is not the same as T_1 in general, even though T_1 and $[\alpha/\alpha]\,T_1$ are semantically equivalent, since substitution is delayed at casts! So, after the reduction, the source type of the cast has to be $[\alpha/\alpha]\,T_1$. Side conditions on (E_FORALL) and (E_FUN) ensure that these rules apply only when (E_REFL) does not.

The (E_FORGET), (E_PRECHECK), and (E_CHECK) rules are cast-staging reductions, breaking a complex cast down to a series of simpler casts and checks. All of these rules require that the left- and right-hand sides of the cast be different—if they are the same, then (E_REFL) applies. The (E_FORGET) rule strips a layer of refinement off the left-hand side; in addition to requiring that the left- and right-hand sides are different, the preconditions require that the right-hand side is not a refinement of the left-hand side. The (E_PRECHECK) rule breaks a cast into two parts: one that checks exactly one level of refinement and another that checks the remaining parts. We only apply this rule when the two sides of the cast are different and when the left-hand side is not a refinement. The (E_CHECK) rule applies when the right-hand side refines the left-hand side; it takes the cast value and checks that it satisfies the right-hand side. (We do not have to check the left-hand side, since that is the type we are casting *from*.) If the check

succeeds, then the active check evaluates to the checked value (by (E_OK)); otherwise, it is blamed with l (by (E_FAIL)).

We offer a few reduction examples, which are also put in Greenberg [43]. First, here is a reduction using (E_CHECK), (E_COMPAT), (E_OP), and (E_OK):

$$\begin{array}{rcl} \langle \mathsf{int} \Rightarrow \{x{:}\mathsf{int} \mid x \geq 0\} \rangle^l \, 5 & \longrightarrow & \langle \{x{:}\mathsf{int} \mid x \geq 0\}, 5 \geq 0, 5 \rangle^l \\ & \longrightarrow & \langle \{x{:}\mathsf{int} \mid x \geq 0\}, \mathsf{true}, 5 \rangle^l \\ & \longrightarrow & 5 \end{array}$$

A failed check will work in the same way until the last reduction, which will use (E_FAIL) rather than (E_OK):

$$\begin{array}{rcl} \langle \mathsf{int} \Rightarrow \{x{:}\mathsf{int} \mid x \geq 0\} \rangle^l \, (-1) & \longrightarrow & \langle \{x{:}\mathsf{int} \mid x \geq 0\}, -1 \geq 0, -1 \rangle^l \\ & \longrightarrow & \langle \{x{:}\mathsf{int} \mid x \geq 0\}, \mathsf{false}, -1 \rangle^l \\ & \longrightarrow & \Uparrow l \end{array}$$

Notice that the blame label comes from the cast that failed. Here is a similar reduction that needs some staging, using (E_FORGET) followed by the first reduction we gave:

$$\begin{array}{rcl} \langle \{x{:}\mathsf{int} \mid x = 5\} \Rightarrow \{x{:}\mathsf{int} \mid x \geq 0\} \rangle^l \, 5 & \longrightarrow & \langle \mathsf{int} \Rightarrow \{x{:}\mathsf{int} \mid x \geq 0\} \rangle^l \, 5 \\ & \longrightarrow & \langle \{x{:}\mathsf{int} \mid x \geq 0\}, 5 \geq 0, 5 \rangle^l \\ & \longrightarrow^* & 5 \end{array}$$

There are two cases where we need to use (E_PRECHECK). First, when nested refinements are involved:

$$\begin{array}{rl} & \langle \mathsf{int} \Rightarrow \{x{:}\{y{:}\mathsf{int} \mid y \geq 0\} \mid x = 5\} \rangle^l \, 5 \\ \longrightarrow & \langle \{y{:}\mathsf{int} \mid y \geq 0\} \Rightarrow \{x{:}\{y{:}\mathsf{int} \mid y \geq 0\} \mid x = 5\} \rangle^l \, (\langle \mathsf{int} \Rightarrow \{y{:}\mathsf{int} \mid y \geq 0\} \rangle^l \, 5) \\ \longrightarrow^* & \langle \{y{:}\mathsf{int} \mid y \geq 0\} \Rightarrow \{x{:}\{y{:}\mathsf{int} \mid y \geq 0\} \mid x = 5\} \rangle^l \, 5 \\ \longrightarrow & \langle \{x{:}\{y{:}\mathsf{int} \mid y \geq 0\} \mid x = 5\}, 5 = 5, 5 \rangle^l \\ \longrightarrow^* & 5 \end{array}$$

Second, when a function or universal type is cast into a refinement of a *different* function or universal type:

$$\begin{array}{rl} & \langle \mathsf{bool} \to \{x{:}\mathsf{bool} \mid x\} \Rightarrow \{f{:}\mathsf{bool} \to \mathsf{bool} \mid f\ \mathsf{true} = f\ \mathsf{false}\} \rangle^l \, v \\ \longrightarrow & \langle \mathsf{bool} \to \mathsf{bool} \Rightarrow \{f{:}\mathsf{bool} \to \mathsf{bool} \mid f\ \mathsf{true} = f\ \mathsf{false}\} \rangle^l \\ & \qquad (\langle \mathsf{bool} \to \{x{:}\mathsf{bool} \mid x\} \Rightarrow \mathsf{bool} \to \mathsf{bool} \rangle^l \, v) \end{array}$$

(E_REFL) is necessary for simple cases, like $\langle \mathsf{int} \Rightarrow \mathsf{int} \rangle^l \, 5 \longrightarrow 5$. Hopefully, such a useless cast would never be written, but it could arise as a result of (E_FUN) or (E_FORALL). (We also need (E_REFL) in our proof of parametricity; see Section 3.4.)

The two high-level ways given by Greenberg [43] would be useful to understand the cast semantics in $\mathrm{F}^{\sigma}_{\mathrm{H}}$: one is a recursive function to unfold cast forms; the other is a regular schema to indicate the order in which cast reduction rules are applied. Interested readers can refer to his dissertation [43].

3.2.3 Static Typing

The type system comprises three mutually recursive judgments: context well formedness ($\vdash \Gamma$), type well formedness ($\Gamma \vdash T$), and term typing ($\Gamma \vdash e : T$). The rules for contexts and types are unsurprising. The rules for terms are mostly standard. First, the

Context well formedness $\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset}\ \text{WF\_EMPTY} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash T}{\vdash \Gamma, x{:}T}\ \text{WF\_EXTENDVAR} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, \alpha}\ \text{WF\_EXTENDTVAR}$$

Type well formedness $\boxed{\Gamma \vdash T}$

$$\frac{\vdash \Gamma}{\Gamma \vdash B}\ \text{WF\_BASE} \qquad \frac{\vdash \Gamma \quad \alpha \in \Gamma}{\Gamma \vdash \alpha}\ \text{WF\_TVAR} \qquad \frac{\Gamma, \alpha \vdash T}{\Gamma \vdash \forall \alpha. T}\ \text{WF\_FORALL}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash x{:}T_1 \to T_2}\ \text{WF\_FUN} \qquad \frac{\Gamma \vdash T \quad \Gamma, x{:}T \vdash e : \mathsf{bool}}{\Gamma \vdash \{x{:}T \mid e\}}\ \text{WF\_REFINE}$$

Term typing $\boxed{\Gamma \vdash e : T}$

$$\frac{\vdash \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_VAR} \qquad \frac{\vdash \Gamma}{\Gamma \vdash k : \mathsf{ty}(k)}\ \text{T\_CONST} \qquad \frac{\emptyset \vdash T \quad \vdash \Gamma}{\Gamma \vdash \Uparrow l : T}\ \text{T\_BLAME}^*$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma, x{:}T_1 \vdash e_{12} : T_2}{\Gamma \vdash \lambda x{:}T_1.\ e_{12} : x{:}T_1 \to T_2}\ \text{T\_ABS} \qquad \frac{\Gamma \vdash e_1 : (x{:}T_1 \to T_2) \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : [e_2/x]\,T_2}\ \text{T\_APP}$$

$$\frac{\begin{array}{c}\vdash \Gamma \qquad \mathsf{ty}(op) = x_1 : T_1 \to \ \ldots\ \to x_n : T_n \to T \\ \forall i \in \{1, \ldots, n\}, \Gamma \vdash e_i : [e_1/x_1, \ldots, e_{i-1}/x_{i-1}]\,T_i\end{array}}{\Gamma \vdash op\,(e_1, \ \ldots, e_n) : [e_1/x_1, \ldots, e_n/x_n]\,T}\ \text{T\_OP}$$

$$\frac{\Gamma, \alpha \vdash e : T}{\Gamma \vdash \Lambda \alpha.\ e : \forall \alpha. T}\ \text{T\_TABS} \qquad \frac{\Gamma \vdash e_1 : \forall \alpha. T \quad \Gamma \vdash T_2}{\Gamma \vdash e_1\ T_2 : [T_2/\alpha]\,T}\ \text{T\_TAPP}$$

$$\frac{\Gamma \vdash \sigma(T_1) \quad \Gamma \vdash \sigma(T_2) \quad T_1 \parallel T_2 \quad \mathrm{AFV}(\sigma) \subseteq \mathrm{dom}(\Gamma)}{\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : \sigma(T_1) \to \sigma(T_2)}\ \text{T\_CAST}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{bool} \quad [v/x]e_1 \longrightarrow^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle^l : \{x{:}T \mid e_1\}}\ \text{T\_CHECK}^*$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash e : T \quad \emptyset \vdash T' \quad T \equiv T'}{\Gamma \vdash e : T'}\ \text{T\_CONV}^* \qquad \frac{\vdash \Gamma \quad \emptyset \vdash v : \{x{:}T \mid e\}}{\Gamma \vdash v : T}\ \text{T\_FORGET}^*$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash v : T \quad \emptyset \vdash \{x{:}T \mid e\} \quad [v/x]e \longrightarrow^* \mathsf{true}}{\Gamma \vdash v : \{x{:}T \mid e\}}\ \text{T\_EXACT}^*$$

FIGURE 3.4: Typing rules for $\mathrm{F}^{\sigma}_{\mathrm{H}}$. The rules marked $*$ are for "run-time" terms.

(T_CONST) and (T_OP) rules use the ty function to assign well-formed, closed (possibly dependent) monomorphic first-order types to constants and operations, respectively. To formalize the demand to constants, we define $\mathrm{unref}(T)$ as T without any outer refinements (though refinements on, e.g., the domain of a function would be unaffected):

$$\mathrm{unref}(T) = \begin{cases} \mathrm{unref}(T') & \text{if } T = \{x{:}T' \mid e\} \\ T & \text{otherwise} \end{cases}$$

We require constants to belong to $\mathcal{K}_{\mathrm{unref}(\mathsf{ty}(k))}$ and satisfy the predicate (if any) of $\mathsf{ty}(k)$ and $[\![op]\!]$ to be a function that returns a value satisfying the predicate of the codomain type of $\mathsf{ty}(op)$ when each argument value satisfies the predicate of the corresponding domain type of $\mathsf{ty}(op)$. The (T_APP) rule is dependent, to account for dependent function types. The (T_CAST) rule allows casts between compatibly structured well formed types, demanding that both source and target types after applying delayed substitution be well-formed. Compatibility of type structures is defined in Figure 3.5; intuitively, compatible types are identical when predicates in them are ignored. In particular, it is critical that type variables are compatible with only (refinements of) themselves because we have no idea what type will be substituted for α. If we allow type variable α to be compatible with another type, say, B, then the check with the cast from α to B would not work when α is replaced with a function type or a quantified type. Moreover, this definition helps us avoid nontermination due to non-parametric operations (e.g., Girard's J operator); it is imperative that a term like

$$\mathsf{let}\ \delta = \Lambda\alpha.\ \lambda x{:}\alpha.\ \langle \alpha \Rightarrow \forall\beta.\beta\to\beta\rangle^{l}\ x\ \alpha\ x\ \mathsf{in}\ \delta\ (\forall\beta.\beta\to\beta)\ \delta$$

is not well typed. Note that, in (T_CAST), we assign casts a non-dependent function type and that we do not require well typedness/formedness of terms/types that appear in the range of a delayed substitution in a direct way—though well typed programs will start with and preserve well typed substitutions. Finally, it is critical that compatibility is substitutive, i.e., that if $T_1 \parallel T_2$, then $([e/x]\,T_1) \parallel T_2$ (Lemma B.3.8).

Some of the typing rules—(T_CHECK), (T_BLAME), (T_EXACT), (T_FORGET), and (T_CONV)—are "run-time only." These rules are not needed to typecheck source programs, but we need them to guarantee preservation. (T_CHECK), (T_EXACT), and (T_CONV) are excluded from source programs because we do not want the typing of source programs to rely on the evaluation relation; such an interaction is acceptable in this setting, but disrupts the phase distinction and is ultimately incompatible with nontermination and effects. We exclude (T_BLAME) because programs should not *start* with failures. Finally, we exclude (T_FORGET) because we imagine that source programs have all type changes explicitly managed by casts. The conclusions of these rules use a context Γ, but all terms and types in premises have to be well typed and well formed under the empty context. Even though run-time terms and their typing rules should only ever occur in the empty context, the (T_APP) rule substitutes terms into types—so a run-time term could end up under a binder. We therefore allow the run-time typing rules to apply in any well formed context, so long as the terms they typecheck are closed. The (T_BLAME) rule allows us to give any type to blame—this is necessary for preservation. The (T_CHECK) rule types an active check, $\langle\{x{:}T \mid e_1\}, e_2, v\rangle^{l}$. Such a term arises when a term like $\langle T \Rightarrow \{x{:}T \mid e_1\}\rangle^{l}\ v$ reduces by (E_CHECK). The premises of the rule are all intuitive except for $[v/x]e_1 \longrightarrow^{*} e_2$, which ensures that e_2 is an intermediate state during checking $[v/x]e_1$. The (T_EXACT) rule allows us to retype a closed value of type T at $\{x{:}T \mid e\}$ if $[v/x]e \longrightarrow^{*} \mathsf{true}$. This typing rule guarantees type

Type compatibility $\boxed{T_1 \parallel T_2}$

$$\frac{}{\alpha \parallel \alpha}\ \textsc{Sim\_Var} \qquad \frac{}{B \parallel B}\ \textsc{Sim\_Base}$$

$$\frac{T_1 \parallel T_2}{\{x{:}T_1 \mid e\} \parallel T_2}\ \textsc{Sim\_RefineL} \qquad \frac{T_1 \parallel T_2}{T_1 \parallel \{x{:}T_2 \mid e\}}\ \textsc{Sim\_RefineR}$$

$$\frac{T_{11} \parallel T_{21} \quad T_{12} \parallel T_{22}}{x{:}T_{11} \to T_{12} \parallel x{:}T_{21} \to T_{22}}\ \textsc{Sim\_Fun} \qquad \frac{T_1 \parallel T_2}{\forall\alpha.T_1 \parallel \forall\alpha.T_2}\ \textsc{Sim\_Forall}$$

Conversion $\boxed{\sigma_1 \longrightarrow^* \sigma_2}$ $\boxed{T_1 \equiv T_2}$

$$\sigma_1 \longrightarrow^* \sigma_2 \iff \begin{array}{l}\mathsf{dom}(\sigma_1) = \mathsf{dom}(\sigma_2) \subset \mathsf{TmVar} \wedge \\ \forall x \in \mathsf{dom}(\sigma_1).\ \sigma_1(x) \longrightarrow^* \sigma_2(x)\end{array}$$

$$\frac{}{\alpha \equiv \alpha}\ \textsc{C\_Var} \qquad \frac{}{B \equiv B}\ \textsc{C\_Base} \qquad \frac{\sigma_1 \longrightarrow^* \sigma_2 \quad T_1 \equiv T_2}{\{x{:}T_1 \mid \sigma_1(e)\} \equiv \{x{:}T_2 \mid \sigma_2(e)\}}\ \textsc{C\_Refine}$$

$$\frac{T_1 \equiv T_1' \quad T_2 \equiv T_2'}{x{:}T_1 \to T_2 \equiv x{:}T_1' \to T_2'}\ \textsc{C\_Fun} \qquad \frac{T \equiv T'}{\forall\alpha.T \equiv \forall\alpha.T'}\ \textsc{C\_Forall}$$

$$\frac{T_2 \equiv T_1}{T_1 \equiv T_2}\ \textsc{C\_Sym} \qquad \frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}\ \textsc{C\_Trans}$$

FIGURE 3.5: Type compatibility and conversion for $\mathrm{F}^{\sigma}_{\mathrm{H}}$

preservation for (E_OK): $\langle\{x{:}T \mid e_1\}, \mathsf{true}, v\rangle^l \longrightarrow v$. If the active check was well typed, then we know that $[v/x]e_1 \longrightarrow^* \mathsf{true}$, so (T_EXACT) applies. (T_EXACT) is a suitably extensional, syntactic, and subtyping-free replacement for the technique using selfified types and subtyping [82].

Finally, the (T_CONV) rule is motivated by the requirement that terms of $[e_1/x]T$ and $[e_2/x]T$ should be able to be typed at both types if $e_1 \longrightarrow e_2$—it is necessary to prove preservation; see also the discussion in Section 3.1.2. These types are convertible in $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and (T_CONV) allows terms to be retyped at convertible types. We define a conversion relation $\equiv$, which we also call *common-subexpression reduction*, or CSR, using rules in Figure 3.5. Roughly speaking, T_1 and T_2 are convertible when there is a common type T and subexpressions e_1 and e_2 of T_1 and T_2 such that $T_1 = [e_1/x]T$ and $T_2 = [e_2/x]T$ and $e_1 \longrightarrow^* e_2$. The only interesting rule is (C_REFINE), which says that refinement types $\{x{:}T_1 \mid e_1\}$ and $\{x{:}T_2 \mid e_2\}$ are convertible when T_1 and T_2 are convertible and there are some substitutions σ_1, σ_2 and a common subexpression e such that $e_1 = \sigma_1(e)$ and $e_2 = \sigma_2(e)$ and each term which appears in the range of σ_1 reduces to one of σ_2. We remark that this conversion relation is different from that given in the prior ESOP 2011 work [14][4], where their conversion relation is defined in terms of parallel reduction. As discussed in Section 3.1.3, however, it turns out that their conversion relation is flawed. Another remark is that Belo et al. [14] also (falsely) claimed that symmetry of convertible relation was not necessary for type soundness

[4] Actually, the paper omits a formal definition, which appears in Greenberg [43].

or parametricity, but symmetry is in fact used in the proof of preservation (Theorem 6, when a term typed by (T_APP) steps by (E_REDUCE)/(E_REFL)).

3.3 Properties of F^{σ}_{H}

We show that well-typed programs do not get stuck—a well typed term evaluates to a result, i.e., a value or a blame (if evaluation terminates at all[5])—via progress and preservation [121].

As Greenberg [43] has pointed out, the "value inversion" lemma (Lemma 13), which says values typed at refinement types must satisfy their refinements, is a critical component of any sound manifest contract system, especially for proving progress. The type soundness proof in Belo et al. [14] is missing this lemma—and can never have it, due to the flawed conversion relation. Greenberg [43] leaves a property which the value inversion depends on as a conjecture—which turns out to be false. This value inversion lemma is not merely a technical device to prove progress. Together with progress and preservation, it means that if a term typed at a refinement type evaluates to a value, then it satisfies the predicate of the type, giving a slightly stronger guarantee about well typed programs.

Perhaps surprisingly, the value inversion lemma is not trivial due to (T_CONV): we must show that predicates of convertible refinement types are semantically equivalent. The proof of this property rests on cotermination (Lemma 11), which says that common-subexpression reduction does not change the behavior of terms. Finally, using these properties, we show progress (Theorem 5) and preservation (Theorem 6), which imply type soundness (Theorem 7). In this section, we only give statements of main lemmas and theorems; proofs are in Appendix.

3.3.1 Cotermination

First, we show cotermination, which both type soundness and parametricity rest on. We start with cotermination in the most simple situation, namely, where substitutions map only one term variable, and then show general cases. The key observation in proving cotermination is that the relation $\{([e_1/x]e, [e_2/x]e) \mid e_1 \longrightarrow e_2\}$ is weak bisimulation (Lemmas 8 and 9).

Lemma 8 (Weak bisimulation, left side)**.** *Suppose that* $e_1 \longrightarrow e_2$*. If* $[e_1/x]e \longrightarrow e'$*, then* $[e_2/x]e \longrightarrow^* [e_2/x]e''$ *for some* e'' *such that* $e' = [e_1/x]e''$*.*

Lemma 9 (Weak bisimulation, right side)**.** *Suppose that* $e_1 \longrightarrow e_2$*. If* $[e_2/x]e \longrightarrow e'$*, then* $[e_1/x]e \longrightarrow^* [e_1/x]e''$ *for some* e'' *such that* $e' = [e_2/x]e''$*.*

Lemma 10 (Cotermination, one variable)**.** *Suppose that* $e_1 \longrightarrow^* e_2$*.*

1. *If* $[e_1/x]e \longrightarrow^* \mathsf{true}$*, then* $[e_2/x]e \longrightarrow^* \mathsf{true}$*.*
2. *If* $[e_2/x]e \longrightarrow^* \mathsf{true}$*, then* $[e_1/x]e \longrightarrow^* \mathsf{true}$*.*

Lemma 11 (Cotermination)**.** *Suppose that* $\sigma_1 \longrightarrow^* \sigma_2$*.*

1. *If* $\sigma_1(e) \longrightarrow^* \mathsf{true}$*, then* $\sigma_2(e) \longrightarrow^* \mathsf{true}$*.*
2. *If* $\sigma_2(e) \longrightarrow^* \mathsf{true}$*, then* $\sigma_1(e) \longrightarrow^* \mathsf{true}$*.*

Proof. By induction on the size of $\mathsf{dom}(\sigma_1)$ with Lemma 10. □

[5] In fact, F^{σ}_{H} is terminating, as we will discover in Section 3.4.

3.3.2 Type Soundness

Using cotermination, we show value inversion and then type soundness in a standard syntactic way, starting with various substitution lemmas.

Lemma 12 (Cotermination of refinement types). *If* $\{x{:}T_1 \mid e_1\} \equiv \{x{:}T_2 \mid e_2\}$ *then* $T_1 \equiv T_2$ *and* $[v/x]e_1 \longrightarrow^*$ true *iff* $[v/x]e_2 \longrightarrow^*$ true, *for any closed value* v.

Value inversion (Lemma 13) uses unref_n, which is a function to remove only the n outermost refinements, to ensure that the value satisfies *all* of the predicates in its (possibly nested) refinement type. The function unref_n is defined as follows:

$$\mathrm{unref}_n(T) = \begin{cases} \mathrm{unref}_{n-1}(T') & \text{if } T = \{x{:}T' \mid e\} \text{ and } n > 0 \\ T & \text{otherwise} \end{cases}$$

Lemma 13 (Value inversion). *If* $\emptyset \vdash v : T$ *and* $\mathrm{unref}_n(T) = \{x{:}T_n \mid e_n\}$ *then* $[v/x]e_n \longrightarrow^*$ true.

Lemma 14 (Term substitutivity of conversion).
If $T_1 \equiv T_2$ *and* $e_1 \longrightarrow^* e_2$ *then* $[e_1/x]\,T_1 \equiv [e_2/x]\,T_2$.

Lemma 15 (Type substitutivity of conversion).
If $T_1 \equiv T_2$ *then* $[T/\alpha]\,T_1 \equiv [T/\alpha]\,T_2$.

Lemma 16 (Term weakening). *If* x *is fresh and* $\Gamma \vdash T'$ *then*

1. $\Gamma, \Gamma' \vdash e : T$ *implies* $\Gamma, x{:}T', \Gamma \vdash e : T$,
2. $\Gamma, \Gamma' \vdash T$ *implies* $\Gamma, x{:}T', \Gamma' \vdash T$, *and*
3. $\vdash \Gamma, \Gamma'$ *implies* $\vdash \Gamma, x{:}T', \Gamma'$.

Lemma 17 (Type weakening). *If* α *is fresh then*

1. $\Gamma, \Gamma' \vdash e : T$ *implies* $\Gamma, \alpha, \Gamma \vdash e : T$,
2. $\Gamma, \Gamma' \vdash T$ *implies* $\Gamma, \alpha, \Gamma' \vdash T$, *and*
3. $\vdash \Gamma, \Gamma'$ *implies* $\vdash \Gamma, \alpha, \Gamma'$.

Lemma 18 (Term substitution). *If* $\Gamma \vdash e' : T'$, *then*

1. *if* $\Gamma, x{:}T', \Gamma' \vdash e : T$ *then* $\Gamma, [e'/x]\Gamma' \vdash [e'/x]e : [e'/x]\,T$,
2. *if* $\Gamma, x{:}T', \Gamma' \vdash T$ *then* $\Gamma, [e'/x]\Gamma' \vdash [e'/x]\,T$, *and*
3. *if* $\vdash \Gamma, x{:}T', \Gamma'$ *then* $\vdash \Gamma, [e'/x]\Gamma'$.

Lemma 19 (Type substitution). *If* $\Gamma \vdash T'$ *then*

1. *if* $\Gamma, \alpha, \Gamma' \vdash e : T$, *then* $\Gamma, [T'/\alpha]\Gamma' \vdash [T'/\alpha]e : [T'/\alpha]\,T$,
2. *if* $\Gamma, \alpha, \Gamma' \vdash T$, *then* $\Gamma, [T'/\alpha]\Gamma' \vdash [T'/\alpha]\,T$, *and*
3. *if* $\vdash \Gamma, \alpha, \Gamma'$, *then* $\vdash \Gamma, [T'/\alpha]\Gamma'$.

As is standard for type systems with conversion rules, we must prove inversion lemmas to reason about typing derivations in a syntax-directed way. We offer the statement of inversion for functions here; the rest are in Section B.3.

Lemma 20 (Lambda inversion). *If* $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$, *then there exists some* T_2 *such that*

1. $\Gamma \vdash T_1$,
2. $\Gamma, x{:}T_1 \vdash e_{12} : T_2$, *and*
3. $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T)$.

Inversion lemmas in hand, we prove a canonical forms lemma to support a proof of progress. The canonical forms proof is "modulo" the unref function: the shape of the values of type $\{x{:}T \mid e\}$ are determined by the inner type T.

Lemma 21 (Canonical forms). *If* $\emptyset \vdash v : T$, *then:*

1. *If* $\mathrm{unref}(T) = B$ *then* v *is* $k \in \mathcal{K}_B$ *for some* k.
2. *If* $\mathrm{unref}(T) = x{:}T_1 \to T_2$ *then*
   (a) v *is* $\lambda x{:}T_1'.\ e_{12}$ *and* $T_1' \equiv T_1$ *for some* x, T_1', *and* e_{12}, *or*
   (b) v *is* $\langle T_1' \Rightarrow T_2' \rangle^l_\sigma$ *and* $\sigma(T_1') \equiv T_1$ *and* $\sigma(T_2') \equiv T_2$ *for some* T_1', T_2', σ, *and* l.
3. *If* $\mathrm{unref}(T) = \forall\alpha.T'$ *then* v *is* $\Lambda\alpha.\ e$ *for some* e.

Theorem 5 (Progress). *If* $\emptyset \vdash e : T$, *then either*

1. $e \longrightarrow e'$, *or*
2. e *is a result* r, *i.e., a value or blame.*

The following regularity property formalizes an important property of the type system: all contexts and types involved are well formed. This is critical for the proof of preservation: when a term raises blame, we must show that the blame is well typed. With regularity, we can immediately know that the original type is well formed.

Lemma 22 (Context and type well formedness). *(1) If* $\Gamma \vdash e : T$, *then* $\vdash \Gamma$ *and* $\Gamma \vdash T$; *and (2) if* $\Gamma \vdash T$ *then* $\vdash \Gamma$.

Theorem 6 (Preservation). *If* $\emptyset \vdash e : T$ *and* $e \longrightarrow e'$, *then* $\emptyset \vdash e' : T$.

Theorem 7 (Type Soundness). *If* $\emptyset \vdash e : T$ *and* $e \longrightarrow^* e'$ *and* e' *does not reduce, then* e' *is a result. Moreover, if* $e' = v$ *and* $T = \{x{:}T'' \mid e''\}$, *then* $[v/x]e'' \longrightarrow^* \mathsf{true}$.

Proof. The first half is shown by Theorems 5 and 6, and the second is by $\emptyset \vdash v : T$ and Lemma 13. □

3.4 Parametricity

Parametricity, which is coined by Wadler [119] and was originally called abstraction theorem [90], is a foundation of type abstraction [77] and information hiding [85, 107] in lambda calculi. Intuitively, it means that a polymorphic function behaves in the same way whatever types are substituted for the quantified type variable. We prove relational parametricity for three reasons: (1) it yields powerful reasoning techniques such as free theorems [119], contextual equivalence [87, 5], and the upcast lemma [14]; (2) it indicates that contracts do not interfere with type abstraction, i.e., that $\mathrm{F}^{\sigma}_{\mathrm{H}}$ supports polymorphism in the same way that System F does; (3) we want to correct Belo et al.

Closed results and terms $\boxed{r_1 \sim r_2 : T;\theta;\delta}$ $\boxed{e_1 \simeq e_2 : T;\theta;\delta}$

$$
\begin{aligned}
k \sim k : B;\theta;\delta &\iff k \in \mathcal{K}_B \\
v_1 \sim v_2 : \alpha;\theta;\delta &\iff \exists R\, T_1\, T_2,\ \alpha \mapsto R, T_1, T_2 \in \theta \wedge v_1\ R\ v_2 \\
v_1 \sim v_2 : (x{:}T_1 \to T_2);\theta;\delta &\iff \forall v_1' v_2',\ v_1' \sim v_2' : T_1;\theta;\delta \implies \\
&\qquad v_1\ v_1' \simeq v_2\ v_2' : T_2;\theta;\delta[(v_1', v_2')/x] \\
v_1 \sim v_2 : \forall\alpha.T;\theta;\delta &\iff \forall R\, T_1\, T_2,\ v_1\ T_1 \simeq v_2\ T_2 : T;\theta[\alpha \mapsto R, T_1, T_2];\delta \\
v_1 \sim v_2 : \{x{:}T \mid e\};\theta;\delta &\iff v_1 \sim v_2 : T;\theta;\delta \wedge \\
&\qquad [v_1/x]\theta_1(\delta_1(e)) \longrightarrow^* \mathsf{true} \wedge [v_2/x]\theta_2(\delta_2(e)) \longrightarrow^* \mathsf{true} \\
\Uparrow l \sim \Uparrow l : T;\theta;\delta & \\
\\
e_1 \simeq e_2 : T;\theta;\delta &\iff \exists r_1 r_2,\ e_1 \longrightarrow^* r_1 \wedge e_2 \longrightarrow^* r_2 \wedge r_1 \sim r_2 : T;\theta;\delta
\end{aligned}
$$

Types $\boxed{T_1 \simeq T_2 : *;\theta;\delta}$

$$
\begin{aligned}
B \simeq B : *;\theta;\delta & \\
\alpha \simeq \alpha : *;\theta;\delta & \\
x{:}T_{11} \to T_{12} \simeq x{:}T_{21} \to T_{22} : *;\theta;\delta &\iff T_{11} \simeq T_{21} : *;\theta;\delta \wedge \\
&\qquad \forall v_1 v_2,\ v_1 \sim v_2 : T_{11};\theta;\delta \implies \\
&\qquad\quad T_{12} \simeq T_{22} : *;\theta;\delta[(v_1, v_2)/x] \\
\forall\alpha.T_1 \simeq \forall\alpha.T_2 : *;\theta;\delta &\iff \forall R\, T_1'\, T_2',\ T_1 \simeq T_2 : *;\theta[\alpha \mapsto R, T_1', T_2'];\delta \\
\{x{:}T_1 \mid e_1\} \simeq \{x{:}T_2 \mid e_2\} : *;\theta;\delta &\iff T_1 \simeq T_2 : *;\theta;\delta \wedge \\
&\qquad \forall v_1 v_2,\ v_1 \sim v_2 : T_1;\theta;\delta \implies \\
&\qquad\quad [v_1/x]\theta_1(\delta_1(e_1)) \simeq [v_2/x]\theta_2(\delta_2(e_2)) : \mathsf{bool};\theta;\delta
\end{aligned}
$$

Open terms and types $\boxed{\Gamma \vdash \theta;\delta}$ $\boxed{\Gamma \vdash e_1 \simeq e_2 : T}$ $\boxed{\Gamma \vdash T_1 \simeq T_2 : *}$

$$
\begin{aligned}
\Gamma \vdash \theta;\delta &\iff \forall x{:}T \in \Gamma,\ \theta_1(\delta_1(x)) \simeq \theta_2(\delta_2(x)) : T;\theta;\delta \wedge \\
&\qquad \forall \alpha \in \Gamma, \exists R\, T_1\, T_2,\ \alpha \mapsto R, T_1, T_2 \in \theta \\
\Gamma \vdash e_1 \simeq e_2 : T &\iff \forall \theta\delta,\ \Gamma \vdash \theta;\delta \implies \theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_2)) : T;\theta;\delta \\
\Gamma \vdash T_1 \simeq T_2 : * &\iff \forall \theta\delta,\ \Gamma \vdash \theta;\delta \implies T_1 \simeq T_2 : *;\theta;\delta
\end{aligned}
$$

FIGURE 3.6: The logical relation for parametricity

[14] and Greenberg [43]. The proof is mostly standard—we define a (syntactic) logical relation on terms and types, where each type is interpreted as a relation on terms and the relation at type variables is given as a parameter—except that our logical relation includes not only well-typed terms and well-formed types but also ill-typed terms and ill-formed types.

3.4.1 Logical Relation

We begin by defining two relations: $r_1 \sim r_2 : T;\theta;\delta$ relates closed results, defined by induction on types; $e_1 \simeq e_2 : T;\theta;\delta$ relates closed expressions which evaluate to results in the first relation. (These results and expressions are *not* necessarily well typed. See the discussion below.) The definitions are shown in Figure 3.6.[6] Both relations have

[6] To save space, we write $\Uparrow l \sim \Uparrow l : T;\theta;\delta$ separately instead of manually adding such a clause for each type.

three indices: a (possibly open) type T, a substitution θ for type variables, and a substitution δ for term variables. A type substitution θ, which gives the interpretation of free type variables in T, maps type variables α to triples (R, T_1, T_2) comprising a binary relation R on closed results and two closed types T_1 and T_2, to be used as the concrete substitution of α on the left- and right-hand terms. (The results in R and the two types T_1 and T_2 *do not* have to be well typed/formed.) A term substitution δ maps from variables to pairs of closed (not necessarily well typed) values. We write projections δ_i $(i = 1, 2)$ to denote projections from this pair. We similarly write θ_i $(i = 1, 2)$ for a substitution that maps a type variable α to T_i where $\theta(\alpha) = (R, T_1, T_2)$. We also use the following notations:

$$\begin{array}{lcll} \theta[\alpha \mapsto R, T_1, T_2] & = & \theta \cup \{\alpha \mapsto R, T_1, T_2\} & \text{if } \alpha \notin \mathrm{dom}(\theta) \\ \delta[(v_1, v_2)/x] & = & \delta \cup \{x \mapsto v_1, v_2\} & \text{if } x \notin \mathrm{dom}(\delta) \end{array}$$

With these definitions out of the way, the result relation is mostly straightforward. First, $\Uparrow l$ is related to itself at every type. A base type B gives the identity relation on $\mathcal{K}_B$, the set of constants of type B. A type variable α simply uses the relation assumed in the substitution θ. Related functions map related arguments to related results. Type abstractions are related when their bodies are parametric in the interpretation of the type variable. Finally, two values are related at a refinement type when they are related at the underlying type and both satisfy the predicate; here, the predicate e gets closed by applying the substitutions. We require that both values satisfy their refinements rather than having the first satisfy the predicate iff the second does because we want to know that values related at refinement types *actually inhabit those types*, i.e., actually satisfy the predicates of the refinement. The $\sim$ relation on results is extended to the relation $\simeq$ on closed terms in a straightforward manner: terms are related if and only if they both terminate at related results. Divergent terms are not related to each other—though we will discover that divergent well typed terms do not exist in $\mathrm{F}^{\sigma}_{\mathrm{H}}$. We extend the relation to open terms, written $\Gamma \vdash e_1 \simeq e_2 : T$, relating open terms that are related when closed by any "Γ-respecting" pair of substitutions θ and δ (written $\Gamma \vdash \theta; \delta$, also defined in Figure 3.6).

To show that (well-typed) casts yield related results when applied to related inputs, we also need a relation on types $T_1 \simeq T_2 : *; \theta; \delta$; we define this relation in Figure 3.6. We can use the logical relation on results to handle the arguments of function types and refinement types. Note that the T_1 and T_2 in this relation are not necessarily closed; terms in refinement types, which should be related at bool, are closed by applying substitutions. In the function and refinement type cases, the relation on a smaller type is universally quantified over logically related values. There are two choices of the type at which they should be related (for example, the second line of the function type case could change T_{11} to T_{21}). It does not really matter which side we choose, since they are related types. We are "left-leaning." Finally, we lift the type relation to open types, writing $\Gamma \vdash T_1 \simeq T_2 : *$ when two types are equivalent for any Γ-respecting substitutions.

It is worth discussing two points peculiar to this formulation: terms in the logical relation are not necessarily well typed, and the type indices are open.

We allow any relation on terms to be used in θ; terms related at T need not be well typed at T. The standard formulation of a logical relation is well typed throughout, requiring that the relation R in every triple be well typed, only relating values of type T_1 to values of type T_2 (e.g., Pitts [87]). We have two motivations for allowing ill typed terms in our relation. First, functions of type $x{:}T_1 \to T_2$ must map related values

($v_1 \sim v_2 : T_1$) to related results... but at which type? While $[v_1/x]T_2$ and $[v_2/x]T_2$ are related in the type relation, terms that are well typed at one type will not necessarily be well typed at the other, whether definitions are left- or right-leaning. Second, this parametricity relation is designed so that a certain kind of casts have no effect, as Belo et al. [14] attempt. Ultimately, we would like to define a subtype relation $T_1 <: T_2$, and show what we call upcast lemma that, if $T_1 <: T_2$, then $\langle T_1 \Rightarrow T_2 \rangle^l \sim \lambda x{:}T_1.\ x : T_1 \to T_2$. That is, we want a cast $\langle T_1 \Rightarrow T_2 \rangle^l$, of type $T_1 \to T_2$, to be related to the identity $\lambda x{:}T_1.\ x$, of type $T_1 \to T_1$. There is one small hitch: $\lambda x{:}T_1.\ x$ has type $T_1 \to T_1$, not $T_1 \to T_2$! We therefore do not demand that two expressions related at T be well typed at T, and we allow *any* relation to be chosen as R.

The type indices of the term relation are not necessarily closed. Instead, just as the interpretation of free type variables in the logical relation's type index are kept in a substitution θ, we keep δ as a substitution for the free term variables that can appear in type indices. Keeping this substitution separate avoids a problem in defining the logical relation at function types. Consider a function type $x{:}T_1 \to T_2$: the *logical* relation says that values v_1 and v_2 are related at this type when they take related values to related results, i.e., if $v_1' \sim v_2' : T_1; \theta; \delta$, then we should be able to find $v_1\ v_1' \simeq v_2\ v_2'$ at some type. The question here is which type index we should use. If we keep type indices closed (with respect to term variables), we cannot use T_2 on its own—we have to choose a binding for x! Knowles and Flanagan [64] deal with this problem by introducing the "wedge product" operator, which merges two types—one with v_1' substituted for x and the other with v_2' for x—into one. Instead of substituting eagerly, we put both bindings in δ and apply them when needed—the refinement type case. We think this formulation is more uniform with regard to free term/type variables, since eager substitution is a non-starter for type variables, anyway.

As we developed the original proof [14], we found that the (E_REFL) rule $\langle T \Rightarrow T \rangle^l\ v \rightsquigarrow v$ is not just a convenient way to skip decomposing a trivial cast into smaller trivial casts (when T is a polymorphic or dependent function type); (E_REFL) is, in fact, crucial to obtaining parametricity in this syntactic setting. On the one hand, the evaluation of well-typed programs never encounters casts with uninstantiated type variables—a key property of our evaluation relation. On the other hand, by parametricity, we expect every value of type $\forall\alpha.\alpha \to \alpha$ to behave the same as the polymorphic identity function (modulo blame). One of the values of this type is $\Lambda\alpha.\ \langle \alpha \Rightarrow \alpha \rangle^l$. Without (E_REFL), however, applying this type abstraction to a compound type, say $\mathsf{bool} \to \mathsf{bool}$, and a function f of type $\mathsf{bool} \to \mathsf{bool}$ would return, by (E_FUN), a wrapped version of f that is syntactically different from the f we passed in—that is, the function broke parametricity! We expect the returned value should behave the same as the input, though—the results are just *syntactically* different. With (E_REFL), $\langle T \Rightarrow T \rangle^l$ returns the input immediately, regardless of T—just as the identity function. So, this rule is a technical necessity, ensuring that casts containing type variables behave parametrically.

3.4.2 Parametricity

Now we can set about proving parametricity. The proof of parametricity (Theorem 8) of $\mathrm{F}^{\sigma}_{\mathrm{H}}$ is trickier than that of the standard polymorphic lambda calculus, due to (1) dependent functions, (2) type convertibility, and (3) casts. Before stating parametricity, we discuss these issues; see Appendix for the proofs of it and lemmas.

In $\mathrm{F}^{\sigma}_{\mathrm{H}}$, It is not as easy as in System F to show that a well-typed term application is logically related to itself due to dependent function types. To see the reason, let us

Complexity of casts

$$
\begin{array}{ll}
cc(\langle T \Rightarrow T\rangle^l) & = 1 \\
cc(\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l) & = cc(\langle [y/x]\,T_{12} \Rightarrow T_{22}\rangle^l) + cc(\langle T_{21} \Rightarrow T_{11}\rangle^l) + 1 \\
 & \qquad (y \text{ is fresh}) \\
cc(\langle \forall\alpha.\, T_1 \Rightarrow \forall\alpha.\, T_2\rangle^l) & = cc(\langle T_1 \Rightarrow T_2\rangle^l) + 1 \\
cc(\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l) & = cc(\langle T_1 \Rightarrow T_2\rangle^l) + 1 \\
 & \quad (\text{if } T_2 \neq \{x{:}T_1 \mid e\} \text{ and } T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e'\}) \\
cc(\langle T_1 \Rightarrow \{x{:}T_1 \mid e\}\rangle^l) & = 1 \\
cc(\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l) & = cc(\langle T_1 \Rightarrow T_2\rangle^l) + 2 \\
 & \quad (\text{if } T_1 \neq T_2 \text{ and } T_1 \text{ is not a refinement type})
\end{array}
$$

FIGURE 3.7: Complexity of casts

consider term application $v_1\, v_2$ such that v_1 and v_2 are typed at $x{:}T_1 \to T_2$ and T_1, respectively. Parametricity states that, if v_1 and v_2 are logically related to themselves with θ and δ, respectively, then so is $v_1\, v_2$ at $[v_2/x]\,T_2$. The definition of the logical relation, however, states only that $v_1\, v_2$ are logically related to T_2, not $[v_2/x]\,T_2$, with θ and $\delta[(v_2, v_2)/x]$. Fortunately, as expected, these are equivalent: $v_1\, v_2$ are logically related to itself at $[v_2/x]\,T_2$ with θ and δ iff $v_1\, v_2$ are logically related to itself at T_2 with θ and $\delta[(v_2, v_2)/x]$. Term compositionality stated below generalizes this.

Lemma 23 (Term compositionality). *If* $\theta_1(\delta_1(e)) \longrightarrow^* v_1$ *and* $\theta_2(\delta_2(e)) \longrightarrow^* v_2$ *then* $r_1 \sim r_2 : T; \theta; \delta[(v_1, v_2)/x]$ *iff* $r_1 \sim r_2 : [e/x]\,T; \theta; \delta$.

For a similar reason, we show type compositionality, which is used also in other polymorphic lambda calculi (e.g., Pitts [87]). In what follows, we write $R_{T,\theta,\delta}$ for $\{(r_1, r_2) \mid r_1 \sim r_2 : T; \theta; \delta\}$.

Lemma 24 (Type compositionality).
$r_1 \sim r_2 : T; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$ *iff* $r_1 \sim r_2 : [T'/\alpha]\,T; \theta; \delta$.

For the typing rule (T_CONV) with type convertibility, we have to show that terms are logically related to themselves at convertible types.

Lemma 25 (Convertibility). *If* $T_1 \equiv T_2$ *then* $r_1 \sim r_2 : T_1; \theta; \delta$ *iff* $r_1 \sim r_2 : T_2; \theta; \delta$.

Showing that casts are logically related to themselves is the most cumbersome case in the proof of parametricity. We prove it by induction on a cast complexity metric, cc, defined in Figure 3.7. The complexity of a cast is the number of steps it and its subparts can take. This definition is carefully dependent on our definition of type compatibility and our cast reduction rules. Doing induction on this metric greatly simplifies the proof: we show that stepping casts at related types yields either related non-casts, or lower complexity casts between related types. Note that we omit the σ, since the evaluation of casts *does not depend on delayed substitutions*. It may be easier for the reader to think of $cc(\langle T_1 \Rightarrow T_2\rangle^l)$ as a three argument function—taking two types and a blame label—rather than a single argument function taking a cast. The cc is well defined though the case for casts between dependent function types chooses an *arbitrary* fresh variable, because, for any variable y and z, $cc(\langle [y/x]\,T_1 \Rightarrow T_2\rangle^l) = cc(\langle [z/x]\,T_1 \Rightarrow T_2\rangle^l)$ if y and z do not occur free in T_1 and T_2.

Lemma 26 (Cast reflexivity). *If* $\vdash \Gamma$ *and* $T_1 \parallel T_2$ *and* $\Gamma \vdash \sigma(T_1) \simeq \sigma(T_1) : *$ *and* $\Gamma \vdash \sigma(T_2) \simeq \sigma(T_2) : *$ *and* $\mathrm{AFV}(\sigma) \subseteq \mathrm{dom}(\Gamma)$, *then* $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l_\sigma \simeq \langle T_1 \Rightarrow T_2\rangle^l_\sigma : \sigma(\_{:}T_1 \to T_2)$.

Finally, we can prove relational parametricity—every well-typed term (under Γ) is related to itself for any Γ-respecting substitutions.

Theorem 8 (Parametricity). *(1) If* $\Gamma \vdash e : T$ *then* $\Gamma \vdash e \simeq e : T$*; and (2) if* $\Gamma \vdash T$ *then* $\Gamma \vdash T \simeq T : *$.

We have that logically related programs are by definition *behaviorally equivalent*: if $\emptyset \vdash e_1 \simeq e_2 : B$, then e_1 and e_2 coterminate at equal results. Ideally, logically related terms are also contextually equivalent and vice versa, but we leave study of this problem for future work.

3.5 Three Versions of F_H

We compare F_H^σ with two prior formulations of F_H without delayed substitution: Belo et al. [14] from ESOP 2011 and Greenberg's thesis [43]. Both of these define variants of F_H, claiming type soundness, parametricity, and upcast elimination. All of these results depend on two properties of the F_H type conversion relation: substitutivity (Lemma 14) and cotermination (Lemma 11).

3.5.1 F_H 1.0: Belo et al.'s Work

Belo et al. [14] got rid of subtyping and explicitly used the symmetric, transitive closure of parallel reduction $\Rrightarrow$ (Figure 3.8, quoted from Greenberg [43]) as the conversion relation. (Parallel reduction is reflexive by definition.) The use of parallel reduction is inspired by Greenberg et al. [44], in which type soundness of λ_H is proved by using cotermination and another property called *substitutivity* (if $e_1 \Rrightarrow e_2$ and $e_1' \Rrightarrow e_2'$ then $[e_1'/x]e_1 \Rrightarrow [e_2'/x]e_2$) of parallel reduction. These properties were needed also for type soundness of F_H. Unfortunately, it turns out that parallel reduction in F_H is *not* substitutive—the proof was wrong—and cotermination, which was left as a conjecture ([14], p. 15), does not hold, either. Figure 3.9 offers three counterexamples:[7] two to substitutivity and one to cotermination.

Why does not substitutivity hold in F_H, when it did (so easily) in λ_H? Sources of the trouble are that (1) the F_H cast rules depend upon certain (syntactic) equalities between types and that (2) parallel reduction is defined over open terms. As a result, substitution may change reduction rules to be applied—both counterexamples to substitutivity in Figure 3.9 take advantage of it.

Cotermination breaks also because substitutions can affect which reduction rule applies to a cast, which in turn can force us to perform checks under one substitution that are not performed under another, related one (counterexample 3 in Figure 3.9).

3.5.2 F_H 2.0: Greenberg's Thesis

In his thesis, Greenberg tried to correct this problem using a fix due to Sekiyama: he takes *common-subexpression reduction* (CSR) as the conversion relation [43]. We repeat F_H^σ's identical definition of CSR (Figure 3.5) again here, in Figure 3.10. As we can see from the definition, CSR is designed to be substitutive (and *is* substitutive). However, cotermination still fails: we can construct ill-typed terms that do not satisfy cotermination in Greenberg's operational semantics—they look like the term in counterexample 3

[7] The first two have been shown in Greenberg [43] but they are discovered by Igarashi, Greenberg, and Sekiyama.

Parallel term reduction $\boxed{e_1 \Rrightarrow e_2}$

$$\frac{v_i \Rrightarrow v_i'}{op\,(v_1,\,\ldots,\,v_n) \Rrightarrow [\![\mathrm{op}]\!]\,(v_1',\,\ldots,\,v_n')}\ \text{EP\_ROP} \qquad \frac{e_{12} \Rrightarrow e_{12}' \quad v_2 \Rrightarrow v_2'}{(\lambda x{:}T.\ e_{12})\ v_2 \Rrightarrow [v_2'/x]e_{12}'}\ \text{EP\_RBETA}$$

$$\frac{e \Rrightarrow e' \quad T_2 \Rrightarrow T_2'}{(\Lambda\alpha.\ e)\ T_2 \Rrightarrow [T_2'/\alpha]e'}\ \text{EP\_RTBETA} \qquad \frac{v \Rrightarrow v'}{\langle T \Rightarrow T\rangle^l\, v \Rrightarrow v'}\ \text{EP\_RREFL}$$

$$\frac{T_2 \neq \{x{:}T_1 \mid e\} \quad T_2 \neq \{y{:}\{x{:}T_1 \mid e\} \mid e_2\} \quad T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2' \quad v \Rrightarrow v'}{\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle^l\, v \Rrightarrow \langle T_1' \Rightarrow T_2'\rangle^l\, v'}\ \text{EP\_RFORGET}$$

$$\frac{T_1 \neq T_2 \quad T_1 \neq \{x{:}T \mid e\} \quad T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2' \quad e \Rrightarrow e' \quad v \Rrightarrow v'}{\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l\, v \Rrightarrow \langle T_2' \Rightarrow \{x{:}T_2' \mid e'\}\rangle^l\,(\langle T_1' \Rightarrow T_2'\rangle^l\, v')}\ \text{EP\_RPRECHECK}$$

$$\frac{T \Rrightarrow T' \qquad e \Rrightarrow e' \qquad v \Rrightarrow v'}{\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l\, v \Rrightarrow \langle \{x{:}T' \mid e'\}, [v'/x]e', v'\rangle^l}\ \text{EP\_RCHECK}$$

$$\frac{v \Rrightarrow v'}{\langle \{x{:}T \mid e_1\}, \mathsf{true}, v\rangle^l \Rrightarrow v'}\ \text{EP\_ROK} \qquad \frac{}{\langle \{x{:}T \mid e_1\}, \mathsf{false}, v\rangle^l \Rrightarrow \Uparrow l}\ \text{EP\_RFAIL}$$

$$\frac{\begin{array}{c} x{:}T_{11} \to T_{12} \neq x{:}T_{21} \to T_{22} \\ T_{11} \Rrightarrow T_{11}' \qquad T_{12} \Rrightarrow T_{12}' \qquad T_{21} \Rrightarrow T_{21}' \qquad T_{22} \Rrightarrow T_{22}' \qquad v \Rrightarrow v' \end{array}}{\begin{array}{l} \langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l\, v \Rrightarrow \\ \qquad \lambda x{:}T_{21}'.\ (\langle [\langle T_{21}' \Rightarrow T_{11}'\rangle^l\, x/x]\, T_{12}' \Rightarrow T_{22}'\rangle^l\,(v'\,(\langle T_{21}' \Rightarrow T_{11}'\rangle^l\, x))) \end{array}}\ \text{EP\_RFUN}$$

$$\frac{\forall\alpha.T_1 \neq \forall\alpha.T_2 \qquad T_1 \Rrightarrow T_1' \qquad T_2 \Rrightarrow T_2' \qquad v \Rrightarrow v'}{\langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l\, v \Rrightarrow \Lambda\alpha.\ (\langle T_1' \Rightarrow T_2'\rangle^l\,(v'\,\alpha))}\ \text{EP\_RFORALL}$$

$$\frac{}{e \Rrightarrow e}\ \text{EP\_REFL} \qquad \frac{T_1 \Rrightarrow T_1' \quad e_{12} \Rrightarrow e_{12}'}{\lambda x{:}T_1.\ e_{12} \Rrightarrow \lambda x{:}T_1'.\ e_{12}'}\ \text{EP\_ABS} \qquad \frac{e_1 \Rrightarrow e_1' \quad e_2 \Rrightarrow e_2'}{e_1\ e_2 \Rrightarrow e_1'\ e_2'}\ \text{EP\_APP}$$

$$\frac{e \Rrightarrow e'}{\Lambda\alpha.\ e \Rrightarrow \Lambda\alpha.\ e'}\ \text{EP\_TABS} \qquad \frac{e_1 \Rrightarrow e_1' \quad T_2 \Rrightarrow T_2'}{e_1\ T_2 \Rrightarrow e_1'\ T_2'}\ \text{EP\_TAPP}$$

$$\frac{e_i \Rrightarrow e_i'}{op\,(e_1,\,\ldots,\,e_n) \Rrightarrow op\,(e_1',\,\ldots,\,e_n')}\ \text{EP\_OP} \qquad \frac{T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2'}{\langle T_1 \Rightarrow T_2\rangle^l \Rrightarrow \langle T_1' \Rightarrow T_2'\rangle^l}\ \text{EP\_CAST}$$

$$\frac{T \Rrightarrow T' \quad e \Rrightarrow e'}{\langle T, e, k\rangle^l \Rrightarrow \langle T', e', k\rangle^l}\ \text{EP\_CHECK} \qquad \frac{}{E\,[\Uparrow l] \Rrightarrow \Uparrow l}\ \text{EP\_BLAME}$$

Parallel type reduction $\boxed{T_1 \Rrightarrow T_2}$

$$\frac{}{T \Rrightarrow T}\ \text{EP\_TREFL} \qquad \frac{\sigma_1 \longrightarrow^* \sigma_2 \quad T_1 \Rrightarrow T_2}{\{x{:}T_1 \mid \sigma_1(e)\} \Rrightarrow \{x{:}T_2 \mid \sigma_2(e)\}}\ \text{EP\_TREFINE}$$

$$\frac{T_1 \Rrightarrow T_1' \quad T_2 \Rrightarrow T_2'}{x{:}T_1 \to T_2 \Rrightarrow x{:}T_1' \to T_2'}\ \text{EP\_TFUN} \qquad \frac{T \Rrightarrow T'}{\forall\alpha.T \Rrightarrow \forall\alpha.T'}\ \text{EP\_TFORALL}$$

FIGURE 3.8: Parallel reduction (for open terms).

Counterexample 1: substitutivity

Let T be a type with a free variable x.

$$\begin{array}{rcl} e_1 & = & \langle T \Rightarrow \{y{:}[5/x]\,T \mid \mathsf{true}\}\rangle^l\, 0 \\ e_2 & = & \langle [5/x]\,T \Rightarrow \{y{:}[5/x]\,T \mid \mathsf{true}\}\rangle^l\, (\langle T \Rightarrow [5/x]\,T\rangle^l\, 0) \\ e_1' = e_2' & = & 5 \end{array}$$

Observe that $e_1' \Rrightarrow e_2'$ (by (EP_REFL)) and $e_1 \Rrightarrow e_2$ (by (EP_RPRECHECK)) but $[5/x]e_1 = \langle [5/x]\,T \Rightarrow \{y{:}[5/x]\,T \mid \mathsf{true}\}\rangle^l\, 0 \Rrightarrow \langle \{y{:}[5/x]\,T \mid \mathsf{true}\}, \mathsf{true}, 0\rangle^l$ by (EP_RCHECK), not $[5/x]e_2$. Note that the definition of substitution $[e'/x]e$ is a standard one, in which substitution goes down into casts.

Counterexample 2: substitutivity

Let T_2 be a type with a free variable x.

$$\begin{array}{rcl} e_1 & = & \langle T_1 \to T_2 \Rightarrow\ T_1 \to [5/x]\,T_2\rangle^l\, v \\ e_2 & = & \lambda y{:}T_1.\ \langle T_2 \Rightarrow [5/x]\,T_2\rangle^l\, (v\ (\langle T_1 \Rightarrow\ T_1\rangle^l\, y)) \\ e_1' = e_2' & = & 5 \end{array}$$

Observe that $e_1' \Rrightarrow e_2'$ (by (EP_REFL)) and $e_1 \Rrightarrow e_2$ (by (EP_RFUN)). We have $[5/x]e_1 = \langle T_1 \to [5/x]\,T_2 \Rightarrow\ T_1 \to [5/x]\,T_2\rangle^l\, v \Rrightarrow [5/x]v$ by (EP_RREFL), not $[5/x]e_2$.

Counterexample 3: cotermination

$$\begin{array}{rcl} e & = & \langle \{x{:}\mathsf{bool} \mid \mathsf{false}\} \Rightarrow \{x{:}\mathsf{bool} \mid y\}\rangle^l\, \mathsf{true} \\ e_1 & = & 0 = 5 \\ e_2 & = & \mathsf{false} \end{array}$$

Observe that $e_1 \longrightarrow e_2$ (and so $e_1 \Rrightarrow e_2$, by (EP_ROP)) and cotermination says that $[e_1/y]e$ terminates at a value iff so does $[e_2/x]e$. Here, by (E_CHECK), $[e_1/y]e \longrightarrow \langle \{x{:}\mathsf{bool} \mid e_1\}, e_1, \mathsf{true}\rangle^l \longrightarrow^* \Uparrow l$ but by (E_REFL), $[e_2/x]e \longrightarrow \mathsf{true}$.

FIGURE 3.9: Counterexamples to substitutivity and cotermination of parallel reduction in $\mathrm{F_H}$

Conversion $\boxed{\sigma_1 \longrightarrow^* \sigma_2}$ $\boxed{T_1 \equiv T_2}$

$$\sigma_1 \longrightarrow^* \sigma_2 \iff \begin{array}{l}\mathsf{dom}(\sigma_1) = \mathsf{dom}(\sigma_2) \subset \mathsf{TmVar} \wedge \\ \forall x \in \mathsf{dom}(\sigma_1).\ \sigma_1(x) \longrightarrow^* \sigma_2(x)\end{array}$$

$$\frac{}{\alpha \equiv \alpha}\ \text{C\_VAR} \qquad \frac{}{B \equiv B}\ \text{C\_BASE} \qquad \frac{\sigma_1 \longrightarrow^* \sigma_2 \quad T_1 \equiv T_2}{\{x{:}T_1 \mid \sigma_1(e)\} \equiv \{x{:}T_2 \mid \sigma_2(e)\}}\ \text{C\_REFINE}$$

$$\frac{T_1 \equiv T_1' \quad T_2 \equiv T_2'}{x{:}T_1 \to T_2 \equiv x{:}T_1' \to T_2'}\ \text{C\_FUN} \qquad \frac{T \equiv T'}{\forall\alpha.T \equiv \forall\alpha.T'}\ \text{C\_FORALL}$$

$$\frac{T_2 \equiv T_1}{T_1 \equiv T_2}\ \text{C\_SYM} \qquad \frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}\ \text{C\_TRANS}$$

FIGURE 3.10: Type conversion via common-subexpression reduction

(Figure 3.9). The essential issue is that we can fire (E_REFL) under one substitution and force a check under another. If the term is ill typed, then we have no way of knowing whether the argument of the cast satisfies its input type—so the check can fail where (E_REFL) succeeded. Well typed terms do not have this problem, but we need our conversion relation to prove progress and preservation—we cannot use arguments about typing in our proof of cotermination. In short, Greenberg's Conjecture 3.2.1 on page 88 is false; it seems that the evaluation relation is defined in such a way that substitutions can affect which cast reduction rules are chosen.

3.5.3 $\mathbf{F}^{\sigma}_{\mathbf{H}}$

Our calculus, $\mathrm{F}^{\sigma}_{\mathrm{H}}$, can see statically which cast reduction rule is chosen thanks to our definition of substitution (Definition 2). In Lemma 11, we show that terms related by CSR coterminate at true using $\mathrm{F}^{\sigma}_{\mathrm{H}}$'s substitution semantics; this is enough to prove type soundness and parametricity. F_{H} tried to use entirely syntactic techniques to achieve type soundness, avoiding the semantic techniques necessary for λ_{H}. But we failed: we need to prove cotermination to get type soundness; our proof amounts to showing that type conversion is a weak bisimulation. Our metatheory is, on the one hand, simpler than that of Greenberg et al. [44], which needs cotermination *and* semantic type soundness. On the other hand, we must use a nonstandard substitution operation, which is a hassle.

Chapter 4

Manifest Contracts with Algebraic Datatypes

This chapter extends manifest contracts with algebraic datatypes. The idea of introducing algebraic datatypes to manifest contracts is natural because manifest contracts make fine-grained specifications explicit as types and algebraic datatypes, which are often formalized as the combination of sum types and recursive types, are common in type systems with data structures, especially, in functional programming (e.g., [40, 59, 15, 68, 66]); in fact, as we will see later, the manifest contract calculus given in this chapter can represent various specifications on data structures. We also investigate representation of contracts on datatypes through our manifest contract calculus with algebraic datatypes. In particular, we focus on a computational aspect of contract checking. Since contracts were originally conceived as a mechanism to check software properties dynamically, it was also clear that contract checking could cause significant overhead. This overhead is specially notable when we consider contracts on data structures because naive contract checking for datatypes can make asymptotic time complexity worse, as pointed out by Findler et al. [36]. To see how representation of contracts affects computational efficiency, we start with comparison of two simple, complementary approaches to giving refinements to data structures.

Refinements on type constructors versus refinements on data constructors There are two approaches to specifying contracts for data structures. One is to put refinements on the type constructor for a plain data structure and the other is to put refinements on (types for) data constructors. For example, a type slist for sorted integer lists can be written $\{x{:}\mathsf{int\ list} \mid \mathsf{sorted}\ x\}$ in which sorted is a familiar Boolean function that returns whether the argument list is sorted in the former, or defined as another datatype with refined cons of type $x{:}\mathsf{int} \times \{xs{:}\mathsf{slist} \mid \mathsf{nil}\ xs \text{ or } x \leq \mathsf{head}\ xs\} \to \mathsf{slist}$ in the latter. Here, the argument type is a dependent product type, expressing the relationship between the two components in the pair. However, as pointed out by Findler, Guo, and Rogers [36], neither approach by itself is very satisfactory.

On the one hand, the former approach, which is arguably easier for ordinary programmers, may cause significant overhead in contract checking to make asymptotic time complexity worse. To see how it happens, let us consider function `insert_sort` for insertion sort. The sorting function and its auxiliary function insert can be defined in the ML-like syntax as follows.

```
type slist1 = {x:int list | sorted x}

let rec insert (x:int) (l:slist1) : slist1 =
  match l with
  | []    -> ⟨slist1 ⇐ int list⟩^ℓ1 [x]
```

```
  | y::ys ->
    if x <= y then ⟨slist1 ⇐ int list⟩^ℓ2 (x::l)
    else ⟨slist1 ⇐ int list⟩^ℓ3
      (y::(insert x (⟨slist1 ⇐ int list⟩^ℓ4 ys)))

let rec insert_sort (l:int list) : slist1 =
  match l with
  | []    -> []
  | x::xs -> insert x (insert_sort xs)
```

Without gray-colored casts, `insert_sort` would be an ordinary insertion-sort function. However, in `insert`, the four subexpressions `[x]`, `x::l`, `y::(insert x ys)` and `ys`, which are given type `int list`, are actually expected to have type `slist1` by the context. To fill the gap[1], we have to check whether these subexpressions satisfy the contract `sorted`. Notice that these casts cannot be eliminated by simple subtype checking because `int list` is obviously not a subtype of `slist1`. As far as we understand, existing technologies cannot verify these casts will be successful, at least, without giving hints to the verifier. Unfortunately, leaving these casts (especially ones with ℓ_2, ℓ_3, and ℓ_4) has an unpleasant effect: They traverse the entire lists to check sortedness, even though the lists have already been sorted, making the asymptotic time complexity of `insert` from $O(m)$ to $O(m^2)$, where m stands for the length of the input.

On the other hand, the latter approach, which exploits refinement in argument types of data constructors, does not have this efficiency problem (if not always). For example, we can define sorted lists as a datatype with refined constructors:

```
type slist2 =
  SNil
| SCons of
    x:int × {xs:slist2 | nil xs or x <= head xs}
```

Here, `nil` and `head` are functions[2] that return whether a given list is empty and the first element of a given list, respectively, and a type of the form $x{:}T_1 \times T_2$ is a dependent product type, which denotes pairs (v_1, v_2) of values such that v_1 and v_2 are of types T_1 and $T_2\{v_1/x\}$, respectively. So, `SCons` takes an integer `x` and a (sorted) list whose head (if any) is equal to or greater than `x`. Using `slist2`, we can modify the functions `insert` and `insert_sort` to perform less dynamic checking.

```
let rec insert' (x:int) (l:slist2) : slist2 =
  match l with
  | SNil -> SCons (x,⟨slist_x ⇐ slist2⟩^ℓ SNil)
  | SCons (y, ys) ->
    if x <= y then SCons (x,⟨slist_x ⇐ slist2⟩^ℓ l)
    else SCons
           (y,⟨slist_y ⇐ slist2⟩^ℓ (insert' x ys))

let rec insert_sort' (l:int list) : slist2 =
  match l with
  | []    -> SNil
  | x::xs -> insert' x (insert_sort' xs)
```

[1] Actually, there are subexpressions whose expected types are `int list` but actual types are `slist1`. We assume that `slist1` can be converted to `int list` for free.

[2] Precisely speaking, these functions have to be defined together with `slist2` but we omit them for brevity.

Here, $\texttt{slist}_e$ stands for `{xs:slist2 | nil xs or` e `<= head xs}`. Since the contract in the cast $\langle \texttt{slist}_x \Leftarrow \texttt{slist2} \rangle^\ell$ does not traverse `xs`, it is more efficient than the first definition; in fact, the time complexity of `insert'` remains to be $O(m)$. Moreover, it would be possible to eliminate the cast on `l` by collecting conditions (`l` is equal to `SCons(y, ys)` and `x <= y`) guarding this branch [91]. (It is more difficult to eliminate the other cast because the verifier would have to know that the head of the list returned by the recursive call to `insert'` is greater than `y`.)

However, this approach has complementary problems. First, we have to maintain the predicate function `sorted` and the corresponding type definition `slist2` separately. Second, it may not be a trivial task to write down the specification as data constructor refinement. For example, consider the type of lists whose elements contain a given integer n. A refinement type of such lists can be written `{l:int list | member n l}` using the familiar `member` function. One possible datatype definition corresponding to the refinement type above would be given by using an auxiliary datatype, parameterized over an integer `n` and a Boolean flag `p` to represent whether `n` has to appear in a list.

```
type incl_aux ⟨p:bool, n:int⟩ =
  LNil  of {unit|not p}
| LCons of x:int × incl_aux ⟨not (x=n) and p, n⟩

type list_including ⟨n:int⟩ = incl_aux ⟨true,n⟩
```

(Notice that `incl_aux⟨false,n⟩` is essentially `int list` and, if a list without n is given type `incl_aux`$\langle p, n \rangle$, then p must be `false`.) We do not think it is as easy to come up with a datatype definition like this as the refinement type above.

Another issue is interoperability between a plain type and its refined versions: Just as casts between `slist1` and `int list` are allowed, we would hope that the language supports casts between `slist2` and `int list`, even when they have different sets of data constructors. Such interoperability is crucial for code reuse [36]—without it, we must reimplement many list-processing functions, such as `sort`, `member`, `map`, etc., every time a refined datatype is given. As pointed out in Vazou, Rondon, and Jhala [116], one can give one generic datatype definition, which is parameterized over predicates on components of the datatype, and instantiate it to obtain plain and sorted list types but, as we will show later, refined datatype definitions may naturally come with more data constructors than the plain one, in which case parameterization would not work (the number of constructors is the same for every instantiation).

In short, the two approaches are complementary.

Contributions Our work aims at taking the best of both approaches. First, we give a provably correct syntactic translation from refinements on type constructors, such as the Boolean function `sorted`, to equivalent type definitions where data constructors are refined, namely, `slist2`. This translation is closely related to the work by Atkey, Johann, and Ghani [10] and McBride [70], also concerned about systematic generation of a new datatype; see Section 5.6 for comparison. Second, we extend casts so that casts between similar but different datatypes (what we call compatible types, which are declared explicitly in datatype definitions) are possible. For example, $\langle$`slist2 ⇐ int list`$\rangle^\ell$`(1 :: 2 :: [])` yields `SCons(1, SCons(2, SNil))`, whereas $\langle$`slist2 ⇐ int list`$\rangle^\ell$ `(1 :: 0 :: [])` raises blame ℓ. Thanks to the two ideas, a programmer can automatically derive a datatype definition from a familiar Boolean function, exploit the resulting datatype for less dynamic checking as we saw in

the example of insertion sort, and also use it, when necessary, as if it were a refinement type using the Boolean function.

We formalize these ideas as a manifest contract calculus $\lambda^H_{\mathbf{dt}}$ and prove basic properties such as progress and preservation. $\lambda^H_{\mathbf{dt}}$ is defined by following the ideas in Chapter 3—i.e., $\lambda^H_{\mathbf{dt}}$ does not have subsumption (for subtyping) and its cast semantics is designed to be insensitive to substitutions—but it does not use delayed substitution, unlike $\mathrm{F}^{\sigma}_{\mathrm{H}}$ given in Chapter 3. In Chapter 3, delayed substitution played a crucial role to determine how casts reduce statically (in the presence of the reduction rule (E_REFL) to eliminate reflexive casts) and prove parametricity whereas it made the metatheory of $\mathrm{F}^{\sigma}_{\mathrm{H}}$ more complicated than F_{H} [14, 43]. To make the metatheory of $\lambda^H_{\mathbf{dt}}$ as simple as possible, we do not introduce delayed substitution in this chapter. The lack of delayed substitution, however, raises one question: without delayed substitutions, how can we obtain cast semantics where substitutions do not affect the behavior of casts? We achieve it by designing cast semantics where all refinements on target types of casts are checked regardless of what values are substituted.

Our contributions are summarized as follows:

- We propose casts between compatible datatypes to enhance interoperability among a plain datatype and its refined versions.
- We define a manifest contract calculus $\lambda^H_{\mathbf{dt}}$ to formalize the semantics of the casts.
- We formally define a translation from refinements on type constructors to type definitions where data constructors are refined and prove the translation is correct.

We note that this work gives type translation but does *not* give translation from a program with refinement types to one with refined datatypes, so if a programmer has a program with, for example, `slist1`, then he has to rewrite it to one with a datatype like `slist2` by hand. Automatic program transformation is left as future work.

Outline The rest of this chapter is organized as follows. Section 4.1 gives an overview of our datatype mechanism and Section 4.2 formalizes $\lambda^H_{\mathbf{dt}}$, shows its type soundness, and compares it with $\mathrm{F}^{\sigma}_{\mathrm{H}}$ in detail. Section 4.3 gives a translation from refinement types to datatypes and proves its correctness.

4.1 Overview

In this section, we informally describe our proposals of datatype definitions, casts between compatible datatypes, and translation, mainly by means of examples.

As we have seen already in the example of sorted lists, our datatype definition allows the argument types of data constructors to be refined using the set comprehension notation $\{x{:}T \mid e\}$ and dependent product types $x{:}T_1 \times T_2$. We also allow parameterization over terms as in `incl_aux` in the previous section.

4.1.1 Casts for Datatypes

As we have discussed in the beginning of this chapter, in order to enhance interoperability between refined datatypes, we allow casts between two different datatypes if they are "compatible"; in other words, compatibility is used to disallow casts between unrelated types (for example, the integer type and a function type). Compatibility for

types other than datatypes means that two types are the same by ignoring refinements; compatibility for datatypes means that there is a correspondence between the sets of the data constructors from two datatypes and the argument types of the corresponding constructors are also compatible. In our proposal, a correspondence between constructors has to be explicitly declared. So, the type `slist2` in the previous section is actually written as follows:

```
type slist2 =
  SNil  || []
| SCons || (::) of
    x:int × {xs:slist2 | nil xs or x <= head xs}
```

The symbol `||` followed by a data constructor from an existing datatype declares how constructors correspond. The types `int list` and `slist2` are compatible because both `SNil` and `[]` take no arguments and the argument type `x:int × {xs:slist2 | nil xs or x <= head xs}` of `SCons` is compatible with `int × int list` of `(::)`. (Precisely speaking, compatibility is defined coinductively.) Readers may think that explicit declaration of a correspondence of data constructors seems cumbersome. However, we could replace these declarations by a compatibility declaration for type names as `slist2 || int list` and let the system infer the correspondence between data constructors. Such inference is easy for many cases, where the argument types of data constructors are of different shapes, as in this example.

A cast for datatypes converts data constructors to the corresponding ones and puts a new cast on components. For example, $\langle \mathsf{slist} \Leftarrow \mathsf{int\ list} \rangle^{\ell}\,(1 :: 2 :: 3 :: [\,])$ reduces to $\mathsf{SCons}\,(1, \mathsf{SCons}\,(2, \mathsf{SCons}\,(3, \mathsf{SNil})))$ as follows:

$$
\begin{aligned}
& \langle \mathsf{slist} \Leftarrow \mathsf{int\ list} \rangle^{\ell}\,(1 :: 2 :: 3 :: [\,]) \\
\to\ & \mathsf{SCons}(\langle x{:}\mathsf{int} \times \{xs{:}\mathsf{slist} \mid \mathsf{nil}\ xs\ \mathsf{or}\ x \leq \mathsf{head}\ xs\} \Leftarrow \mathsf{int} \times \mathsf{int\ list} \rangle^{\ell} \\
& \qquad (1, 2 :: 3 :: [\,])) \\
\to\ & \mathsf{SCons}(1, \langle \{xs{:}\mathsf{slist} \mid \mathsf{nil}\ xs\ \mathsf{or}\ 1 \leq \mathsf{head}\ xs\} \Leftarrow \mathsf{int\ list} \rangle^{\ell}\,(2 :: 3 :: [\,])) \\
\to\ & \ldots \\
\to\ & \mathsf{SCons}\,(1, \mathsf{SCons}\,(2, \mathsf{SCons}\,(3, \mathsf{SNil})))
\end{aligned}
$$

In the example above, the correspondence between data constructors is bijective but we actually allow nonbijective correspondence, too. This means that a new datatype can have two or more (or even no) data constructors corresponding to a single data constructor from an existing type. For example, an alternative definition of `list_including` is as follows:

```
type list_including ⟨n:int⟩ =
  LConsEq   || (::) of {x:int|x=n} × int list
| LConsNEq  || (::) of
                {x:int|x<>n} × list_including ⟨n⟩
```

This version of `list_including` has no constructors compatible to `Nil` because the empty list does not include `n`. By contrast, there are two constructors, `LConsEq` and `LConsNEq`, both compatible to `(::)`. The constructor `LConsEq` is used to construct lists where the head is equal to n, and `LConsNEq` to construct lists where the head is not equal to n but the tail list includes n. A cast to the new version of `list_including` works by choosing either `LConsEq` or `LConsNEq`, depending on the head of the input list:

$$
\begin{aligned}
\langle \mathsf{list\_including}\langle 0 \rangle \Leftarrow \mathsf{int\ list} \rangle^{\ell}\,[\,] & \longrightarrow^{*} \Uparrow \ell \\
\langle \mathsf{list\_including}\langle 0 \rangle \Leftarrow \mathsf{int\ list} \rangle^{\ell}\,(2 :: 0 :: 1 :: [\,]) & \longrightarrow^{*} \\
\mathsf{LConsNEq}\,(2, (\mathsf{LConsEq}\,(0, 1 :: [\,])))
\end{aligned}
$$

This cast does not have to traverse a given list when it succeeds (notice int list in the argument type of LConsEq and $1 :: [\,]$ in the second example above).

Although it is fairly clear how to choose an appropriate constructor in the example above, it may not be as easy in general. In the formal semantics we give in this chapter, we model these choices as oracles. In practice, a constructor choice function is specified along with a datatype definition either manually or often automatically—in fact, we will show that a constructor choice function can be systematically derived when a new datatype is generated from our translation. More interestingly, the asymptotic time complexity of the cast from a plain list to the generated datatype is no worse than the cast to the original refinement type. In this sense, the translation preserves efficiency of casts. This efficiency preservation lets us conjecture that, when a programmer rewrites a program with the refinement type to one with the generated datatype, the asymptotic time complexity of the latter program becomes no worse than the former. We discuss efficiency preservation in detail in Section 4.3.3.

Allowing nonbijective correspondence between constructors simplifies our translation and makes dynamic contract checking more efficient as in the example above.

4.1.2 Ideas for Translation

We informally describe the ideas behind our translation through the example of list_including above. We start with the refinement type $\{x{:}\mathsf{int\ list} \mid \mathsf{member}\ n\ x\}$, where `member n x` is a usual function, which returns whether `n` appears in list `x`:

```
let rec member (n:int) (l:int list) =
  match l with
  | []    -> false
  | x::xs ->
    if x = n then true
             else member n xs
```

Through this chapter, we always suppose that some logical operations such as `&&` and `||` are desugared to simplify our formalization, and so here we write `if x = n then true else member n xs` instead of `x = n || member n xs`. We examine how list_including corresponds to `member`. For reference, the definition of list_including is shown below again:

```
type list_including ⟨n:int⟩ =
| LConsEq  || (::) of {x:int|x=n} × int list
| LConsNEq || (::) of
                {x:int|x<>n} × list_including ⟨n⟩
```

We expect that a value of list_including $\langle n \rangle$ returns true when it is passed to `member n` (modulo constructor names).

It is not difficult to observe two things. First, each constructor and its argument type represent when the predicate returns true. In this example, there are two reasons that `member n x` returns true: either (1) `n` is equal to the first element of `x` or (2) `n` is not equal to the first element of `x` but `member n` is true for the tail of `x`. The constructors LConsEq and LConsNEq and their argument types represent these conditions. Since `member n x` never returns true when `x` is the empty list, there is no constructor in list_including. Second, a recursive call on a substructure corresponds to type-level recursion: `member n xs` in the `else`-branch in `member` is represented by list_including$\langle n \rangle$ in the argument type of LConsNEq.

Types

$$T \quad ::= \quad \mathsf{bool} \mid x{:}T_1 \to T_2 \mid x{:}T_1 \times T_2 \mid \{x{:}T \mid e\} \mid \tau\langle e\rangle$$

Constants, Values, Terms

$$\begin{array}{lll} c & ::= & \mathsf{true} \mid \mathsf{false} \\ v & ::= & c \mid \mathtt{fix}\, f(x{:}T_1){:}T_2 = e \mid \langle T_1 \Leftarrow T_2\rangle^{\ell} \mid (v_1, v_2) \mid C\langle e\rangle v \\ e & ::= & c \mid x \mid \mathtt{fix}\, f(x{:}T_1){:}T_2 = e \mid e_1\ e_2 \mid (e_1, e_2) \mid e.1 \mid e.2 \mid \\ & & C\langle e_1\rangle e_2 \mid \mathsf{match}\ e\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i} \mid \\ & & \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \langle T_1 \Leftarrow T_2\rangle^{\ell} \end{array}$$

Datatype definitions

$$\begin{array}{lll} \varsigma & ::= & \tau\,\langle x{:}T\rangle = \overline{C_i\ :\ T_i}^{\,i} \mid \tau\,\langle x{:}T\rangle = \overline{C_i \parallel D_i\ :\ T_i}^{\,i} \\ \Sigma & ::= & \emptyset \mid \Sigma, \varsigma \end{array}$$

FIGURE 4.1: Program syntax.

$\mathit{TypDefOf}_{\Sigma}(\tau)$	The definition of τ.
$\mathit{ArgTypeOf}_{\Sigma}(\tau)$	The parameter name and its type of τ.
$\mathit{CtrsOf}_{\Sigma}(\tau)$	The set of constructors that belong to τ.
$\mathit{TypSpecOf}_{\Sigma}(C)$	The type specification of C.
$\mathit{TypNameOf}_{\Sigma}(C)$	The datatype that C belongs to.
$\mathit{CtrArgOf}_{\Sigma}(C)$	The argument type of C.

TABLE 4.1: Lookup functions.

So, the basic idea of our translation scheme is to analyze the body of a given predicate function and collect guarding conditions on branches reaching `true`. As mentioned above, recursive calls on the tail become type-level recursion. This correspondence between execution paths and data constructors is also useful to derive a constructor choice function for a cast. For example, a cast to list_including$\langle n\rangle$ will choose LConsEq when (the list being checked is not empty and) the head is equal to n, just because LConsEq corresponds to the path guarded by `x=n` in the definition of `member`.

4.2 A Manifest Contract Calculus $\lambda^{H}_{\mathbf{dt}}$

We formalize a manifest contract calculus $\lambda^{H}_{\mathbf{dt}}$ of datatypes with its syntax, type system, and operational semantics, and prove its type soundness. Following Belo et al. [14], we drop subtyping and subsumption from the core of the calculus to simplify the definition and metatheory.

In the following, we write a sequence with an overline: for example, $\overline{C_i}^{\,i \in \{1,\ldots,n\}}$ means a sequence $C_1, \ldots, C_n$ of data constructors. We often omit the index set $\{1, ..., n\}$ when it is clear from the context or not important. Given a binary relation R, the relation R^* denotes the reflexive and transitive closure of R.

4.2.1 Syntax

We present the program syntax of $\lambda^{H}_{\mathbf{dt}}$ in Figure 4.1, where there are various metavariables: T ranges over types, τ names of datatypes, C and D constructors, c constants,

x, y, z, f, etc. variables, v values, e terms, ℓ blame labels, Γ typing contexts, ς datatype definitions, Σ type definition environments.

Types consist of base types (we have only Boolean here but addition of other base types causes no problems), dependent function types, dependent product types, refinement types, and datatypes. In a dependent function type $x{:}T_1 \to T_2$ and a dependent product type $x{:}T_1 \times T_2$, variable x is bound in T_2. A refinement type $\{x{:}T \mid e\}$, in which x is bound in e, denotes the subset of type T whose value v satisfies the Boolean contract e, that is, $e\,\{v/x\}$ evaluates to true. Finally, a datatype $\tau\langle e\rangle$ takes the form of an application of τ to a term e. Note that, similarly to $\mathrm{F}^{\sigma}_{\mathrm{H}}$, the predicate e is allowed to be an arbitrary Boolean expression, which may diverge or raise blame, unlike some refinement type systems [123, 122, 91, 59, 116, 117], which aim at decidable static verification. Static verification amounts to checking a given cast is in fact an upcast, where the source type is a subtype of the target, and subtyping is not, in general, decidable but the language is not equipped with subsumption.

Terms are basically those from the λ-calculus with Booleans, recursive functions, products, datatypes, and casts. A term $\mathtt{fix}\, f(x{:}T_1){:}T_2 = e$ represents a recursive function in which variables x and f denote an argument and the function itself, respectively, and are bound in e. We often omit type annotations. A data constructor application $C\langle e_1\rangle e_2$ takes two arguments: e_1 represents one for the type definition and e_2 for data constructors, respectively. A match expression $\mathsf{match}\, e\, \mathsf{with}\, \overline{C_i\, x_i \to e_i}^{\,i}$ is as usual and binds each variable x_i in e_i.

The last form is a cast $\langle T_1 \Leftarrow T_2\rangle^{\ell}$, consisting of a target type T_1, a source type T_2, and a label ℓ, and, when applied to a value v of type T_2, checks that the value v can behave as T_1. The label ℓ is used to identify the cast when it is blamed. Unlike $\mathrm{F}^{\sigma}_{\mathrm{H}}$, casts in $\lambda^{H}_{\mathbf{dt}}$ do not contain delayed substitutions.

A datatype definition ς can take two forms. The form $\tau\,\langle x{:}T\rangle = \overline{C_i\, :\, T_i}^{\,i}$, where x is bound in $\overline{T_i}^{\,i}$, declares a datatype τ, parameterized over x of type T, with data constructors C_i whose argument types are T_i. The other form $\tau\,\langle x{:}T\rangle = \overline{C_i \parallel D_i\, :\, T_i}^{\,i}$ is the same except that it declares that C_i is compatible with D_i from another datatype.

A type definition environment Σ is a sequence of datatype definitions. We assume that datatype and constructor names declared in a type definition environment are distinct. Table 4.1 shows metafunctions to look up information on datatype definitions. Their definitions are omitted since they are straightforward. A type specification, returned by *TypSpecOf* and written $x{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle x\rangle$, of a constructor C consists of the datatype τ that C belongs to, the parameter x of τ and the type T_1 of x, and the argument type T_2 of C. In other words, $\tau = \mathit{TypNameOf}_{\Sigma}(C)$, $x{:}T_1 = \mathit{ArgTypeOf}_{\Sigma}(\tau)$ and $T_2 = \mathit{CtrArgOf}_{\Sigma}(C)$. We omit the subscript Σ from these metafunctions for brevity if it is clear from the context.

We use the following familiar notations. We write $\mathsf{FV}\,(e)$ to denote the set of free variables in a term e, and $e\,\{e'/x\}$ capture avoiding substitution of e' for x in e. We apply similar notations to values and types. We say that a term/value/type is closed if it has no free variables, and identify α-equivalent ones. In addition, we introduce several shorthands. A function type $T_1 \to T_2$ means $x{:}T_1 \to T_2$ where the variable x does not occur free in T_2. We write $\lambda\, x{:}T_1.e$ to denote $\mathtt{fix}\, f(x{:}T_1){:}T_2 = e$ if f does not occur in the term e. A let-expression $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ denotes $(\lambda\, x{:}T.e_2)\, e_1$ where T is an appropriate type. Finally, a datatype τ is said to be *monomorphic* if the definition of τ does not refer to a type argument variable, and then we abbreviate $\tau\langle e\rangle$ to τ and $C\langle e_1\rangle e_2$ to $C\; e_2$ when C is a data constructor of τ.

$\boxed{\vdash \Gamma}$ **Typing Context Well-Formedness Rules**

$$\frac{}{\vdash \emptyset}\ \text{WC\_EMPTY} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash T}{\vdash \Gamma, x{:}T}\ \text{WC\_EXTENDVAR}$$

$\boxed{\Gamma \vdash T}$ **Type Well-Formedness Rules**

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{bool}}\ \text{WT\_BASE} \qquad \frac{\Gamma \vdash T_1 \quad \Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash x : T_1 \to T_2}\ \text{WT\_FUN}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma, x{:}T_1 \vdash T_2}{\Gamma \vdash x{:}T_1 \times T_2}\ \text{WT\_PROD} \qquad \frac{\Gamma \vdash T \quad \Gamma, x{:}T \vdash e : \mathsf{bool}}{\Gamma \vdash \{x{:}T \mid e\}}\ \text{WT\_REFINE}$$

$$\frac{\mathit{ArgTypeOf}(\tau) = x{:}T \quad \Gamma \vdash e : T}{\Gamma \vdash \tau\langle e\rangle}\ \text{WT\_DATATYPE}$$

$\boxed{\Gamma \vdash e : T}$ **Typing Rules**

$$\frac{\vdash \Gamma \quad c \in \{\mathsf{true}, \mathsf{false}\}}{\Gamma \vdash c : \mathsf{bool}}\ \text{T\_CONST} \qquad \frac{\vdash \Gamma \quad x{:}T \in \Gamma}{\Gamma \vdash x : T}\ \text{T\_VAR}$$

$$\frac{\Gamma, f{:}(x{:}T_1 \to T_2), x{:}T_1 \vdash e : T_2 \quad f \notin \mathsf{FV}(T_2)}{\Gamma \vdash \mathtt{fix}\, f(x{:}T_1){:}T_2 = e : x{:}T_1 \to T_2}\ \text{T\_ABS}$$

$$\frac{\Gamma \vdash T_1 \quad \Gamma \vdash T_2 \quad T_1 \parallel T_2}{\Gamma \vdash \langle T_1 \Leftarrow T_2\rangle^{\ell} : T_2 \to T_1}\ \text{T\_CAST} \qquad \frac{\Gamma \vdash e_1 : x{:}T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2\{e_2/x\}}\ \text{T\_APP}$$

$$\frac{\Gamma, x{:}T_1 \vdash T_2 \quad \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2\{e_1/x\}}{\Gamma \vdash (e_1, e_2) : x{:}T_1 \times T_2}\ \text{T\_PAIR}$$

$$\frac{\Gamma \vdash e : x{:}T_1 \times T_2}{\Gamma \vdash e.1 : T_1}\ \text{T\_PROJ1} \qquad \frac{\Gamma \vdash e : x{:}T_1 \times T_2}{\Gamma \vdash e.2 : T_2\{e.1/x\}}\ \text{T\_PROJ2}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}\ \text{T\_IF}$$

$$\frac{\mathit{TypSpecOf}(C) = x{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle x\rangle \quad \Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2\{e_1/x\} \quad \Gamma \vdash \tau\langle e_1\rangle}{\Gamma \vdash C\langle e_1\rangle e_2 : \tau\langle e_1\rangle}\ \text{T\_CTR}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : \tau\langle e\rangle \quad \Gamma \vdash T \\ \mathit{CtrsOf}(\tau) = \overline{C_i}^{\,i \in \{1,\ldots,n\}} \quad \mathit{ArgTypeOf}(\tau) = y{:}T' \\ \text{for all } i,\ \mathit{CtrArgOf}(C_i) = T_i \quad \text{for all } i,\ \Gamma, x_i{:}T_i\{e/y\} \vdash e_i : T\end{array}}{\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\, x_i \to e_i}^{\,i \in \{1,\ldots,n\}} : T}\ \text{T\_MATCH}$$

FIGURE 4.2: Typing rules for $\lambda^{H}_{\mathbf{dt}}$.

4.2.2 Type System

This section introduces a type system for source programs in $\lambda^{H}_{\mathbf{dt}}$; later we extend the syntax to include run-time terms to define operational semantics and give additional typing rules for those terms. The type system consists of three judgments: context well-formedness $\Sigma \vdash \Gamma$, type well-formedness $\Sigma; \Gamma \vdash T$, and typing $\Sigma; \Gamma \vdash e : T$. Here, a

$\boxed{T_1 \parallel T_2}$ **Type Compatibility**

$$\frac{T_1 \parallel T_2}{\{x{:}T_1 \mid e_1\} \parallel T_2}\ \text{C\_RefineL}$$

$$\frac{\begin{array}{c}\mathit{TypDefOf}(\tau_1) = (\mathsf{type}\,\tau_1\,\langle x{:}T\rangle = \overline{C_i \parallel D_i\ :\ T_i}^{\,i}) \\ \text{for all } i,\ \mathit{TypNameOf}(D_i) = \tau_2\end{array}}{\tau_1\langle e_1\rangle \parallel \tau_2\langle e_2\rangle}\ \text{C\_Datatype}$$

FIGURE 4.3: Type compatibility for $\lambda^H_{\mathbf{dt}}$.

typing context Γ is a sequence of variable declarations:

$$\Gamma ::= \emptyset \mid \Gamma, x{:}T$$

where declared variables are pairwise distinct. We show inference rules in Figure 4.2, where a type definition environment Σ in judgments are omitted for simplification. Typing rules for atomic terms, such as Booleans, variables, etc. demand that types of a typing context of a judgment be well-formed; in other rules, well-formedness of a typing context and a type of a term is shown as a derived property.

Inference rules for context and type well-formedness judgments are standard except for (WT_Datatype), which requires an argument to a datatype τ to be typed at the declared argument type.

Most of typing rules are also standard or similar to the previous work [14]. The rule (T_Cast) means that the source and target types in a cast have to be compatible. Intuitively, two types are compatible when a cast from one type to the other may succeed. More formally, type compatibility, written $T_1 \parallel T_2$, is the least congruence satisfying rules in Figure 4.3: the rule (C_RefineL) allows casts from and to refinement types; and the rule (C_Datatype) says that if datatypes are declared to be compatible in the type definition, then they are compatible. The typing rule (T_Ctr) demands that arguments e_2 and e_1 respect the argument types of C and the datatype that C belongs to, respectively. The rule (T_Match) for match expressions demands the matched term e_0 to be typed at a datatype $\tau\langle e\rangle$. Using the metafunction *CtrsOf*, the rule demands that the patterns $\overline{C_i\,x_i}^{\,i}$ be exhaustive. Moreover, each branch e_i has to be given the same type T, which cannot contain pattern variables x_i (and so is well formed under Γ).

4.2.3 Semantics

The semantics of $\lambda^H_{\mathbf{dt}}$ is given in the small-step style by using two relations over closed terms: the reduction relation ($\rightsquigarrow$), which represents basic computation such as β-reduction, and the evaluation relation ($\longrightarrow$), in which a subexpression is reduced.

The semantics is parameterized by a type definition environment and a *constructor choice function* δ, which is a partial function that maps a term of the form $\langle\tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^\ell\ C_2\langle e\rangle v$ to a constructor C_1. We introduce this function as an oracle to decide which constructor a given constructor is converted to by a cast between datatypes, as discussed in Section 4.1. The constructor C_1 has to not only belong to τ_1 but also be compatible with C_2. We will give a more precise condition on δ later.

Precisely speaking, the two relations are parameterized by Σ and δ but we fix certain Σ and δ in what follows and usually omit them from relations and judgments.

$\boxed{e_1 \rightsquigarrow e_2}$ **Reduction Rules**

$$(\texttt{fix}\, f(x{:}T_1){:}T_2 = e)\, v \rightsquigarrow e\,\{v/x, \texttt{fix}\, f(x{:}T_1){:}T_2 = e/f\} \quad \text{(R\_BETA)}$$

$$(v_1, v_2).1 \rightsquigarrow v_1 \quad \text{(R\_PROJ1)}$$

$$(v_1, v_2).2 \rightsquigarrow v_2 \quad \text{(R\_PROJ2)}$$

$$\mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow e_1 \quad \text{(R\_IFTRUE)}$$

$$\mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \rightsquigarrow e_2 \quad \text{(R\_IFFALSE)}$$

$$\mathsf{match}\ C_j\langle e\rangle v\ \mathsf{with}\ \overline{C_i\, x_i \to e_i}^{\,i} \rightsquigarrow e_j\,\{v/x_j\} \quad (\text{where } C_j \in \overline{C_i}^{\,i}) \quad \text{(R\_MATCH)}$$

$$\langle \mathsf{bool} \Leftarrow \mathsf{bool}\rangle^{\ell}\, v \rightsquigarrow v \quad \text{(R\_BASE)}$$

$$\langle x{:}T_{11} \to T_{12} \Leftarrow x{:}T_{21} \to T_{22}\rangle^{\ell}\, v \rightsquigarrow (\lambda\, x{:}T_{11}.\mathsf{let}\, y = \langle T_{21} \Leftarrow T_{11}\rangle^{\ell}\, x\ \mathsf{in}\ \langle T_{12} \Leftarrow (T_{22}\,\{y/x\})\rangle^{\ell}\,(v\ y)) \quad \text{(where } y \text{ is fresh)} \quad \text{(R\_FUN)}$$

$$\langle x{:}T_{11} \times T_{12} \Leftarrow x{:}T_{21} \times T_{22}\rangle^{\ell}\,(v_1, v_2) \rightsquigarrow \mathsf{let}\, x = \langle T_{11} \Leftarrow T_{21}\rangle^{\ell}\, v_1\ \mathsf{in}\ (x, \langle T_{12} \Leftarrow (T_{22}\,\{v_1/x\})\rangle^{\ell}\, v_2) \quad \text{(R\_PROD)}$$

$$\langle T_1 \Leftarrow \{x{:}T_2 \mid e\}\rangle^{\ell}\, v \rightsquigarrow \langle T_1 \Leftarrow T_2\rangle^{\ell}\, v \quad \text{(R\_FORGET)}$$

$$\langle \{x{:}T_1 \mid e\} \Leftarrow T_2\rangle^{\ell}\, v \rightsquigarrow \langle\!\langle \{x{:}T_1 \mid e\}, \langle T_1 \Leftarrow T_2\rangle^{\ell}\, v\rangle\!\rangle^{\ell} \quad \text{(R\_PRECHECK)}$$

(where T_2 is not a refinement type)

$$\langle \tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^{\ell}\, C_2\langle e\rangle v \rightsquigarrow C_1\langle e_1\rangle(\langle T_1'\,\{e_1/x_1\} \Leftarrow T_2'\,\{e_2/x_2\}\rangle^{\ell}\, v) \quad \text{(R\_DATATYPE)}$$

(where $\tau_1 \neq \tau_2$ or τ_1 is not monomorphic, and $\delta(\langle \tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^{\ell}\, C_2\langle e\rangle v) = C_1$ and $\mathit{ArgTypeOf}(\tau_i) = x_i{:}T_i$ and $\mathit{CtrArgOf}(C_i) = T_i'$ for $i \in \{1,2\}$)

$$\langle \tau \Leftarrow \tau\rangle^{\ell}\, v \rightsquigarrow v \quad \text{(R\_DATATYPEMONO)}$$

$$\langle \tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^{\ell}\, v \rightsquigarrow \Uparrow\ell \quad \text{(R\_DATATYPEFAIL)}$$

(where $\tau_1 \neq \tau_2$ or τ_1 is not monomorphic, and $\delta(\langle \tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^{\ell}\, v)$ is undefined)

$$\langle\!\langle \{x{:}T \mid e\}, v\rangle\!\rangle^{\ell} \rightsquigarrow \langle \{x{:}T \mid e\}, e\,\{v/x\}, v\rangle^{\ell} \quad \text{(R\_CHECK)}$$

$$\langle \{x{:}T \mid e\}, \mathsf{true}, v\rangle^{\ell} \rightsquigarrow v \quad \text{(R\_OK)}$$

$$\langle \{x{:}T \mid e\}, \mathsf{false}, v\rangle^{\ell} \rightsquigarrow \Uparrow\ell \quad \text{(R\_FAIL)}$$

$\boxed{e_1 \longrightarrow e_2}$ **Evaluation Rules**

$$\frac{e_1 \rightsquigarrow e_2}{E[e_1] \longrightarrow E[e_2]}\ \text{E\_RED} \qquad \frac{E \neq [\,]}{E[\Uparrow\ell] \longrightarrow \Uparrow\ell}\ \text{E\_BLAME}$$

FIGURE 4.4: Operational semantics for $\lambda^{H}_{\mathbf{dt}}$.

Before reduction and evaluation rules, we introduce several run-time terms to express dynamic contract checking in the semantics. These run-time terms are assumed not to appear in a source program (or datatype definitions). The syntax is extended as below:

$$e \quad ::= \quad \ldots \mid \Uparrow\ell \mid \langle \{x{:}T \mid e_1\}, e_2, v\rangle^{\ell} \mid \langle\!\langle \{x{:}T \mid e_1\}, e_2\rangle\!\rangle^{\ell}$$

The term $\Uparrow\ell$ denotes a cast failure blaming ℓ, which identifies which cast failed. An active check $\langle \{x{:}T \mid e_1\}, e_2, v\rangle^{\ell}$ verifies that the value v of type T satisfies the contract e_1. The term e_2 represents an intermediate state of a check, which starts by reducing $e_1\,\{v/x\}$. If the check succeeds, namely e_2 reduces to true, then the active check evaluates to v; otherwise, if e_2 reduces to false, then it is blamed with ℓ. A waiting check

$\langle\!\langle \{x{:}T \mid e_1\}, e_2 \rangle\!\rangle^{\ell}$, which appears when an application of a cast to a refinement type is reduced, checks that the value of e_2 satisfies e_1. Waiting checks are introduced to determine how casts behave regardless of what values are substituted. We will discuss it in more detail at the end of this section.

Figure 4.4 shows reduction and evaluation rules. Reduction rules are standard except for those about casts and active/waiting checks. There are six reduction rules for casts. The rule (R_BASE) means that a cast between the same base type simply behaves like an identity function. The rule (R_FUN), which shows that casts between function types behave like function contracts [17, 44], produces a lambda abstraction which wraps the value v with the contravariant cast $\langle T_{21} \Leftarrow T_{11} \rangle^{\ell}$ between the argument types and the covariant cast $\langle T_{12} \Leftarrow T_{22} \rangle^{\ell}$ between the return types. To avoid capture of the bound variable of T_{12}, we take a fresh variable y and rename variable x in T_{22} to it. Similar renaming is performed in (R_PROD). The rule (R_PROD) means that elements v_1 and v_2 are checked by covariant casts obtained by decomposing the source and target types. The rules (R_FORGET) and (R_PRECHECK) are applied when source and target types of a cast are refinement types, respectively: the rule (R_FORGET) peels the outermost refinement of the source type; and the rule (R_PRECHECK) means that inner refinements in the target type are first checked and then the outermost one is. The side condition in (R_PRECHECK) are needed to make the semantics deterministic. For example, the term $\langle \{x{:}\mathsf{int} \mid 0 < x + 1\} \Leftarrow \{x{:}\mathsf{int} \mid 0 < x\} \rangle^{\ell}\, v$ reduces to $\langle\!\langle \{x{:}\mathsf{int} \mid 0 < x + 1\}, \langle \mathsf{int} \Leftarrow \mathsf{int} \rangle^{\ell}\, v \rangle\!\rangle^{\ell}$ by applying first (R_FORGET) and then (R_PRECHECK). A waiting check turns into an active check when its second argument becomes a value (by (R_CHECK)).

There are three rules (R_DATATYPE), (R_DATATYPEMONO), and (R_DATATYPEFAIL) for datatype casts. The rule (R_DATATYPE) is applied when the choice function δ gives the constructor C_1; then the original constructor argument v is passed to a cast between the argument types of C_2 and C_1. Here, note that e_1 and e_2 are substituted for variables x_1 and x_2 in the argument types of C_1 and C_2, respectively, because these types depend on these variables. The rule (R_DATATYPEMONO) is similar to (R_BASE). The rule (R_DATATYPEFAIL) says that, if the choice function δ is undefined for the cast, the cast application is blamed with ℓ.

The last three rules (R_CHECK), (R_OK), and (R_FAIL) follow the intuitive meaning of active checks explained above.

Evaluation rules are also shown in Figure 4.4. Here, evaluation contexts [33], ranged over by E, are defined as usual:

$$
\begin{aligned}
E ::= {} & [\,] \mid E\ e_2 \mid v_1\ E \mid (E, e_2) \mid (v_1, E) \mid E.1 \mid E.2 \mid C\langle e_1 \rangle E \mid \\
& \mathsf{match}\ E\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i} \mid \mathsf{if}\ E\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \\
& \langle \{x{:}T \mid e\}, E, v \rangle^{\ell} \mid \langle\!\langle \{x{:}T \mid e\}, E \rangle\!\rangle^{\ell}
\end{aligned}
$$

The rule (E_RED) means that evaluation proceeds by reducing the redex indicated by an evaluation context; the rule (E_BLAME) means that a raised blame will abort program execution.

We present a few examples of cast reduction. In the examples, we use int list, which is declared as follows (we assume that int and unit are provided as base types):

$$\mathsf{int\ list}\ \langle n{:}\mathsf{unit} \rangle = [\,] : \mathsf{unit} \mid (::) : \mathsf{int} \times \mathsf{int\ list}$$

We suppose that we have function length, which takes values of int list and returns their length. The first example is refinement checking, which is performed through waiting

checks:

$$
\begin{array}{lll}
 & \langle \{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x > 0\} \Leftarrow \mathsf{int\ list} \rangle^{\ell}\ (1 :: [\,]) & \\
\longrightarrow & \langle\!\langle \{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x > 0\}, \langle \mathsf{int\ list} \Leftarrow \mathsf{int\ list} \rangle^{\ell}\ (1 :: [\,]) \rangle\!\rangle^{\ell} & \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_PRECHECK))} & \\
\longrightarrow & \langle\!\langle \{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x > 0\}, 1 :: [\,] \rangle\!\rangle^{\ell} & \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_DATATYPEMONO))} & \\
\longrightarrow & \langle \{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x > 0\}, \mathsf{length}\ (1 :: [\,]) > 0, 1 :: [\,] \rangle^{\ell} & \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_CHECK))} & \\
\longrightarrow^{*} & \langle \{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x > 0\}, \mathsf{true}, 1 :: [\,] \rangle^{\ell} & \\
\longrightarrow & 1 :: [\,] \qquad\qquad \text{(by (E\_RED)/(R\_OK))} &
\end{array}
$$

Note that int list is monomorphic. Unlike $\mathrm{F}^{\sigma}_{\mathrm{H}}$, $\lambda^{H}_{\mathbf{dt}}$ does not skip checks of "trivial" refinements. The cast semantics of $\lambda^{H}_{\mathbf{dt}}$ checks all refinements in the target type of a cast, even if it is reflexive, after forgetting refinements in the source type (in $\mathrm{F}^{\sigma}_{\mathrm{H}}$ reflexive casts return target values immediately by (E_REFL)):

$$
\begin{array}{lll}
 & \langle \{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x > 0\} \Leftarrow \{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x > 0\} \rangle^{\ell}\ (1 :: [\,]) & \\
\longrightarrow & \langle \{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x > 0\} \Leftarrow \mathsf{int\ list} \rangle^{\ell}\ (1 :: [\,]) & \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_FORGET))} & \\
\longrightarrow^{*} & \langle \{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x > 0\}, \mathsf{length}\ (1 :: [\,]) > 0, 1 :: [\,] \rangle^{\ell} & \\
\longrightarrow^{*} & 1 :: [\,] &
\end{array}
$$

The cast semantics of $\lambda^{H}_{\mathbf{dt}}$ determines which reduction rule is chosen by examining "kinds" of types on casts, namely, whether the types on a cast are refinement types, dependent function types, dependent product types, or datatypes. Such kinds are not changed by value substitutions, so the cast semantics of $\lambda^{H}_{\mathbf{dt}}$ is insensitive to substitutions. Second is reduction of casts between dependent product types. Such casts check the left component of a given product and then the right one because the type of the right may depend on the left:

$$
\begin{array}{lll}
 & \langle x{:}\mathsf{int\ list} \times \{y{:}\mathsf{int\ list} \mid \mathsf{length}\ y > \mathsf{length}\ x\} \Leftarrow \mathsf{int\ list} \times \mathsf{int\ list} \rangle^{\ell}\ ([\,], 1 :: [\,]) & \\
\longrightarrow & \mathsf{let}\ x = \langle \mathsf{int\ list} \Leftarrow \mathsf{int\ list} \rangle^{\ell}\ [\,]\ \mathsf{in} & \\
 & \quad (x, \langle \{y{:}\mathsf{int\ list} \mid \mathsf{length}\ y > \mathsf{length}\ x\} \Leftarrow \mathsf{int\ list} \rangle^{\ell}\ 1 :: [\,]) & \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_PROD))} & \\
\longrightarrow^{*} & ([\,], \langle \{y{:}\mathsf{int\ list} \mid \mathsf{length}\ y > \mathsf{length}\ [\,]\} \Leftarrow \mathsf{int\ list} \rangle^{\ell}\ 1 :: [\,]) & \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_DATATYPEMONO))} & \\
\longrightarrow^{*} & ([\,], \langle \{y{:}\mathsf{int\ list} \mid \mathsf{length}\ y > \mathsf{length}\ [\,]\}, \mathsf{length}\ (1 :: [\,]) > \mathsf{length}\ [\,], 1 :: [\,] \rangle^{\ell}) & \\
\longrightarrow^{*} & ([\,], 1 :: [\,]) &
\end{array}
$$

Finally, as a reduction example of casts for (nonmonomorphic) datatypes, consider a cast from int list to list_including. The datatype list_including is presented in Section 4.1.1 and can be formalized as follows:

$$
\begin{array}{l}
\mathsf{list\_including}\ \langle n{:}\mathsf{int} \rangle\ = \\
\quad \mathsf{LConsEq} \parallel (::)\ :\ \{x{:}\mathsf{int} \mid x = n\} \times \mathsf{int\ list} \\
\mid\ \ \mathsf{LConsNEq} \parallel (::)\ :\ \{x{:}\mathsf{int} \mid x \neq n\} \times \mathsf{list\_including} \langle n \rangle
\end{array}
$$

Let us reduce cast application $\langle \mathsf{list\_including} \langle 2 \rangle \Leftarrow \mathsf{int\ list} \rangle^{\ell}\ (1 :: 2 :: [\,])$. Evaluation of casts for datatypes rests on constructor choice functions. If the domain of a constructor choice function δ does not contain $\langle \mathsf{list\_including} \langle 2 \rangle \Leftarrow \mathsf{int\ list} \rangle^{\ell}\ (1 :: 2 :: [\,])$, the evaluation of the cast application under δ results in blame with label ℓ (by (R_DATATYPEFAIL)).

Otherwise, since list_including is not monomorphic, the reduction rule (R_DATATYPE) is applied. If δ returns LConsEq, then the evaluation proceeds as follows:

$$
\begin{array}{ll}
 & \langle \mathsf{list\_including}\langle 2\rangle \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,(1 :: 2 :: [\,]) \\
\longrightarrow & \mathsf{LConsEq}\langle 2\rangle((\{x{:}\mathsf{int} \mid x = 2\} \times \mathsf{int\ list} \Leftarrow \mathsf{int} \times \mathsf{int\ list}\rangle^{\ell}\,(1, 2 :: [\,])) \\
\longrightarrow & \mathsf{LConsEq}\langle 2\rangle \\
 & \quad (\mathsf{let}\ y = \langle \{x{:}\mathsf{int} \mid x = 2\} \Leftarrow \mathsf{int}\rangle^{\ell}\,1\ \mathsf{in}\ (y, \langle \mathsf{int\ list} \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,2 :: [\,])) \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_PROD))} \\
\longrightarrow^{*} & \mathsf{LConsEq}\langle 2\rangle \\
 & \quad (\mathsf{let}\ y = \langle \{x{:}\mathsf{int} \mid x = 2\}, 1 = 2, 1\rangle^{\ell}\ \mathsf{in}\ (y, \langle \mathsf{int\ list} \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,2 :: [\,])) \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_PRECHECK), (R\_BASE), and (R\_CHECK))} \\
\longrightarrow & \mathsf{LConsEq}\langle 2\rangle \\
 & \quad (\mathsf{let}\ y = \langle \{x{:}\mathsf{int} \mid x = 2\}, \mathsf{false}, 1\rangle^{\ell}\ \mathsf{in}\ (y, \langle \mathsf{int\ list} \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,2 :: [\,])) \\
\longrightarrow & \mathsf{LConsEq}\langle 2\rangle(\mathsf{let}\ y = \Uparrow \ell\ \mathsf{in}\ (y, \langle \mathsf{int\ list} \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,2 :: [\,])) \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_FAIL))} \\
\longrightarrow & \Uparrow \ell \qquad\qquad \text{(by (E\_BLAME))}
\end{array}
$$

Otherwise, if δ returns LConsNEq, then:

$$
\begin{array}{ll}
 & \langle \mathsf{list\_including}\langle 2\rangle \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,(1 :: 2 :: [\,]) \\
\longrightarrow & \mathsf{LConsNEq}\langle 2\rangle((\{x{:}\mathsf{int} \mid x \neq 2\} \times \mathsf{list\_including}\langle 2\rangle \Leftarrow \mathsf{int} \times \mathsf{int\ list}\rangle^{\ell}\,(1, 2 :: [\,])) \\
\longrightarrow & \mathsf{LConsNEq}\langle 2\rangle \\
 & \quad (\mathsf{let}\ y = \langle \{x{:}\mathsf{int} \mid x \neq 2\} \Leftarrow \mathsf{int}\rangle^{\ell}\,1\ \mathsf{in}\ (y, \langle \mathsf{list\_including}\langle 2\rangle \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,2 :: [\,])) \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_PROD))} \\
\longrightarrow^{*} & \mathsf{LConsNEq}\langle 2\rangle \\
 & \quad (\mathsf{let}\ y = \langle \{x{:}\mathsf{int} \mid x \neq 2\}, 1 \neq 2, 1\rangle^{\ell}\ \mathsf{in}\ (y, \langle \mathsf{list\_including}\langle 2\rangle \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,2 :: [\,])) \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_PRECHECK), (R\_BASE), and (R\_CHECK))} \\
\longrightarrow & \mathsf{LConsNEq}\langle 2\rangle \\
 & \quad (\mathsf{let}\ y = \langle \{x{:}\mathsf{int} \mid x = 2\}, \mathsf{true}, 1\rangle^{\ell}\ \mathsf{in}\ (y, \langle \mathsf{list\_including}\langle 2\rangle \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,2 :: [\,])) \\
\longrightarrow^{*} & \mathsf{LConsNEq}\langle 2\rangle(1, \langle \mathsf{list\_including}\langle 2\rangle \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,2 :: [\,]) \\
 & \qquad\qquad \text{(by (E\_RED)/(R\_OK))}
\end{array}
$$

If $\delta(\langle \mathsf{list\_including}\langle 2\rangle \Leftarrow \mathsf{int\ list}\rangle^{\ell}\,2 :: [\,]) = \mathsf{LConsEq}$, then the cast application results in $\mathsf{LConsNEq}\,(1, \mathsf{LConsEq}\,(2, [\,]))$; if not so, it reduces to blame $\Uparrow \ell$. As we can see, evaluation results of cast applications for nonmonomorphic datatypes rest on constructor choice functions being considered.

4.2.4 Type Soundness

We show type soundness of $\lambda^{H}_{\mathbf{dt}}$. As usual, type soundness means that a well-typed term does not go wrong and is proved via progress and preservation [121, 86]. Moreover, we will show that, if a term is given a refinement type, its value (if it exists) satisfies the contract.

Before stating the type soundness theorem, we start with extending the type system to run-time terms, and define well-formedness of type definition environments and constructor choice functions.

Typing for Run-time Terms

Typing rules for run-time terms are shown in Figure 4.5. Most of them are the same as $\mathrm{F}^{\sigma}_{\mathrm{H}}$. The rule (T_BLAME) means that a blame $\Uparrow \ell$ can have any type because it is

$\boxed{\Gamma \vdash e : T}$

$$\frac{\vdash \Gamma \quad \emptyset \vdash T}{\Gamma \vdash \Uparrow \ell : T}\ \text{T\_BLAME} \qquad \frac{\vdash \Gamma \quad \emptyset \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash v : T \quad \emptyset \vdash e_2 : \mathsf{bool} \quad e_1\{v/x\} \longrightarrow^* e_2}{\Gamma \vdash \langle \{x{:}T \mid e_1\}, e_2, v\rangle^\ell : \{x{:}T \mid e_1\}}\ \text{T\_ACHECK}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash \{x{:}T \mid e_1\} \quad \emptyset \vdash e_2 : T}{\Gamma \vdash \langle\!\langle \{x{:}T \mid e_1\}, e_2 \rangle\!\rangle^\ell : \{x{:}T \mid e_1\}}\ \text{T\_WCHECK}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash e : T_1 \quad T_1 \equiv T_2 \quad \emptyset \vdash T_2}{\Gamma \vdash e : T_2}\ \text{T\_CONV}$$

$$\frac{\vdash \Gamma \quad \emptyset \vdash v : \{x{:}T \mid e\}}{\Gamma \vdash v : T}\ \text{T\_FORGET} \qquad \frac{\vdash \Gamma \quad \emptyset \vdash \{x{:}T \mid e\} \quad \emptyset \vdash v : T \quad e\{v/x\} \longrightarrow^* \mathsf{true}}{\Gamma \vdash v : \{x{:}T \mid e\}}\ \text{T\_EXACT}$$

FIGURE 4.5: Typing rules for run-time terms.

an exception. In the rule (T_ACHECK) for an active check $\langle \{x{:}T \mid e_1\}, e_2, v\rangle^\ell$, the last premise $e_1\{v/x\} \longrightarrow^* e_2$ means that e_2 is an intermediate state of checking, which started from $e_1\{v_1/x\}$. The rule (T_WCHECK) for a waiting check is easy to understand. The rule (T_FORGET) is needed because (R_FORGET) peels off the refinements in the source type of a cast. The rule (T_EXACT) allows a value which succeeds in dynamic checking to be typed at a refinement type. The rule (T_CONV) allows run-time terms to be retyped at convertible types; see the discussion in Sections 3.1.2 and 3.2.3 for details. Convertibility is formalized by the type equivalence (denoted by $\equiv$) given as follows.

Definition 3 (Type Equivalence)**.**

1. *The* common subexpression reduction relation $\Rrightarrow$ *over types is defined as follows:* $T_1 \Rrightarrow T_2$ *iff there exist some* T, x, e_1 *and* e_2 *such that* $T_1 = T\{e_1/x\}$ *and* $T_2 = T\{e_2/x\}$ *and* $e_1 \longrightarrow e_2$.

2. *The* type equivalence $\equiv$ *is the symmetric and transitive closure of* $\Rrightarrow$.

The type equivalence given here takes a different form from type convertibility in $\mathrm{F}^\sigma_\mathrm{H}$, but they denote the same notion; we think that this formalization is slightly simpler than that of $\mathrm{F}^\sigma_\mathrm{H}$.

The fact that typing contexts in premises are empty reflects that run-time terms are closed; however, they can appear under binders as part of types (notice term substitution in the typing rules) and so weakening is needed.

Well-formed Type Definition Environments

Intuitively, a type definition environment is well formed when the parameter type is well formed, constructor argument types are well formed, and the argument types of compatible constructors are compatible.

Definition 4 (Well-Formed Type Definition Environments)**.**

1. *Let* $\varsigma = \tau\,\langle x{:}T\rangle = \overline{C_i : T_i}^{\,i \in \{1,\ldots,n\}}$. *A type definition* ς *is* well formed under a type definition environment Σ *if it satisfies the followings: (a)* $0 < n$. *(b)* $\Sigma; \emptyset \vdash T$ *holds. (c) For any* $i \in \{1, \ldots, n\}$, $\Sigma, \varsigma; x{:}T \vdash T_i$ *holds.*

2. *Let* $\varsigma = \tau\langle x{:}T\rangle = \overline{C_i \parallel D_i : T_i}^{\,i\in\{1,\ldots,n\}}$. *A type definition* ς *is* well formed under a type definition environment Σ *if it satisfies the followings: (a)* $0 < n$. *(b)* $\Sigma;\emptyset \vdash T$ *holds. (c) For any* $i \in \{1,...,n\}$, $\Sigma, \varsigma; x{:}T \vdash T_i$ *holds. (d) There exists some datatype* τ' *in* Σ *such that constructors* $\overline{D_i}^{\,i\in\{1,\ldots,n\}}$ *belong to it. (e) For any* $i \in \{1,...,n\}$, T_i *is compatible with the argument type of* D_i *under* Σ, ς, *that is,* $\Sigma, \varsigma \vdash T_i \parallel \mathit{CtrArgOf}_\Sigma(D_i)$ *holds.*

3. *A type definition environment* Σ *is* well formed *if for any* Σ_1, ς, *and* Σ_2, $\Sigma = \Sigma_1, \varsigma, \Sigma_2$ *implies that* ς *is well formed under* Σ_1. *We write* $\vdash \Sigma$ *to denote that* Σ *is well formed.*

Intuitively, a constructor choice function is well formed when it returns a constructor related by $\parallel$ in type definitions and respects term equivalence, which is defined similarly to type equivalence.

Definition 5 (Compatible Constructors)**.** *The compatibility relation* $\parallel$ *over constructors is the least equivalence relation satisfying the following rule.*

$$\frac{\mathit{TypNameOf}(C_i) = \tau \qquad \mathit{TypDefOf}(\tau) = \mathsf{type}\,\tau\langle y{:}T\rangle = \overline{C_j \parallel D_j : T_j}^{\,j}}{C_i \parallel D_i}$$

The function CompatCtrsOf, which maps a datatype τ *and a constructor* C *to the set of compatible constructors of* τ, *is defined as follows:*

$$\mathit{CompatCtrsOf}(\tau, C) = \{D \mid C \parallel D \text{ and } \mathit{TypNameOf}(D) = \tau\}.$$

Definition 6 (Term Equivalence)**.**

1. *The* common subexpression reduction relation $\Rrightarrow$ *over terms is defined as follows:* $e_1 \Rrightarrow e_2$ *iff there exist some* e, x, e_1' *and* e_2' *such that* $e_1 = e\{e_1'/x\}$ *and* $e_2 = e\{e_2'/x\}$ *and* $e_1' \longrightarrow e_2'$.

2. *The* term equivalence $\equiv$ *is the symmetric and transitive closure of* $\Rrightarrow$.

Definition 7 (Well-Formed Constructor Choice Functions)**.** *A constructor choice function* δ *is* well formed *iff*

1. *if* $C_1 = \delta(\langle\tau_1\langle e_1\rangle \Leftarrow \tau_2\langle e_2\rangle\rangle^\ell\ C_2\langle e\rangle v)$, *then* $C_1 \in \mathit{CompatCtrsOf}(\tau_1, C_2)$; *and*

2. *for any* e_1, e_2, *and* C, *if* $e_1 \equiv e_2$ *and* $\delta(e_1) = C$, *then* $\delta(e_2) = C$.

We suppose that the type definition environment and the choice function are well formed in what follows.

Lemma 27 (Progress)**.** *If* $\emptyset \vdash e : T$, *then*

1. $e \longrightarrow e'$ *for some* e',

2. e *is a value, or*

3. $e = \Uparrow\ell$ *for some* ℓ.

Lemma 28 (Preservation)**.** *Suppose that* $\emptyset \vdash e : T$.

(1) If $e \rightsquigarrow e'$, *then* $\emptyset \vdash e' : T$.

(2) If $e \longrightarrow e'$, *then* $\emptyset \vdash e' : T$.

To show the additional property mentioned above about refinement types, we need to show cotermination; the proof is similar to the one (Lemma 11) for $\mathrm{F}^{\sigma}_{\mathrm{H}}$.

Lemma 29 (Cotermination). *Suppose that* $e_1 \Rrightarrow^* e_2$.

(1) If $e_1 \longrightarrow^* \mathsf{true}$, *then* $e_2 \longrightarrow^* \mathsf{true}$.

(2) If $e_2 \longrightarrow^* \mathsf{true}$, *then* $e_1 \longrightarrow^* \mathsf{true}$.

Theorem 9 (Type Soundness). *If* $\emptyset \vdash e : T$, *then*

1. $e \longrightarrow^* v$ *for some* v *such that* $\emptyset \vdash v : T$;
2. $e \longrightarrow^* \Uparrow \ell$ *for some* ℓ; *or*
3. *there is an infinite sequence of evaluation* $e \longrightarrow e_1 \longrightarrow \cdots$.

Moreover, if T *is a refinement type* $\{x{:}T_0 \mid e_0\}$ *and the first case holds, then* $e_0\{v/x\} \longrightarrow^* \mathsf{true}$.

Proof. (1)–(3) follow from progress and preservation. For the additional property, it suffices to show that if $\emptyset \vdash v : T$, then v satisfies all contracts of type T. We proceed by induction on the derivation of $\emptyset \vdash v : T$. In the case of (T_CONV), we use Lemma 29 and the fact that for any v', if $v' \Rrightarrow^* \mathsf{true}$ or $\mathsf{true} \Rrightarrow^* v'$, then $v' = \mathsf{true}$. □

4.2.5 Comparison of $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and $\lambda^{H}_{\mathbf{dt}}$

Both cast mechanisms of $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and $\lambda^{H}_{\mathbf{dt}}$ are designed to prevent substitutions from affecting cast reduction for avoiding flaws in F_{H} [14, 43] (see Section 3.1.3). $\mathrm{F}^{\sigma}_{\mathrm{H}}$ avoids such flaws with the help of delayed substitutions. Support for delayed substitutions, however, would make the metatheory of a manifest contract calculus complicated—for example, the definition of substitution in $\mathrm{F}^{\sigma}_{\mathrm{H}}$ is unusual in the case for casts and its cast semantics have apparently peculiar side conditions to remove garbage bindings. $\lambda^{H}_{\mathbf{dt}}$ does not need delayed substitution and instead its cast semantics determines which cast reduction rule is chosen by examining kinds of types on casts, unlike $\mathrm{F}^{\sigma}_{\mathrm{H}}$, where choice of cast reduction rules depends on (in)equalities of types on casts. Since such kinds are not changed by term substitutions, we can see how casts behave statically; *all* refinements in the target type of a cast will be checked at run time after forgetting refinements in the source type. Waiting checks are introduced to record what refinements will be checked after forgetting refinements in the source type. The price we pay for removing type (in)equalities from the cast semantics is that we have to *prove* that reflexive casts can be eliminated. Support for elimination of reflexive casts has been important to show parametricity in $\mathrm{F}^{\sigma}_{\mathrm{H}}$, so it is not clear whether we can give a sound polymorphic manifest contract calculus without delayed substitutions. Another difference between $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and $\lambda^{H}_{\mathbf{dt}}$ appears in formalization of type convertibility, but the type conversion relations in $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and $\lambda^{H}_{\mathbf{dt}}$ denote the same notion.

4.3 Translation from Refinement Types to Datatypes

We give a translation from refinement types to datatypes and prove that the datatype obtained by the translation has the same meaning as the refinement type in the sense

Trans
input:
$\texttt{fix}\, f(y{:}T, x{:}\mathsf{int\ list}) = \mathsf{match}\ x\ \mathsf{with}\ [\,] \to e_1 \mid z_1 :: z_2 \to e_2$
returns:
1 **let** τ be a fresh type name **in**
2 **let** $\{T_i\}_i =$
$$\left\{ z_1{:}\mathsf{int} \times \{z_2{:}T_0 \mid e_0\} \;\middle|\; \begin{array}{l}(e_{\mathbf{opt}}, e) \in \mathit{GenContracts}\,(e_2),\\ (T_0, e_0) = \mathit{Aux}(\tau, e_{\mathbf{opt}}, e)\end{array} \right\} \textbf{ in}$$
3 **let** D and $\overline{D_i}^{\,i}$ be fresh constructor names, and
z be a fresh variable **in**
4 $\mathsf{type}\,\tau\,\langle y{:}T\rangle = D \parallel [\,] \,:\, \{z{:}\mathsf{unit} \mid e_1\} \mid \overline{D_i \parallel (::) \,:\, T_i}^{\,i}$

where
$\mathit{Aux}(\tau, e_{\mathbf{opt}}, e) =$
let $e' = e\,\{\texttt{fix}\, f(y{:}T, x{:}\mathsf{int\ list}) = .../f\}$ **in**
match $e_{\mathbf{opt}}$ **with**
$\mid \mathit{Some}\, e'' \to (\tau\langle e''\rangle, \mathsf{let}\ z_2 = \langle \mathsf{int\ list} \Leftarrow \tau\langle e''\rangle\rangle^{\ell}\ z_2\ \mathsf{in}\ e')$
$\mid \mathit{None} \to (\mathsf{int\ list}, e')$

FIGURE 4.6: Translation.

$$\begin{array}{lll}
\mathit{GenContracts}\,(\mathsf{true}) & = & \{(\mathit{None}, \mathsf{true})\} \\
\mathit{GenContracts}\,(\mathsf{false}) & = & \emptyset \\
\mathit{GenContracts}\,(\mathsf{if}\ f\ e_1\ z_2\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & = & \\
\multicolumn{3}{r}{\{(\mathit{Some}\ e_1, e_2)\} \cup \{(e_{\mathbf{opt}}, \mathsf{if}\ f\ e_1\ z_2\ \mathsf{then\ false\ else}\ e_3') \mid (e_{\mathbf{opt}}, e_3') \in \mathit{GenContracts}\,(e_3)\}} \\
\multicolumn{3}{r}{(\mathsf{if}\ \mathsf{FV}\,(e_1) \subseteq \{y, z_1\})} \\
\mathit{GenContracts}\,(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3) & = & \\
\multicolumn{3}{r}{\{(e_{\mathbf{opt}}, \mathsf{if}\ e_1\ \mathsf{then}\ e_2'\ \mathsf{else\ false}) \mid (e_{\mathbf{opt}}, e_2') \in \mathit{GenContracts}\,(e_2)\} \cup} \\
\multicolumn{3}{r}{\{(e_{\mathbf{opt}}, \mathsf{if}\ e_1\ \mathsf{then\ false\ else}\ e_3') \mid (e_{\mathbf{opt}}, e_3') \in \mathit{GenContracts}\,(e_3)\}} \\
\multicolumn{3}{r}{(\text{if a term of the form } f\ e\ z_2 \text{ occurs in } e_2 \text{ or } e_3)} \\
\mathit{GenContracts}\,(\mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i \in \{1,\ldots,n\}}) = & & \\
\multicolumn{3}{r}{\bigcup_{j \in \{1,\ldots,n\}} \{(e_{\mathbf{opt}}, \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i'}^{\,i \in \{1,\ldots,n\}}) \mid} \\
\multicolumn{3}{r}{(e_{\mathbf{opt}}, e_j') \in \mathit{GenContracts}\,(e_j) \wedge \forall i \neq j\,.\ e_i' = \mathsf{false}\}} \\
\multicolumn{3}{r}{(\text{if a term of the form } f\ e\ z_2 \text{ occurs in some } e_i)} \\
\mathit{GenContracts}\,(e) & = & \{(\mathit{None}, e)\} \qquad (\text{otherwise})
\end{array}$$

FIGURE 4.7: Generation of base contracts and arguments to recursive calls.

that a cast from the refinement type to the datatype always succeeds, and vice versa. We formalize our translation and prove its correctness using integer lists for simplicity and conciseness but our translation scheme can be generalized to other datatypes. We will informally discuss a more general case of binary trees later.

In this section, we assume that we have unit and int as base types and int list with $[\,]$ and infix cons $x :: y$ as constructors (the formal definition is given in Section 4.2). For simplicity, we also assume that the input predicate function is well typed and of the

following form:

$$\texttt{fix}\, f\,(y{:}T, x{:}\mathsf{int\ list}) = \mathsf{match}\ x\ \mathsf{with}\ [\,] \to e_1 \mid z_1 :: z_2 \to e_2$$

where $x \notin \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2)$. We will use `sorted` as a running example. For expository reasons, the definition is slightly verbose; the nested `if` expression at the end is essentially `z1 <= y and sorted () z2`.

```
let rec sorted (y:unit, x:int list) =
  match x with
  | []    -> true
  | z1::z2 -> e2^sorted
```

where

```
e2^sorted =
    match z2 with
    | []    -> true
    | y::ys -> if z1 <= y then
                  if sorted () z2 then true else false
               else false
```

4.3.1 Translation, Formally

We show the translation function *Trans* in Figure 4.6 and the auxiliary function *GenContracts* in Figure 4.7. The main function *Trans* takes a recursive function as an input and returns a corresponding datatype definition (on line 4).

On line 2, information on how e_2, which is the contract for $(::)$, can evaluate to true is gathered by the auxiliary function *GenContracts*. In the definition, variables f, y, z_1, and z_2 come from the input function and are fixed names. This function takes an expression as an input and returns a set of pairs $(e_{\mathbf{opt}}, e_2')$, each of which expresses one execution path that returns true in e_2. e_2' is derived from e_2 by substituting false for all but one path and $e_{\mathbf{opt}}$ is the first argument to a recursive call (if any) on this path. Intuitively, conjunction of e_2' and $f\ e_{\mathbf{opt}}\ z_2$ gives one sufficient condition for e_2 to be true and disjunction of the pairs in the returned set is logically equivalent to e_2. For example, *GenContracts*($e_2{}^{\texttt{sorted}}$) returns a set consisting of (None, e_{21}) where e_{21} is

```
match z2 with [] -> true | y::ys -> false
```

and $(\mathit{Some}\,(), e_{22})$ where e_{22} is

```
match z2 with
  []    -> false
| y::ys -> if z1 <= y then true else false .
```

(Gray bits show differences from $e_2{}^{\texttt{sorted}}$.) The first expression means that a (non-empty) list `x` is sorted when the tail is empty; and the second means that `x` is sorted when the head `z1` is equal to or smaller than the second element `y` and the recursive call `sorted () z2` returns true. *GenContracts* performs a kind of disjunctive normal form translation and each disjunct will correspond to a data constructor in the generated datatype.

Now let us take a look at the definition of *GenContracts*. The first two clauses are trivial: if the expression is true, then it returns the trivial contract and if it is false, then this branch should not be taken and hence the empty sequence is returned. The third clause deals with a conditional on a recursive call $f\ e_1\ z_2$ on the tail. In this case, it returns *Some* e_1, to signal there is a designated recursive call in this branch, with the

additional condition e_2 and also the condition when the recursive call returns false but e_3 is true. The following two clauses are for the other cases where the input expression is case analysis. In this case, from each branch, *GenContracts* recursively collects execution paths and reconstruct conditional expressions by replacing other branches with false. The side conditions on these clauses mean that we can stop DNF translation if there is no recursive calls on the tail and simply return the given contract as it is, by calling for the last clause, which deals with other forms of expressions.

The collected execution path information is further transformed into dependent product types with the help of another auxiliary function *Aux*. This function takes a pair $(e_{\mathbf{opt}}, e)$ (obtained by *GenContracts*) together with the new datatype name τ as an input and returns the base type and its refinement for the tail part. If there was no recursive call on the tail in a given execution path (namely, $e_{\mathbf{opt}} = \mathit{None}$), then the base type is int list and the refinement is e', obtained from e by replacing other recursive occurrences of f with the function itself. Otherwise, the base type is the new datatype applied to the first argument e'' to the recursive call; the refinement is essentially e' (except a cast back to int list). For example, for `sorted`, we obtain

$$T_1 = z_1{:}\mathsf{int} \times \{z_2{:}\mathsf{int\ list} \mid e_{21}\}$$

from (None, e_{21}) and

$$T_2 = z_1{:}\mathsf{int} \times \{z_2{:}\tau \mid \mathsf{let}\ z_2 = \langle \mathsf{int\ list} \Leftarrow \tau \rangle^{\ell}\ z_2\ \mathsf{in}\ e_{22}\}$$

from $(\mathit{Some}\,(), e_{22})$. T_1 is a type for singleton lists, which are trivially sorted and T_2 is a type for a list where the head is equal to or less than the second element and the tail is of type τ, which is supposed to represent sorted lists.

Finally, we combine T_i to make a complete datatype definition. The translation of `sorted` will be:

```
type sorted_t =
  SNil   || []   of {z:unit|true}
| SCons1 || (::) of z1:int × {z2:int list|e21}
| SCons2 || (::) of
    z1:int ×
    {z2:sorted_t|
     let z2 = ⟨int list ⇐ sorted_t⟩^ℓ z2 in e22}
```

Although the datatype `sorted_t` certainly represents sorted lists, its type definition is different from `slist2` given in the beginning of this chapter. The difference comes from the fact that the case for (::) has a case analysis, one of whose branch has a recursive call. While it is possible to "merge" the argument types for `SCons1` and `SCons2` to make a two-constructor datatype, it is difficult in general. It is interesting future work, however, to investigate how to generate type definitions closer to programmers' expectation.

4.3.2 Correctness

We prove that the translation is correct in the sense that the cast from a refinement type to the datatype obtained by the translation always succeeds and vice versa. We use a metavariable F to denote the recursive function used to define the refinement type in the typing judgment and the evaluation relation. We write $\langle \Sigma, \delta \rangle; \Gamma \vdash e : T$ and $\langle \Sigma, \delta \rangle \vdash e_1 \longrightarrow e_2$ to make a type definition environment Σ and a constructor choice function δ explicit in the typing judgment and the evaluation relation.

We start with defining a class of predicate functions which can be given to the translation.

Definition 8. *A recursive predicate function*

$$F \;=\; \texttt{fix}\, f(y{:}T, x{:}\mathsf{int\ list}) = \mathsf{match}\ x\ \mathsf{with}\ [\,] \to e_1 \;\mid\; z_1 :: z_2 \to e_2$$

is translatable under Σ *if*

- $(\Sigma, \emptyset); \emptyset \vdash F \;:\; T \to \mathsf{int\ list} \to \mathsf{bool}$,
- $(\Sigma, \emptyset); y{:}T \vdash e_1 \;:\; \mathsf{bool}$, *and*
- $(\Sigma, \emptyset); f{:}T \to \mathsf{int\ list} \to \mathsf{bool}, y{:}T, z_1{:}\mathsf{int}, z_2{:}\mathsf{int\ list} \vdash e_2 \;:\; \mathsf{bool}$.

We omit Σ if it is clear from the context or not important. The empty constructor choice function in Definition 8 means that F does not contain run-time terms.

We first show that the translation *Trans* always generates a well-formed datatype definition.

Theorem 10 (Translation Generates Well-Formed Datatype)**.** *Let* Σ *be a well-formed type definition environment and* F *be a translatable function under* Σ*. Then, the type definition* $\mathit{Trans}\,(F)$ *is well formed under* Σ*.*

The next theorem states that a cast from a refinement type to the generated datatype always succeeds.

Theorem 11 (From Refinement Types to Datatypes)**.** *Let* Σ *be a well-formed type definition environment,* F *be a translatable function under* Σ*, and* τ *be the name of the datatype* $\mathit{Trans}\,(F)$*. Then, there exists some computable well-formed choice function* δ *such that, for any* $e = \langle \tau\langle e_0\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e_0\ x\}\rangle^{\ell}\ v$*, if* $(\Sigma, \mathit{Trans}\,(F)); \emptyset \vdash e \;:\; \tau\langle e_0\rangle$*, then* $\langle(\Sigma, \mathit{Trans}\,(F)), \delta\rangle \vdash e \longrightarrow^{*} v'$ *for some* v'*.*

Proof. By Lemma C.4.9. □

It is a bit trickier to prove the converse because the first argument to a predicate function is always evaluated whereas a parameter to a datatype is not. So, the converse holds under the following termination condition on a datatype.

Definition 9 (Termination)**.** *Let* Σ *be a type definition environment and* δ *be a constructor choice function. A closed term* e terminates at a value under Σ and δ*, written as* $\langle \Sigma, \delta\rangle \vdash e\downarrow$*, if* $\langle \Sigma, \delta\rangle \vdash e \longrightarrow^{*} v$ *for some* v*. We say that* argument terms to datatype τ in v terminate at values under Σ and δ*, written as* $\langle \Sigma, \delta\rangle \vdash v\downarrow_{\tau}$*, if, for any* E*,* $C \in \mathit{CtrsOf}\,(\tau)$*,* e_1 *and* v_2*,* $v = E[C\langle e_1\rangle v_2]$ *implies* $\langle \Sigma, \delta\rangle \vdash e_1\downarrow$*.*

Theorem 12 (From Datatypes to Refinement Types)**.** *Let* Σ *be a well-formed type definition environment,* F *be a translatable function under* Σ*, and* τ *be the name of the datatype* $\mathit{Trans}\,(F)$*. Then, there exists some computable well-formed choice function* δ *such that, for any* $e = \langle\{x{:}\mathsf{int\ list} \mid F\ e_0\ x\} \Leftarrow \tau\langle e_0\rangle\rangle^{\ell}\ v$*, if* $\langle(\Sigma, \mathit{Trans}\,(F)), \delta\rangle; \emptyset \vdash e \;:\; \{x{:}\mathsf{int\ list} \mid F\ e\ x\}$ *and* $\langle(\Sigma, \mathit{Trans}\,(F)), \delta\rangle \vdash v\downarrow_{\tau}$*, then* e *terminates at a value under* $\langle(\Sigma, \mathit{Trans}\,(F)), \delta\rangle$ *and* δ*.*

Proof. By Lemma C.4.12. □

We expect that the termination condition would not be needed if we change the semantics to evaluate argument terms to datatypes.

4.3.3 Efficiency Preservation

In addition to correctness of the translation, we are also concerned with the following question "When I rewrite my program so that it uses the generated datatype, is it as efficient as the original one?" To answer this question positively, we consider the asymptotic time complexity of a cast *for successful inputs* (which we simply call the complexity of a cast), and show that the complexity of a cast from int list to its refinement is the same as that of a cast from int list to the datatype obtained from its refinement. Here, we consider only successful inputs because we are mainly interested in programs (or program runs) that do not raise blame, where checks caused by casts are successful.[3]

This efficiency preservation is obtained from Theorem 11 and the following observation. As stated in Theorem 11, we can construct a *computable* choice function. In fact, the algorithm of the choice function can be read off from the proof of Theorem 11: it returns constructors of the generated datatype from the execution trace of the refinement checking. Moreover, the orders of both the execution time of the algorithm and the size of output constructors from the algorithm are linear in the size of the input execution trace, which is proportional to the execution time of the refinement checking. Thus, the asymptotic time complexities of computation of the constructors and constructor replacement are no worse than that of the refinement checking.

From this observation, we can implement the cast from int list to the generated datatype by (1) checking the refinement (given to the translation) and (2) the constructor generation and replacement described above. Since the complexity of the second step is the same as that of the refinement checking, the complexities of the cast from int list to a refinement type and the generated datatype are the same.

4.3.4 Extension: Binary Trees

We informally describe how to extend the translation algorithm for lists to trees, a kind of data structure with a data constructor which has more than one recursive part. Here, we take binary trees as an example and show how to obtain a datatype for binary search trees from a predicate function. Although this section deals with only binary trees, this extension can be adapted for other data structures.

A datatype for binary trees and a recursive predicate function which returns whether an argument binary tree is a binary search tree or not are defined as follows:

```
type bt = L | N of int * t * t

let rec bst (min,max:int*int) (t:bt) =
 match t with
 | L          -> true
 | N (x,l,r) -> min<=x and x<=max and
                bst (min,x) l and bst (x,max) r
```

Let τ be a name for the new datatype.

The translation algorithm first calls *GenContracts* with the second branch of `bst`. Observing the predicate function `bst`, we find that the body calls `bst` itself recursively for different recursive parts (`l` and `r`) with different auxiliary arguments (`(min,x)` and `(x,max)`). Thus, *GenContracts* for binary trees looks for the first argument to each recursive call, unlike *GenContracts* for lists, which stops searching for a recursive call after finding one recursive call. For our running example, taking

[3] We conjecture that, for inputs that lead to blame, the time complexity is also preserved by the translation but a proof is left for future work.

the branch for constructor `N`, *GenContracts* for binary trees returns the singleton set $\{(\mathit{Some}\,(\mathit{min}, x), \mathit{Some}\,(x, \mathit{max}), \mathit{min} \leq x \text{ and } x \leq \mathit{max})\}$ (where we use the operator and instead of if expression for brevity), of which the first two optional terms denote arguments for left and right subtrees, respectively.

Next, for each element in the output from *GenContracts*, a dependent product type will be built. In this case, we obtain $T = x{:}\mathsf{int} \times l{:}\tau\langle(\mathit{min}, x)\rangle \times \{r{:}\tau\langle(x, \mathit{max})\rangle \mid \mathit{min} \leq x \text{ and } x \leq \mathit{max}\}$. As we have seen for lists, casts from $\tau\langle e\rangle$ back to bt may have to be inserted.

Finally, the translation makes a datatype definition by using these type arguments and the contract. For `bst`, the corresponding datatype is given as follows:

```
type t ⟨min:int,max:int⟩ =
 | SL
 | SN of x:int × l:t⟨min,x⟩ ×
           {r:t⟨x,max⟩|min<=x and x<=max}
```

4.3.5 Discussion

The translation algorithm works "well" for list-processing functions, in the sense that there is no reference to the input predicate function in the generated datatype, if their definitions meet the two requirements: (1) recursive calls are given the tail part of the input list and occur linearly for each execution path; (2) free variables in arguments to recursive calls are only the argument variable y and the head variable z_1; and (3) the given functions are written in the tail recursion form. In contrast, there can remain recursive calls to an input predicate function in the generated datatype when the predicate function does not meet these requirements. This happens (1) when there is a recursive call on lists other than the tail of the input, (2) as in the following (admittedly quite artificial) example, when recursive calls occur twice or more in one execution path:

```
let rec f () (x:int list) =
  match x with
  | []     -> true
  | z1::z2 -> f () z2 and f () z2
```

(3) when e_2 includes non-branching constructs as in

```
let rec f (y:int) (x:int list) =
  match x with
  | []     -> true
  | z1::z2 -> let z = 5 + y in f z z2
```

or (4) when there is a possibly successful execution path which uses negation of results of recursive calls as in:

```
let rec length_is_even () (x:int list) =
  match x with
  | []     -> true
  | z1::z2 -> not (length_is_even () z2)
```

because we do not support "negation over types." In these cases, generation of a datatype itself succeeds but the obtained datatype is probably not what we expect because `f` is not eliminated. In some cases, fortunately, it is possible to transform refinements so that translation works well. For example, the function `length_is_even` above can be transformed to take the result in the original function as an argument:

```
let rec length_is_even' (even:bool) (x:int list) =
  match x with
  | []     -> even
  | z1::z2 -> if length_is_even' (not even) z2 then true else false
```

where the `if` construct is redundant but necessary for success of translation in accordance with the formal algorithm in Figure 4.7. Intuitively, $\{x$:int list $\mid$ length_is_even $x\}$ is equivalent to $\{x$:int list $\mid$ length_is_even$'$ true $x\}$. The translation result of this function is:

```
type even_t' (even:bool) =
  | ENil  || []   of {z:unit|even}
  | ECons || (::) of z1:int × z2:event_t⟨not even⟩
```

(from which we remove a redundant constructor) and has no calls to the predicate function. We would not be, however, able to transform all refinements, in particular, for data structures with two or more recursive components in such a way. A study on such program transformation is left as future work.

We can generalize to other data structures such as trees the requirements to eliminate all references to an input predicate function from the derived datatype. Similarly to those of lists, the requirements are imposed on execution paths in the input function. Let f be an input predicate function and y be a parameter variable to f. Suppose that we take an execution path in f. Since the body of f starts with deconstructing a given data structure by pattern match, variables to denote components of the data structure are introduced in the execution path. We call such variables $z_1, ..., z_n$ and, without loss of generality, suppose that, for any i, z_{i+1} is declared immediately after z_i. In `bst` given in Section 4.3.4, y corresponds to `min` and `max` and z_1, z_2, and z_3 to `x`, `l`, and `r`. Then, as a condition under which translation works well, we require the rest of the execution path to take the conjunctive normal form

$$e_1 \texttt{ and } ... \texttt{ and } e_m \texttt{ and } f\ e'_1\ z'_1 \texttt{ and } ... \texttt{ and } f\ e'_n\ z'_n$$

where: (1) $z'_1, ..., z'_n$ are distinct variables to denote some recursive components of the deconstructed data structure (note that $\{z'_1, ..., z'_n\}$ is a subset of $\{z_1, ..., z_n\}$); (2) $e_1, ..., e_m$ contain, as free variables, only y and $z_1, ..., z_n$; and, (3) for any $i > 0$, when $z'_i = z_j$, e'_i contains only y and $z_1, ..., z_{j-1}$. When the input function f satisfies these requirements, there remain no calls to f in the derived datatype. In the case for the second branch of the pattern match in `bst`, the rest of the execution path takes the form `min<=x and x<=max and bst (min,x) l and bst (x,max) r` and it meets the requirements: (1) `l` and `r` are distinct; (2) only parameter variables `min` and `max` and variable `x` introduced by the pattern match occur free in `min<=x` and `x<=max`; (3) `(min,x)` and `(x,max)` also contain only `min`, `max`, and `x` (note that `x` is declared before `l` and `r` in the second clause of the pattern match). Since another execution path `true` in `bst` satisfies the requirements, the datatype derived from `bst` has no references to `bst`.

Although our translation works well for many predicates, there is a lot of room to improve. First, the current translation algorithm could generate a datatype with too many constructors even if some of them can be "merged." For example, we demonstrated that the translation generated a datatype with three constructors from predicate function `sorted`, but we can give a datatype with only two constructors for it as shown in Section 4.1.1. Second, our translation algorithm works only for a single recursive Boolean function and so we cannot obtain a datatype from other forms of refinements, for example, conjunction of two predicate function calls. This also means

that the translation cannot deal with a predicate function that returns additional information by using, say, an option type.

Our translation assumes an input refinement to be of a certain form. We think, however, that it is not so restrictive, because we can transform refinements before applying our method. For example, a predicate function of the form

$$\mathsf{if}\ e_1\ \mathsf{then}\ (\mathsf{match}\ x\ \mathsf{with}\ [\,] \rightarrow e_{11}\ \mid\ z_1 :: z_2 \rightarrow e_{12})\ \mathsf{else}\ e_2$$

can be transformed to

$$\begin{array}{rl} \mathsf{match}\ x\ \mathsf{with}\ [\,] & \rightarrow \mathsf{if}\ e_1\ \mathsf{then}\ e_{11}\ \mathsf{else}\ e_2\ \mid \\ z_1 :: z_2 & \rightarrow \mathsf{if}\ e_1\ \mathsf{then}\ e_{12}\ \mathsf{else}\ e_2. \end{array}$$

Even if such transformation cannot be applied, we can always insert pattern matching on the input list in the beginning of a predicate refinement. (It may be the case, though, that we do not obtain an expected type definition.)

Chapter 5

Related Work

5.1 Integration of Static and Dynamic Typing

This section compares the blame calculus given in Chapter 2 with work on integration of static and dynamic typing.

Dynamic typing in statically typed languages A theory of dynamic typing in statically typed languages was studied by Abadi et al. [1]. A motivation of their work is to safely deal with data—e.g., binary objects stored in an external storage or flowed from other processes—difficult to determine their types statically, rather than transformation from an untyped program to a typed one. To this end, Abadi et al.'s calculus regards the dynamic type as a kind of infinite sum types. Run-time values, dynamic values for short, of the dynamic type contain type information of injected values as run-time tags. Tags in their calculus are types themselves whereas ones in our and earlier work [120, 7, 104] on blame calculi are kinds (ground types) of types. Dynamic values can be deconstructed into injected values by type pattern matching construct `typecase`. Given a dynamic value, the `typecase` construct examines whether each type pattern matches with the type tag of the dynamic value and chooses the branch corresponding to the first pattern that matches with the tag; if there are no such patterns, the `else` branch is chosen. Since `typecase` expressions merely decompose dynamic values and do *not* coerce them to concrete types, well-typed programs in Abadi et al.'s calculus never cause run-time type errors. As the price, however, when programmers want to coerce dynamic values to concrete types, they have to decompose dynamic values explicitly by using `typecase` expressions—this programming style would make smooth transformation from untyped programs to typed ones so difficult because programmers would make efforts to eliminate `typecase` expressions manually. In addition, the lack of blame makes it difficult to discuss which component is responsible for a flow of an unexpected value, while blame calculi state it as blame theorem.

Henglein [51] also proposed a lambda calculus with both static and dynamic typing. In Henglein's calculus, coercions play an important role in interaction between typed and untyped parts. That work designed coercion insertion algorithms to obtain an intermediate term with coercions from a source term without coercions and showed that the algorithms insert coercions as less as possible. Coercions are similar to casts in our work but that work does not give semantics of coercions.

Gradual typing Gradual typing, coined by Siek and Taha [101][1], is a typing style for transforming untyped programs to typed ones "gradually"—in gradually typed calculi, untyped programs can be rewritten to typed ones just by adding appropriate type annotations. Since the seminal work by Siek and Taha, gradual typing has been studied extensively. For example, earlier work studies gradual typing with higher-order functions [101], mutable references [101, 52, 105], objects [102], generics [54], first-class classes [109], and so on and applies the idea of combining static and dynamic verification to other systems [28, 95]; interested readers can refer to the survey by Siek et al. [104]. Execution models of gradually typed calculi are given by calculi with run-time casts (or contracts): gradually typed programs are translated to intermediate terms where casts are inserted to inject typed values to and, conversely, draw injected values from the dynamic world at run time. In this sense, we expect our calculus to be an execution model of gradually typed languages with shift/reset.

Blame calculus Blame calculi are execution models of intermediate languages for gradual typing, proposed to study responsibility for blame. Roughly speaking, there are two kinds of blame calculi: one adopts contracts [113, 109, 27, 110] for run-time checking and the other adopts casts [101, 52, 102, 120, 7, 54, 105, 104]. Our calculus belongs to the latter.

The first blame calculus was proposed by Tobin-Hochstadt and Felleisen [113] to study an interlanguage migration process from untyped programs to typed ones and its correctness. Their goal is to establish the gradual typing version of Milner's slogan [75] for static typing (the following is quoted from their paper):

> "typed modules can't go wrong, and all run-time errors originate in untyped modules."

This property is called *blame theorem*, named by Wadler and Findler [120]. All gradually typed languages should satisfy blame theorem because raising run-time type errors are a mechanism of dynamically typed languages to notify that something unexpected happens and so statically typed components are expected never to cause such errors. Tobin-Hochstadt and Felleisen took contracts for run-time checking and supposed that a blame label corresponds to one module. Their blame calculus supports only positive blame, although the succeeding work follows Findler and Felleisen's work [35] and supports blame of finer forms (positive and negative blame).

Wadler and Findler [120] refined Tobin-Hochstadt and Felleisen's blame calculus to make discussion about blame easier and investigated finer conditions under which blame does not happen. Wadler and Findler showed that the *more* precisely typed side, which need not be fully typed, never triggers run-time type errors (Tobin-Hochstadt and Felleisen [113] supposed that a module is fully typed or fully untyped). They also discovered that the notion of being "more precisely typed" can be formalized as naive subtyping. Although it is based on Wadler and Findler, our calculus is *not* a superset of their blame calculus because their calculus supports refinement types, which refine base types with contracts (arbitrary Boolean expressions), whereas our calculus does not. We believe that it is easy to introduce refinement types to our calculus when we are concerned with *pure* [8] refinements because pure expressions appear to involve no control effects. However, it is not trivial to support refinement types with *impure* contracts, that is, Boolean expressions which can manipulate their contexts because what is meant by "contexts of contracts" is unclear.

[1] Tobin-Hochstadt and Felleisen [113] also studied gradual typing independently of Siek and Taha.

Blame calculus with delimited-control operators Most closely related work is Takikawa et al. [110]; they have also studied integration of static and dynamic typing in the presence of control operators. They proposed a contract system for programs with control operators in Racket [39] and showed the Blame Theorem, following Dimoulas et al. [27].

Dimoulas, Tobin-Hochstadt, and Felleisen refined blame theorem for blame calculi with contracts and established a proof technique for showing it. The proof of the Blame Theorem in their work rests on *complete monitoring*, a property justifying both their blame assignment and contract checking strategies. Complete monitoring intuitively states that all value flows between modules are monitored by appropriate contracts; especially, it says that, if a blame with label l aborts program execution, then a value originated by the module corresponding to l violates a contract having an obligation of monitoring values migrated from l. The blame calculus tracks where checked values are migrated from by introducing the notion of ownership for terms and values; a module can manipulate only values which are originated in the module itself or migrated from other modules through contract checking. Obligations are assigned to contracts by following so-called even-odd rule [35].

Naturally, differences between Wadler and Findler [120] and Dimoulas et al. [27] are ones between us and Takikawa, Strickland, and Tobin-Hochstadt. We allow a single module to have both typed and untyped parts whereas they allow it to have either. We show that the more precisely typed (not necessarily fully typed) side does not trigger blame by means of positive and negative subtyping whereas they show that a fully typed module does not by means of ownership and obligation.

Takikawa, Strickland, and Tobin-Hochstadt studied Sitaram's delimited-control operators `fcontrol` and `%` [106] with prompt tags [106, 49, 30]. A `fcontrol` expression with a prompt tag captures the delimited continuation up to the closest `%` with the same tag. Since their calculus always needs prompt tags to manipulate delimited continuations, it monitors capture and call of delimited continuations by providing new forms of contracts for wrapping prompt tags. They also dealt with continuation marks [23] and supported wrapper objects for them. Our work focuses on shift/reset and our cast semantics produces a lambda abstraction as a wrapper of the target value. We also define a CPS transformation for our calculus and investigate the relationship between our calculus and the CPS transformation; a study of CPS transformation is out of the scope of their work. Although shift and reset can be implemented by using control operators in their work [39], it is not very clear whether their contract system can simulate our casts for function types with answer types naturally.

5.2 Integration of Static and Dependent Typing

This section compares our manifest calculi, $\mathrm{F}^{\sigma}_{\mathrm{H}}$ in Chapter 3 and $\lambda^{H}_{\mathbf{dt}}$ in Chapter 4, with work on integration of static and dependent typing in greater detail. We start with other manifest calculi and then discuss other related work; the comparison of $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and $\lambda^{H}_{\mathbf{dt}}$ is described in Section 4.2.5.

Simply typed manifest contract calculi A simply typed contract calculus λ_{H}, originated by Flanagan [37], is proposed as a theoretical foundation of hybrid type checking. As discussed in Section 3.1, however, the metatheory of the original λ_{H} is flawed due to support for subsumption in terms of subtyping, which demands that well typedness of terms occur at negative positions and makes it unclear whether the type system is

well defined, while subtyping plays an important role in the proof of type soundness. The manifest calculus of Gronski and Flanagan [45] has the same problem.

Knowles and Flanagan [64] and Greenberg, Pierce, and Weirich [44] have revised the original λ_{H} to resolve the flaw; we write Knowles and Flanagan's λ_{H} KF and Greenberg et al.'s λ_{H} GPW. To avoid the circularity, they give another source of "well-typed" values: hence, the denotations of types. Both KF and GPW define syntactic term models of types to use as a source of values in subtyping, though the specifics differ. After adding subtyping and denotational semantics, the type systems of both KF and GPW are well defined clearly. Moreover, as a key property of their calculi, they proved semantic soundness theorems (we write $[\![T]\!]$ for the denotations of type T):

$$\Gamma \vdash e : T \text{ and } \Gamma \vdash \sigma \text{ implies } \sigma(e) \in [\![\sigma(T)]\!]$$

in particular

$$\emptyset \vdash e : T \text{ implies } e \in [\![T]\!].$$

This theorem is sufficient for soundness of GPW whereas insufficient for KF—this difference comes from the difference of definitions of $[\![-]\!]$—and so Knowles and Flanagan have proved syntactic type soundness later.

Although these calculi have been *proven* to be sound, the situation in KF and GPW is somewhat unsatisfying. We set out to prove syntactic type soundness and ended up proving semantic type soundness along the way. While not a serious burden for a language as small as λ_{H}, having to use semantic techniques throughout makes adding some features—polymorphism, state and other effects, concurrency—difficult. For example, a semantic proof of type soundness for $\mathrm{F}^{\sigma}_{\mathrm{H}}$ would be very close to a proof of parametricity—must we prove parametricity while proving type soundness? To avoid such a sad situation, Belo et al. propose a syntactic construction of manifest calculi but there are technical flaws in their calculus; see the discussion in Section 3.1.3.

The metatheories of $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and $\lambda^{H}_{\mathbf{dt}}$ are entirely syntactic and correct. Similarly to F_{H}, they solve the problem by avoiding subtyping—which is what forced the circularity and denotational semantics in the first place—and introducing (T_EXACT), (T_CONV), and convertibility $\equiv$ instead. The (T_EXACT) rule

$$\frac{\vdash \Gamma \qquad \emptyset \vdash v : T \qquad \emptyset \vdash \{x{:}T \mid e\} \qquad [v/x]e \longrightarrow^{*} \mathsf{true}}{\Gamma \vdash v : \{x{:}T \mid e\}} \text{ (T\_EXACT)}$$

needs some care to avoid vicious circularity: it is crucial to stipulate v and $\{x{:}T \mid e\}$ be closed. If we "bit the bullet" and allowed nonempty contexts there, then we would need to apply a closing substitution to $[v/x]e$ before checking if it reduces to true but it would lead to the same circularity as subtyping we discussed above. As for (T_CONV) and convertibility, convertibility is much simpler than GPW and Belo et al. [14]. It does not, unfortunately, completely simplify the proof: we must prove that our conversion relation is a weak bisimulation to establish *cotermination* (Lemma 11) before proving type soundness.

SAGE language Gronski et al. [46] develop SAGE language, which supports subsumption for subtyping, casts, general refinements, polymorphism, recursive functions, recursive types, the dynamic type, the Type:Type discipline. SAGE avoids the circularity of Flanagan's λ_{H}, changing formalization of subtyping: in SAGE, $\{x{:}T \mid e_1\}$ is a subtype of $\{x{:}T \mid e_2\}$ if a theorem prover can prove the implication from e_1 to e_2. Since the theorem prover is independent of SAGE, the type system is well defined.

Naturally, the metatheory of SAGE rests on the theorem prover. SAGE states axioms strong enough to show type soundness—for example, it requires the prover to be able to show $[e_1/x]e$ evaluates to true iff $[e_2/x]e$ does when $e_1 \longrightarrow e_2$, which works similarly to cotermination in $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and $\lambda^{H}_{\mathbf{dt}}$. Although Gronski et al. have shown type soundness of SAGE, they do not deal with parametricity, while we show it in $\mathrm{F}^{\sigma}_{\mathrm{H}}$. In fact, it is difficult to show parametricity in calculi with recursive functions [87], recursive types [5], the dynamic type [69], and/or Type:Type. In addition, axiomatization of theorem provers could bring us to an unsatisfactory situation. For example, the axiom system of Gronski et al. is inconsistent, though fixed by Knowles [61].

Dependent types with dynamic typing Ou et al. [82] study integration of certified and uncertified program fragments—all refinements in certified parts are checked statically whereas all those in uncertified parts are checked at run time. They model static checking as subtyping checking and dynamic checking as compilation to predicate checking with `if`-expressions. Their calculus deal with the issues of preservation by supporting a special typing rule to assign "selfified" types to terms and subsumption for subtyping. Unlike manifest calculi, they restrict refinements (and so also arguments to dependent functions) to be syntactically pure in order to make static checking decidable. They also axiomatize requirements on theorem provers, like Gronski et al. [46].

Static analysis using path information Much work on static program analysis (e.g., [53, 84, 124, 20, 80, 91, 59, 63, 22]) employs path information of conditional expressions—for example, when `if`-expressions are verified, the conditional expressions are supposed to hold in then-expressions whereas they are not to hold in else-expressions. In a sense, such information can be thought as "dynamic" because it is a result of an analysis of what values are examined at run time. Although $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and $\lambda^{H}_{\mathbf{dt}}$ do not keep track of path information directly, we can simulate by encoding an if-expression (if e_1 then e_2 else e_3) in source programs as syntax sugar of:

$$\begin{array}{rl} \mathsf{if}\ e_1\ \mathsf{then} & (\mathsf{let}\ x = \langle \mathsf{bool} \Rightarrow \{y{:}\mathsf{bool} \mid e_1\}\rangle^{l}\ \mathsf{true}\ \mathsf{in}\ e_2) \\ \mathsf{else} & (\mathsf{let}\ x = \langle \mathsf{bool} \Rightarrow \{y{:}\mathsf{bool} \mid \mathsf{not}\ e_1\}\rangle^{l}\ \mathsf{true}\ \mathsf{in}\ e_3) \end{array}$$

where x and y are fresh. Under this encoding, e_2 and e_3 are typed under a binding that x is given type $\{y{:}\mathsf{bool} \mid e_1\}$ and $\{y{:}\mathsf{bool} \mid \mathsf{not}\ e_1\}$, respectively. This corresponds to a path-sensitive typing rule for if-expressions, found, e.g., in Rondon et al. [91]:

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma, e_1 \vdash e_2 : T \quad \Gamma, \mathsf{not}\ e_1 \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}$$

(A Boolean expression e in a context intuitively means "if e is true.") Such path information would be useful if we consider static verification for manifest contracts.

5.3 Dependent and/or Refinement Type Systems

Manifest contracts embed contract information into types as refinement types. The term "refinement types" seems to have many related but subtly different meanings in the literature. We use this term for types to denote subsets in some way or another. Refinement types are intensively studied in the context of static program verification.

In Freeman and Pfenning [40], datatypes can be refined by giving data constructors appropriate types. For example, one may give [] a special type null and cons a special

type int $\to$ null $\to$ singleton_list, which means that, if cons takes an element and the empty list, then it yields a singleton list. Here, null and singleton_list are atomic type names. They did not allow refinement types to take arbitrary contracts to make type checking and type inference decidable. On the other hand, they combined refinement types with intersection types to express overloaded functionality of a single constructor.

Xi and Pfenning [123, 122] have designed and developed practical programming languages which support a restricted form of dependent types. Kawaguchi et al. [59] and Vazou et al. [116] have devised type inference algorithms for statically typed lambda calculi with refinement types and recursive refinements, which provides recursive types with refinements, and have implemented it for OCaml and Haskell, respectively. The refinements used there are derived from decidable languages such as (extensions of) Presburger arithmetic because their main focus is static verification. Our type system allows arbitrary Boolean predicates.

Our datatypes studied in Chapter 4 resemble *inductive datatypes* found in interactive proof assistants such as Coq [112] or Agda [4]. Aside from compatibility relation and casts, one difference between our datatypes and inductive datatypes in such systems is that we treat a formal argument $\langle x \rangle$ to a datatype to be parametric whereas inductive types can restrict it by specifying result datatypes of data constructors. For example, in Coq, a type `Vector` of finite lists can be written as follows:

```
Inductive Vector : int -> Type :=
  nil  : Vector 0
| cons : forall n : int, int -> Vector n -> Vector (n+1)
```

where parameters to `Vector` denote lengths of finite lists. Here, each data constructor restricts a parameter given to its result datatype: the parameter in `nil` is `0` because `nil` means the empty list; the parameter in `cons` is `n+1`, given a tail part of length `n`, because a list constructed by `cons` has one additional element. Our calculus $\lambda^{H}_{\mathbf{dt}}$ does not allow such datatypes. However, since an formal argument to a datatype can appear in a refinement, conditions on an actual argument can be enforced and so we do not lose much expressiveness. For example, in our calculus, `Vector` above can be represented as follows:

```
type vector ⟨n:int⟩ =
  Nil  of {unit|n = 0}
| Cons of {int |n > 0} × vector ⟨n-1⟩
```

5.4 Parametricity with Dynamic Type Analysis

Parametricity (or called abstraction theorem) [90, 119] is the essence of the notions of "abstract types" [77] and "information hiding" [85, 107], meaning intuitively that all instances of polymorphic values behave in the same way. Parametricity have been studied well for a long time in the context of static typing [90, 119, 87, 5, 6, 29], but it is not easy to achieve parametricity in languages, including manifest calculi, with run-time type analysis because the behavior of a polymorphic function examining run-time instances of type variables can be different whatever types are substituted for the the type variables; for example, Abadi et al.'s polymorphic calculus with the dynamic type [1, 3] does not parametricity because it allows instances of type variables to be analyzed by injecting their inhabitants to the dynamic type.

A lot of work have dealt with parametricity in such languages. The common key idea is to use dynamic sealing [79, 85, 107] to prevent run-time instances of type variables from being analyzed. Rossberg [92], who attacked the problem for the first time as far as we know, introduced run-time generation of type names in order to refuse run-time analysis of abstract types but did not show parametricity. Neis et al. [81] have shown that parametricity holds if type variables are guarded by dynamically generated type names. In order to bring the notion of parametricity to dynamically typed languages, Guha et al. [48] gave so called parametric contracts, which make values "abstract", that is, the contracts prevent type analysis of values migrated through them, though they did not prove parametricity. Matthews and Ahmed [69] have studied parametricity for a multi-language which combines a polymorphic and an untyped lambda calculus and used dynamic sealing to prevent observing what types are substituted for type variables; their proof of parametricity, unfortunately, has a flaw [81]. Ahmed et al. [7] introduced a polymorphic blame calculus where type substitutions form syntactic "barriers", which can seem to be dynamic sealing with scope. They did not prove parametricity, either. The notions of dynamic sealing in the style of Matthews and Ahmed [69] and syntactic barriers in Ahmed et al. [7] inspired the delayed substitution in $\mathrm{F}^{\sigma}_{\mathrm{H}}$.

$\mathrm{F}^{\sigma}_{\mathrm{H}}$ has parametricity. Although casts are a kind of run-time type analysis, $\mathrm{F}^{\sigma}_{\mathrm{H}}$ rejects dynamic analysis of what types are instantiated for type variables by means of compatibility and delayed substitutions. Compatibility restricts the target type of a cast from a type variable to be (refinements of) the type variable itself. Thus, well-typed terms cannot examine whether type variables are replaced with, say, base types. Delayed substitution guarantees that a cast for a type variable behaves in the same way whatever types are substituted for the type variable. Delayed substitutions at casts work as dynamic sealing, though they do not equip unsealing operation like Matthews and Ahmed [69].

5.5 Contracts for Datatypes

There has been much work about lambda calculi with higher-order contracts since the seminal work by Findler and Felleisen [35], but little of them considers algebraic datatypes in detail and compare the two approaches to datatypes with contracts—in particular, as far as we know, there is no work on conversion between compatible datatypes. One notable exception is Findler et al. [36], who compare the two approaches to datatypes and introduce lazy contract checking in an eager language. Lazy contract checking delays contract checking for arguments to data constructor until they are used. As they already point out, one drawback of lazy contract checking is that it is not suitable for checking, where relationship between elements in a data structure is important. For example, if we take the head of an arbitrary list, which is subject to sortedness checking, it simply returns its head discarding the tail without verifying the tail is sorted. Chitil [21] also made a similar observation in the work on lazy contracts in a lazy language.

Knowles et al. [65] developed Sage, a programming language based on a manifest contract calculus with first-class types and dynamic type. Sage can deal with datatypes with refined constructors by Church-encoding, but does not formalize them in its core calculus. In particular, Knowles et al. did not clarify how casts between datatypes work. Dminor [15], proposed by Bierman et al., is a first-order functional programming language with refinement types, type-test and semantic subtyping. The combination of

these features is as powerful as various types such as algebraic datatypes, intersection types, and union types can be encoded. Unlike our calculus, Dminor does not deal with higher-order functions and dynamic checking with type conversion.

Xu [125] developed a hybrid contract checker for OCaml. In the static checking phase, the checker performs symbolic simplification of program components wrapped by contracts, with the help of context information, to remove blames. If a blame remains in the simplified programs, the compiler reports errors, or it issues warnings and leaves contract checking to run time. Although the checker supports variant types (i.e., datatypes where constructors have no refinements), it does not take care of relationship between elements in data structures nicely. For example, it seems that it cannot prove statically that the tail of a sorted list is also sorted unless programmers give axioms about sorted lists.

In a different line of work, Miranda [115], a statically typed functional programming language, provides datatypes with laws, which are rules to reconstruct data structures according to certain specifications. For example, we suppose that a datatype integer has three constructors Zero, Succ integer and Pred integer, and then a law converts Succ (Pred x) to x. More interestingly, Miranda can control application of laws by giving them conditional expressions. Using laws with conditionals, we can define lists which are sorted automatically. Both Miranda and our calculus provide a mechanism to convert data structures, but the purposes are different: in our work, type conversion is used only to check contracts, and so does not change "structures".

5.6 Systematic Derivation of Datatype Definitions

As mentioned in the beginning of Chapter 4, there is closely related work on systematic derivation of (indexed) datatype definitions.

McBride [70] propose the notion of *ornaments*, which provide a mechanism to extend and to refine datatypes in a dependently-typed programming language. For example, the definition of lists can be derived from that of natural numbers by adding an element type; and the definition of lists indexed by their lengths can be derived. As far as we understand, he does not consider deriving new type definitions by changing the number of data constructors, as is the case for our work. Also, it is not clear whether partial refinements (the case where an index cannot be assigned to some values of the original datatype) can be dealt with in this framework. Partial refinements are important in our setting because our refinement types are for excluding some values in the underlying types.

Atkey et al. [10] developed a derivation technique of inductive types from (partial) refinements from a category-theoretic point of view. Their approach is more general than our work in the sense that it can take recursive functions which are written in the fold form (more generally, paramorphism style [71], where recursive subcomponents of an argument can be referred to in the body of a recursive function) and return values other than Booleans. For example, it derives an inductive type for finite lists from the refinement type $\{x{:}\mathsf{int\ list} \mid \mathsf{length}\ x = n\}$. Our derivation algorithm, given in Section 4.3.1, cannot translate recursive functions which return nonBoolean values because it takes only *predicate* functions. It is possible in some cases, however, to transform a refinement using a recursive function which returns nonBoolean values to a predicate function by adding a parameter to denote the result of the original function and embedding refinement information into the function body (we also discussed a similar technique in Section 4.3.5). For example, given the function length defined as:

```
let rec length (x:int list) =
  match x with
  | []     -> 0
  | z1::z2 -> 1 + length z2
```

the refinement type $\{x$:int list $\mid$ length $x = n\}$ is transformed to $\{x$:int list $\mid$ length′ $n\ x\}$ where:

```
let rec length' (n:int) (x:int list) =
  match x with
  | []     -> n = 0
  | z1::z2 -> length (n-1) z2
```

and then we can derive a datatype from `length'` (unfortunately, it is unclear what recursive functions can be transformed in such a way, though). In fact, Atkey et al.'s derivation mechanism seems to work in a similar manner: it derives an inductive type like the following from the refinement type $\{x$:int list $\mid$ length $x = n\}$:

```
type vector ⟨n:int⟩ =
  | LNil  of { z:unit | 0 = n }
  | LCons of z1:int × n':int × { z2:vector⟨n'⟩ | n'+1 = n }
```

where argument `n` means the result of `length` and the cons constructor takes an additional integer `n'`, which denotes the result of the recursive call on the tail part. Although their approach is more generic, they have not studied the dynamic aspect of result inductive types. Inductive types derived by their approach are not always compatible with the original datatype in the sense introduced in Chapter 4—e.g., the inductive type `vector` above is not compatible with integer lists because the cons constructor in `vector` takes three arguments whereas the one in integer lists does two—whereas our translation always generates a compatible datatype, which enables dynamic conversion from and to the original datatype. In addition, their approach always derives an inductive type each constructor of which corresponds to one constructor in the original datatype exactly whereas our translation can generate a datatype with zero or more constructors corresponding to one in the original datatype, such as list_including in Section 4.1.1. Such datatypes would be useful to efficient dynamic checking.

A similar idea to our translation is found in Kawaguchi et al. [59], who develop a refinement type system for static verification of programs dealing with datatypes. They allow programmers to write special terminating functions called measures, which will be used as hints to the verifier by indexing a datatype with the measure information.

Chapter 6

Conclusion

6.1 This Thesis

This thesis has studied unifying the typing spectrum from dynamic and static typing to dependent typing. These typing styles are complementary—dependent typing has the most powerful static type systems, followed in order by static and dynamic typing whereas dynamic typing allows programs to pass static type checking most easily, followed by static and dependent typing. A challenge of the unification is to establish safe interaction between components in different typing styles, that is, values flown from an uncertified side do not invalidate properties in a certified side. This safety of interaction can be achieved by monitoring the behavior of components at run time. In this thesis, casts, a common tool to monitor value flows between components, play an important role in dynamic checking. We have extended two existing unification mechanisms based on casts—gradual typing, which combines static and dynamic typing, and manifest contracts, which does static and dependent typing—with practical programming features.

Chapter 2 has focused on integration of static and dynamic type checking in the presence of delimited-control operators shift/reset. We have proposed a new cast-based mechanism, which monitors all communications between typed and untyped code through delimited continuations and is inspired by Danvy and Filinski's type system. To justify the design of the cast semantics, we have defined a simply typed blame calculus with shift/reset and shown the Type Soundness, the Blame Theorem, and soundness of the CPS transformation. We have found additional axioms for the equational system in the target language in proving the soundness.

In Chapter 3, we have proposed $\mathrm{F}^{\sigma}_{\mathrm{H}}$ to combine parametric polymorphism and manifest contracts. We offer the first conjecture-free, completely correct operational semantics for *general* refinements, where refinements can apply to any type, not just base types. The cast semantics in $\mathrm{F}^{\sigma}_{\mathrm{H}}$ is designed to be insensitive to substitution, resting on delayed substitution and a new type conversion relation.

Chapter 4 has proposed datatypes for manifest contracts with the mechanism of casts between different but compatible datatypes, and proved type soundness of a manifest contract calculus $\lambda^{H}_{\mathbf{dt}}$ with datatypes. To study a relationship of refinement types and datatypes, we have given a formal translation from a refinement on lists to a datatype definition with refined constructors and proved the translation is correct. We have also found that the translation preserves the efficiency of casts.

6.2 Future Work

This thesis has advanced a theory for integrating different, complementary typing styles. We have not, however, reached our goal, development of a full-fledged language with dynamic, static, and dependent typing, yet. In fact, there are many challenges in theory and practice for achieving it—especially, design and implementation of the language. It is possible, unfortunately, that approaches in dynamic, static, or dependent typing cannot be applied to the language easily. An example of the difficulties is reduction of the time when programmers take to write type annotations. A natural solution in static and dynamic typing is use of type inference but support for dynamic typing would make the problem difficult to solve: does absence of type annotations in program components expect inferring their types or giving the dynamic type to them? Perhaps, in general, there may be no answers which all people agree on—the components are expected to have static types sometimes and the dynamic type at other times. So, we need a design not to "shoot yourself in the foot." Another, critical problem is efficiency of interaction between different typing styles. In particular, run-time checking of refinement types would cause significant run-time overhead. One idea to relax this problem would be to use results of dynamic checking for static checking as in SAGE language [46] but we need a deep study on it. Finally, a study of languages with all typing styles is left as future work.

Apart from design and implementation of the language, there are also many directions of future work, which we discuss in what follows.

Blame calculus First, development of a *surface* language for gradual typing with shift/reset is left as future work (our blame calculus is designed as an *intermediate* language of gradual typing). Such a language should satisfy criteria of gradual typing, advocated by, say, Siek et al. [104]. Second is an extension of our blame calculus with refinement types. Effects in refinements are obviously problematic. One possible solution would be to restrict refinements to be pure but it is interesting to investigate how such purity restriction can be relaxed. Third is to apply succeeding work about blame calculi, such as space-efficiency [52] and parametricity [7], to our calculus. In particular, an extension with parametricity would be challenging because it is not clear how control operators and the ν-operator in that work interact with each other. Fourth, we would like to develop a contract system corresponding to our calculus and to inspect more detailed relationship to the contract system of Takikawa et al. It is also interesting for such a contract system to consider dependency of function types with answer types. Finally, it is an interesting direction to extend blame calculi with other control operators such as control/prompt [32, 39] and a family of shift/reset in the CPS hierarchy [25]. In this work, we choose shift/reset because their type system and CPS transformation are well studied. In fact, CPS transformation for shift/reset has served as a guide to designing our cast mechanism. Given recent studies on relationship of control/prompt to their CPS transformation [100, 16, 30] and a type system for control/prompt [58], we leave an extension of blame calculi with control/prompt as interesting future work.

Manifest contracts We have proposed two variants of manifest contracts: $\mathrm{F}^{\sigma}_{\mathrm{H}}$, where refinements on casts would not be checked at run time if it is obvious from type information on casts that values satisfy refinements, and $\lambda^{H}_{\mathbf{dt}}$, where all refinements (in the target type on a cast) will be checked at run time. The latter does not need delayed substitution and so its metatheory is simpler than the former. Then, a natural question

is raised: can we have a sound, parametrically polymorphic manifest contract calculus without delayed substitution? Another question in polymorphic manifest contracts is about upcast elimination [64, 14], which says that removing upcasts—casts from a type to its supertype—does not change the behavior of a program in a certain sense and is a key theoretical property for static contract checking. We are working on a complete account of another polymorphic manifest contract calculus [96] with recursion, a parametricity relation that has a clear relationship to contextual equivalence, upcast elimination, and no delayed substitutions.

It is interesting to investigate static contract checking using datatypes in $\lambda^H_{\mathbf{dt}}$. We think that bisimulation techniques [94] would be tractable to show upcast elimination in $\lambda^H_{\mathbf{dt}}$. We expect refining constructor argument types is useful also for static checking [59]. A proof that a generalized version of the translation given in Section 4.3 is correct also remains as future work (although we do have translation). It is worth investigating intersection types (or even Boolean operations) in this setting so that properties on data structures can be easily combined. In addition, it is left as interesting future work to extend $\lambda^H_{\mathbf{dt}}$ with casts between "isomorphic" datatypes, such as lists with cons and ones with "snoc."

Finally, extensions of manifest contracts with computational effects such as states and control operators, concurrency, and probability are open challenges.

Bibliography

[1] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 213–227, 1989.

[2] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[3] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, 1995.

[4] Agda Development Team. The Agda Wiki. URL http://wiki.portal.chalmers.se/agda/pmwiki.php. Accessed on 2016-01-15.

[5] Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In *Proceedings of the 15th European Symposium on Programming*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer-Verlag, 2006.

[6] Amal Ahmed, Derek Dreyer, and Andreas Rossberg. State-dependent representation independence. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages*, pages 340–353, 2009.

[7] Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. Blame for all. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages*, pages 201–214, 2011.

[8] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. In *Proceedings of the 5th Asian Symposium on Programming Languages and Systems*, volume 4807 of *Lecture Notes in Computer Science*, pages 239–254, 2007.

[9] Kenichi Asai and Yukiyoshi Kameyama. Polymorphic delimited continuations. CS-TR-07-10, Department of Computer Science, University of Tsukuba, 2007.

[10] Robert Atkey, Patricia Johann, and Neil Ghani. Refining inductive types. *Logical Methods in Computer Science*, 8(2:9):1–30, 2012.

[11] Lennart Augustsson. Cayenne - a language with dependent types. In *Proceedings of the 3rd ACM International Conference on Functional Programming*, pages 239–250, 1998.

[12] John W. Backus. The history of FORTRAN I, II, and III. *IEEE Annals of the History of Computing*, 20(4):68–78, 1998.

[13] Hendrik Pieter Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., 1992.

[14] João Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C. Pierce. Polymorphic contracts. In *Proceedings of the 20th European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 18–37. Springer-Verlag, 2011.

[15] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th ACM International Conference on Functional Programming*, pages 105–116, 2010.

[16] Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. A dynamic continuation-passing style for dynamic delimited continuations. Research Series RS-06-15, BRICS, DAIMI, 2006.

[17] Matthias Blume and David A. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16(4–5):375–414, 2006.

[18] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for Clojure. Unpublished draft.

[19] Luca Cardelli. A polymorphic λ-calculus with Type:Type. Technical Report 10, DEC Systems Research Center, Palo Alto, CA, 1986.

[20] James Cheney and Ralf Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[21] Olaf Chitil. A semantics for lazy assertions. In *Proceedings of the ACM 2011 Workshop on Partial Evaluation and Program Manipulation*, pages 141–150, 2011.

[22] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *Proceedings of the 27th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 587–606, 2012.

[23] John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proceedings of the 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 320–334. Springer-Verlag, 2001.

[24] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. 89/12, DIKU, University of Copenhagen, 1989.

[25] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

[26] Nicolaas G. de Bruijin. A survey of the project Automath. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 579–606. Academic Press, 1980.

[27] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *Proceedings of the 21st European Symposium on Programming*, volume 7211 of *Lecture Notes in Computer Science*, pages 214–233. Springer-Verlag, 2012.

[28] Tim Disney and Cormac Flanagan. Gradual information flow typing. In *International Workshop on Scripts to Programs*, 2011.

[29] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. In *Proceedings of the 24th IEEE Symposium on Logic in Computer Science*, pages 71–80, 2009.

[30] R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 17(6):687–730, 2007.

[31] Facebook. Hack. URL http://hacklang.org/. Accessed on 2016-01-15.

[32] Matthias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.

[33] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[34] Andrzej Filinski. Representing monads. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, pages 446–457, 1994.

[35] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the 7th ACM International Conference on Functional Programming*, pages 48–59, 2002.

[36] Robert Bruce Findler, Shu-yu Guo, and Anne Rogers. Lazy contract checking for immutable data structures. In *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages*, volume 5083 of *Lecture Notes in Computer Science*, pages 111–128. Springer-Verlag, 2008.

[37] Cormac Flanagan. Hybrid type checking. In *Proceedings of the 33rd ACM Symposium on Principles of Programming Languages*, pages 245–256, 2006.

[38] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. URL http://racket-lang.org/tr1/.

[39] Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding delimited and composable control to a production programming environment. In *Proceedings of the 12th ACM International Conference on Functional Programming*, pages 165–176, 2007.

[40] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM 1991 Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

[41] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

[42] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014. ISBN 978-0-13-390069-9.

[43] Michael Greenberg. *Manifest Contracts*. PhD thesis, University of Pennsylvania, 2013.

[44] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages*, pages 353–364, 2010.

[45] Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Proceedings of the 8th Symposium on Trends in Functional Programming*, pages 54–70, 2007.

[46] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[47] Dan Grossman, Greg Morrisett, and Steve Zdancewic. Syntactic type abstraction. *ACM Transactions on Programming Languages and Systems*, 22(6):1037–1080, 2000.

[48] Arjun Guha, Jacob Matthews, Robert Bruce Findler, and Shriram Krishnamurthi. Relationally-parametric polymorphic contracts. In *Proceedings of the 3rd Dynamic Language Symposium*, pages 29–40, 2007.

[49] Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *Proceedings of the 7th ACM International Conference on Functional Programming Languages and Computer Architecture*, pages 12–23, 1995.

[50] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[51] Fritz Henglein. Dynamic typing. In *Proceedings of the 4th European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 233–253. Springer-Verlag, 1992.

[52] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Proceedings of the 8th Symposium on Trends in Functional Programming*, pages 1–18, 2007.

[53] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[54] Lintaro Ina and Atsushi Igarashi. Gradual typing for generics. In *Proceedings of the 26th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 609–624, 2011.

[55] ISO. Information technology - programming languages - C. Standard ISO/IEC 9899-2012, International Organization for Standardization, Geneva, Switzerland, 2012.

[56] ISO. Information technology – programming languages – C++. Standard ISO/IEC 14882:2014, International Organization for Standardization, Geneva, Switzerland, 2014.

[57] Yukiyoshi Kameyama and Masahito Hasegawa. A sound and complete axiomatization of delimited continuations. In *Proceedings of the 8th ACM International Conference on Functional Programming*, pages 177–188, 2003.

[58] Yukiyoshi Kameyama and Takuo Yonezawa. Typed dynamic control operators for delimited continuations. In *Proceedings of the 9th International Symposium on Functional and Logic Programming*, volume 4989 of *Lecture Notes in Computer Science*, pages 239–254, 2008.

[59] Ming Kawaguchi, Patrick M. Rondon, and Ranjit Jhala. Type-based data structure verification. In *Proceedings of the ACM 2009 Conference on Programming Language Design and Implementation*, pages 304–315, 2009.

[60] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proceedings of the ACM 2004 Workshop on Haskell*, pages 96–107, 2004.

[61] Kenneth Knowles. *Executable Refinement Types*. PhD thesis, University of California, Santa Cruz, 2014.

[62] Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *Proceedings of the 16th European Symposium on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 505–519. Springer-Verlag, 2007.

[63] Kenneth Knowles and Cormac Flanagan. Compositional reasoning and decidable checking for dependent contract types. In *Proceedings of the 3rd ACM Workshop on Programming Languages meets Program Verification*, pages 27–38, 2009.

[64] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Transactions on Programming Languages and Systems*, 32(2:6):1–34, 2010.

[65] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and dynamic (extended report). Technical report, UCSC, 2007.

[66] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 4.02): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, 2014. URL http://caml.inria.fr/distrib/ocaml-4.02/ocaml-4.02-refman.pdf. Accessed on 2016-01-15.

[67] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. In *Proceedings of the 8th ACM International Conference on Functional Programming*, pages 213–225, 2003.

[68] Simon Marlow, editor. *Haskell 2010 Language Report*. 2010.

[69] Jacob Matthews and Amal Ahmed. Parametric polymorphism through run-time sealing or, theorems for low, low prices! In *Proceedings of the 17th European Symposium on Programming*, volume 4960 of *Lecture Notes in Computer Science*, pages 16–31. Springer-Verlag, 2008.

[70] Conor McBride. Ornamental algebras, algebraic ornaments. *Journal of Functional Programming*, 2014. To appear.

[71] Lambert G. L. T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5): 413–424, 1992.

[72] Bertrand Meyer. *Object-Oriented Software Construction, 1st Edition*. Prentice-Hall, 1988. ISBN 0-13-629031-0.

[73] Microsoft Corporation. C♯ language specification, . URL https://msdn.microsoft.com/en-us/library/ms228593.aspx. Accessed on 2016-01-15.

[74] Microsoft Corporation. TypeScript language specification, . URL http://www.typescriptlang.org/Content/TypeScript%20Language%20Specification.pdf. Accessed on 2016-01-15.

[75] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[76] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. The MIT Press, 1997. ISBN 0262631814.

[77] John C. Mitchell. Representation independence and data abstraction. In *ACM Symposium on Principles of Programming Languages*, pages 263–276, 1986. doi: 10.1145/512644.512669.

[78] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 37–51, 1985.

[79] James H. Morris Jr. Types are not sets. In *Proceedings of the 1st ACM Symposium on Principles of Programming Languages*, pages 120–124, 1973.

[80] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. In *Proceedings of the 11th ACM International Conference on Functional Programming*, pages 62–73, 2006.

[81] Georg Neis, Derek Dreyer, and Andreas Rossberg. Non-parametric parametricity. In *Proceedings of the 14th ACM International Conference on Functional Programming*, pages 135–148, 2009.

[82] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *Proceedings of the 3rd International Conference on Theoretical Computer Science*, pages 437–450, 2004.

[83] David Lorge Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

[84] Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *Proceedings of the 1st International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.

[85] Benjamin Pierce and Eijiro Sumii. Relating cryptography and polymorphism, 2000.

[86] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002. ISBN 0-262-16209-1.

[87] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10(3):321–359, 2000.

[88] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM 1972 Annual Conference*, pages 717–740, 1972.

[89] John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, pages 408–423, 1974.

[90] John C. Reynolds. Types, abstraction, and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[91] Patrick M. Rondon, Ming Kawaguchi, , and Ranjit Jhala. Liquid types. In *Proceedings of the ACM 2008 Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

[92] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *Proceedings of the 5th ACM International Conference on Principles and Practice of Declarative Programming*, pages 241–252, 2003.

[93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

[94] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. Environmental bisimulations for higher-order languages. *ACM Transactions on Programming Languages and Systems*, 33(1:5):1–69, 2011.

[95] Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. A theory of gradual effect systems. In *Proceedings of the 19th ACM International Conference on Functional Programming*, pages 283–295, 2014.

[96] Taro Sekiyama and Atsushi Igarashi. Logical relations for a manifest contract calculus, fixed. `http://hope2012.mpi-sws.org/`, September 2012. Talk abstract and slides.

[97] Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. Polymorphic manifest contracts, revised and resolved. *ACM Transactions on Programming Languages and Systems*, 2015. Accepted with major revision.

[98] Taro Sekiyama, Yuki Nishida, and Atsushi Igarashi. Manifest contracts for datatypes. In *Proceedings of the 42nd ACM Symposium on Principles of Programming Languages*, pages 195–207, 2015.

[99] Taro Sekiyama, Soichiro Ueda, and Atsushi Igarashi. Shifting the blame - A blame calculus with delimited control. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems*, volume 9458 of *Lecture Notes in Computer Science*, pages 189–207. Springer-Verlag, 2015.

[100] Chung-chieh Shan. Shift to control. In *Scheme and Functional Programming Workshop*, pages 99–107, 2004.

[101] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[102] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer-Verlag, 2007.

[103] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages*, pages 365–376, 2010.

[104] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *Proceedings of the 1st Summit on Advances in Programming Languages*, pages 274–293, 2015.

[105] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. Monotonic references for efficient gradual typing. In *Proceedings of the 24th European Symposium on Programming*, volume 9032 of *Lecture Notes in Computer Science*, pages 432–456. Springer-Verlag, 2015.

[106] Dorai Sitaram. Handling control. In *Proceedings of the ACM 1993 Conference on Programming Language Design and Implementation*, pages 147–155, 1993.

[107] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. *Journal of Computer Security*, 11(4):521–554, 2003.

[108] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM International Conference on Functional Programming*, pages 266–278, 2011.

[109] Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Gradual typing for first-class classes. In *Proceedings of the 27th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 793–810, 2012.

[110] Asumu Takikawa, T. Stephen Strickland, and Sam Tobin-Hochstadt. Constraining delimited control with contracts. In *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 229–248. Springer-Verlag, 2013.

[111] Éric Tanter and Nicolas Tabareau. Gradual certified programming in Coq. In *Proceedings of the 11th Dynamic Language Symposium*, pages 26–40, 2015.

[112] The Coq Development Team. The Coq proof assistant. URL http://coq.inria.fr/. Accessed on 2016-01-15.

[113] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *Proceedings of the 2nd Dynamic Language Symposium*, pages 964–974, 2006.

[114] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages*, pages 395–406, 2008.

[115] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the 2nd International Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 1985.

[116] Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. Abstract refinement types. In *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 209–228. Springer-Verlag, 2013.

[117] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM International Conference on Functional Programming*, pages 269–282, 2014.

[118] Michael M. Vitousek, Andrew M. Kent, Jeremy G. Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *Proceedings of the 10th Dynamic Language Symposium*, pages 45–56, 2014.

[119] Philip Wadler. Theorems for free! In *4th ACM International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, 1989.

[120] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming*, volume 5502 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2009.

[121] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[122] Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.

[123] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 214–227, 1999.

[124] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, pages 224–235, 2003.

[125] Dana N. Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the ACM 2012 Workshop on Partial Evaluation and Program Manipulation*, pages 107–116, 2012.

Appendix A

Proofs of Gradual Typing with Delimited Control

This chapter gives the proofs of properties stated in Chapter 2. Section A.1 proves Progress (Lemma 1) and Preservation (Lemma 2), which imply Type Soundness (Theorem 1). Section A.2 presents the proof of Blame Theorem and Subtype Theorem (Theorem 2). Finally, in Section A.3, we show soundness of our CPS transformation, that is, that the transformation preserves well-typedness (Theorem 3) and term equality (Theorem 4).

As discussed in Section 2.4, ground terms and values have blame labels as subscripts; we omit the labels if they are not important or clear from the context. We write $\mathcal{K}_\iota$ to denote the set of constants of ι.

A.1 Type Soundness

We prove type soundness (Theorem 1) in a usual syntactic manner, that is, via progress (Lemma 1) and preservation (Lemma 2). We start with showing standard lemmas: weakening lemma (Lemma A.1.1), substitution lemma (Lemma A.1.5), inversion lemmas (Lemmas A.1.6—A.1.11), and canonical forms lemma (Lemma A.1.12). Lemma A.1.13 shows that, for any nondynamic type A, there exists a unique ground type G compatible with A. Using these lemmas, we show progress (Lemma 1).

Lemma A.1.1 (Weakening). *If* $\Gamma; \alpha \vdash s : A; \beta$ *and* x *is a fresh variable, then* $\Gamma, x{:}B; \alpha \vdash s : A; \beta$ *for any type* B.

Proof. Straightforward by induction on the typing derivation. □

Lemma A.1.2 (Strenghtening). *If* $\Gamma, x{:}A; \alpha \vdash s : B; \beta$ *and* $x \notin fv(s)$, *then* $\Gamma; \alpha \vdash s : B; \beta$.

Proof. Straightforward by induction on the typing derivation. □

Lemma A.1.3. *If* $\Gamma; \alpha \vdash v : A; \beta$, *then* $\alpha = \beta$.

Proof. Straightforward by induction on the typing derivation. □

Lemma A.1.4. *If* $\Gamma; \alpha \vdash v : A; \beta$, *then* $\Gamma; \gamma \vdash v : A; \gamma$ *for any type* γ.

Proof. Straightforward by induction on the typing derivation. □

Lemma A.1.5 (Substitution). *If* $\Gamma; \alpha \vdash v : A; \alpha$ *and* $\Gamma, x{:}A; \beta \vdash s : B; \gamma$, *then* $\Gamma; \beta \vdash s[x := v] : B; \gamma$.

Proof. Straightforward by induction on the typing derivation of s. Note that, in the case for (T_VAR), we have $\Gamma; \beta \vdash v : A; \beta$ by Lemma A.1.3. □

Lemma A.1.6 (Lambda Inversion). *If* $\Gamma; \alpha \vdash \lambda x.\, s : A/\gamma \to B/\delta; \beta$, *then* $\Gamma, x{:}A; \gamma \vdash s : B; \delta$.

Proof. By case analysis on the typing rule applied last. □

Lemma A.1.7 (Ground Inversion). *If* $\Gamma; \alpha \vdash v : G \Rightarrow \star : \star; \beta$, *then* $\Gamma; \alpha \vdash v : G; \beta$.

Proof. By case analysis on the typing rule applied last. □

Lemma A.1.8 (Shift Inversion). *If* $\Gamma; \alpha \vdash \mathcal{S}k.\, s : A; \beta$, *then* $\Gamma, k{:}A/\gamma \to \alpha/\gamma; \delta \vdash s : \delta; \beta$ *for some* γ *and* δ.

Proof. By case analysis on the typing rule applied last. □

Lemma A.1.9 (Application Inversion). *If* $\Gamma; \alpha \vdash s\ t : B; \delta$, *then* $\Gamma; \gamma \vdash s : A/\alpha \to B/\beta; \delta$ *and* $\Gamma; \beta \vdash t : A; \gamma$ *for some* A, β, γ, *and* δ.

Proof. By case analysis on the typing rule applied last. □

Lemma A.1.10 (Reset Inversion). *If* $\Gamma; \alpha \vdash \langle s \rangle : A; \beta$, *then* $\Gamma; \gamma \vdash s : \gamma; A$ *for some* γ, *and* $\alpha = \beta$.

Proof. By case analysis on the typing rule applied last. □

Lemma A.1.11 (Variable Inversion). *If* $\Gamma; \alpha \vdash x : A; \beta$, *then* $x{:}A \in \Gamma$.

Proof. By case analysis on the typing rule applied last. □

Lemma A.1.12 (Canonical Forms). *Suppose that* $\emptyset; \alpha \vdash v : A; \beta$.

(1) If $A = \iota$, *then* $v = c \in \mathcal{K}_\iota$.

(2) If $A = A'/\alpha' \to B'/\beta'$, *then* $v = \lambda x.\, s$ *for some* s *and* s.

(3) If $A = \star$, *then* $v = v' : G \Rightarrow \star$ *for some* v' *and* G.

Proof. By case analysis on the typing rule applied last to v. □

Lemma A.1.13 (Unique Ground Type). *For any type* $A \neq \star$, *there exists an unique ground type* G *such that* $A \sim G$.

Proof. Straightforward by case analysis on A. □

Lemma 1 (Progress). *If* $\emptyset; \alpha \vdash s : A; \beta$, *then one of the followings holds:*

- $s \longmapsto s'$ *for some* s';
- s *is a value;*
- $s = \mathsf{blame}\, p$ *for some* p; *or*
- $s = F[\mathcal{S}k.\, t]$ *for some* F, k *and* t.

Proof. By induction on the typing derivation.

Case (T_CONST), (T_ABS), (T_BLAME), (T_SHIFT): Obvious.

Case (T_VAR): Contradictory.

Case (T_OP): We are given $\emptyset; \alpha \vdash op(\overline{t_i}^{i \in \{1,\ldots,n\}}) : A; \beta$ for some op and $\overline{t_i}^i$. By inversion, we have $ty(op) = \overline{\iota_i}^i \to A$ and $\emptyset; \gamma_i \vdash t_i : \iota_i; \gamma_{i-1}$ where $\alpha = \gamma_n$ and $\beta = \gamma_0$. If all terms $\overline{t_i}^i$ are values, then we finish by Lemma A.1.12 (1) and (R_OP). Otherwise, suppose that $t_1, \ldots, t_{j-1}$ are values and t_j is not for some j. By case analysis on t_j with the IH.

- Case $t_j \longmapsto u$: By (E_STEP) or (E_ABORT).
- Case $t_j = \mathsf{blame}\, p$: By (E_ABORT).
- Case $t_j = F[\mathcal{S}k.\, u]$: We finish.

Case (T_APP): We are given $\emptyset; \alpha \vdash t\, u : A; \beta$ for some t and u. By inversion, we have $\emptyset; \gamma \vdash t : B/\alpha \to A/\beta'; \beta$ and $\emptyset; \beta' \vdash u : B; \gamma$ for some B, β', and γ. If t or u is not a value, then we finish similarly to the case for (T_OP). Otherwise, suppose that t and u are values. By Lemma A.1.12 (2), $t = \lambda x.\, t'$ for some x and t'. Thus, we finish by (R_BETA).

Case (T_CAST): We are given $\emptyset; \alpha \vdash t : B \Rightarrow^p A : A; \beta$ for some t and B. By inversion, we have $\emptyset; \alpha \vdash t : B; \beta$ and $B \sim A$. If t is not a value, then we finish similarly to the case for (T_OP). Otherwise, if t is a value, we proceed by case analysis on $B \sim A$.

- Case (C_DYNTO): We are given $A = \star$. If $B = \star$, then we finish by (R_DYN). Otherwise, we finish by Lemma A.1.13 and (R_GROUND).
- Case (C_DYNFROM): If $A = \star$, then we finish by (R_DYN). Otherwise, by Lemma A.1.12 (3), and (R_COLLAPSE) or (R_CONFLICT).
- Case (C_BASE): By (R_BASE).
- Case (C_FUN): By (R_WRAP).

Case (T_GROUND): We are given $\emptyset; \alpha \vdash t : G \Rightarrow \star : \star; \beta$ for some t and G. By inversion, we have $\emptyset; \alpha \vdash t : G; \beta$. If t is not a value, then we finish similarly to the case for (T_OP). Otherwise, if t is a value, then so is $t : G \Rightarrow \star$.

Case (T_IS): We are given $\emptyset; \alpha \vdash t\,\mathsf{is}\, G : \mathsf{bool}; \beta$ for some t and G. By inversion, we have $\emptyset; \alpha \vdash t : \star; \beta$. If t is not a value, then we finish similarly to the case for (T_OP). Otherwise, if t is a value, then $t = v : H \Rightarrow \star$ for some v and H by Lemma A.1.12 (3). We finish by (R_ISTRUE) or (R_ISFALSE).

Case (T_RESET): We are given $\emptyset; \alpha \vdash \langle t \rangle : A; \alpha$ for some t. By inversion, we have $\emptyset; \gamma \vdash t : \gamma; A$ for some γ. If t takes a step or is blamed, then we finish similarly to the case for (T_OP). If t is a value, then we finish by (R_RESET). Otherwise, we finish by (R_SHIFT).

□

Next, we show preservation (Lemma 2). Our proof in the case for (R_SHIFT) follows Asai and Kameyama [9], which shows preservation of a polymorphic lambda calculus with shift/reset by introducing a "decomposed" version of the reduction rule for shift. Their proof rests on the facts that a reduction by shift can be decomposed into reductions in the decomposed version and reductions in it preserve well-typedness. We show these in Lemmas A.1.17 and A.1.15, respectively, and then prove preservation. We give decomposed reduction $\twoheadrightarrow$ of shift reduction in Definition 10.

Lemma A.1.14. *Let x be a variable and F be a pure evaluation context such that $x \notin fv(F)$. If $\Gamma, x{:}A; \alpha \vdash F[x] : B; \beta$ and $\Gamma; \beta \vdash s : A; \gamma$, then $\Gamma; \alpha \vdash F[s] : B; \gamma$.*

Proof. By induction on the typing derivation of $F[x]$.

Case (T_CONST), (T_ABS), (T_BLAME), (T_SHIFT), and (T_RESET): Contradictory.

Case (T_VAR): By Lemma A.1.3.

Case (T_OP): We are given $\Gamma, x{:}A; \alpha \vdash op(\overline{v_i}^i, F'[x], \overline{t_j}^j) : B; \beta$ for some op, $\overline{v_i}^i$, F', and $\overline{t_j}^j$. By inversion and Lemma A.1.3, we have $ty(op) = \overline{\iota_i}^i \to \iota' \to \overline{\iota''_j}^j \to B$ and $\overline{\Gamma, x{:}A; \beta \vdash v_i : \iota_i; \beta}^i$ and $\Gamma, x{:}A; \gamma_0 \vdash F'[x] : \iota'; \beta$ and $\overline{\Gamma, x{:}A; \gamma'_j \vdash t_j : \iota''_j; \gamma'_{j-1}}^j$ and $\gamma'_n = \alpha$ (where we assume that $\overline{t_j}^j = t_1, ..., t_n$).

By the IH, $\Gamma; \gamma_0 \vdash F'[s] : \iota'; \gamma$. By Lemmas A.1.2 and A.1.4, $\overline{\Gamma; \gamma \vdash v_i : \iota_i; \gamma}^i$ and $\overline{\Gamma; \gamma'_j \vdash t_j : \iota''_j; \gamma'_{j-1}}^j$. Thus, by (T_OP), we finish.

Case (T_APP): By case analysis on F.

- Case $F = F'\ t$: By inversion, we have $\Gamma, x{:}A; \gamma' \vdash F'[x] : A'/\alpha \to B/\beta'; \beta$ and $\Gamma, x{:}A; \beta' \vdash t : A'; \gamma'$ for some A', β', and γ'. By the IH, Lemma A.1.2, and (T_APP), we finish.
- Case $F = v\ F'$: By inversion, we have $\Gamma, x{:}A; \gamma' \vdash v : A'/\alpha \to B/\beta'; \beta$ and $\Gamma, x{:}A; \beta' \vdash F'[x] : A'; \gamma'$ for some A', β', and γ'. By the IH, Lemmas A.1.2 and A.1.4, and (T_APP), we finish.

Case (T_CAST), (T_GROUND), and (T_IS): By the IH. □

Definition 10. *The relation $\twoheadrightarrow$ is the least contextual relation that contains the following rules:*

$$
\begin{array}{lcl}
op(\overline{v_i}^i, \mathcal{S}k.\ s, \overline{t_j}^j) & \twoheadrightarrow & \mathcal{S}k'.\ s\ [k := \lambda x.\ \langle k'\ op(\overline{v_i}^i, x, \overline{t_j}^j)\rangle] \\
(\mathcal{S}k.\ s)\ t & \twoheadrightarrow & \mathcal{S}k'.\ s\ [k := \lambda x.\ \langle k'\ (x\ t)\rangle] \\
v\ (\mathcal{S}k.\ s) & \twoheadrightarrow & \mathcal{S}k'.\ s\ [k := \lambda x.\ \langle k'\ (v\ x)\rangle] \\
(\mathcal{S}k.\ s) : A \Rightarrow^p B & \twoheadrightarrow & \mathcal{S}k'.\ s\ [k := \lambda x.\ \langle k'\ (x : A \Rightarrow^p B)\rangle] \\
(\mathcal{S}k.\ s) : G \Rightarrow \star & \twoheadrightarrow & \mathcal{S}k'.\ s\ [k := \lambda x.\ \langle k'\ (x : G \Rightarrow \star)\rangle] \\
(\mathcal{S}k.\ s)\ \mathsf{is}\ G & \twoheadrightarrow & \mathcal{S}k'.\ s\ [k := \lambda x.\ \langle k'\ (x\ \mathsf{is}\ G)\rangle] \\
\langle \mathcal{S}k.\ s\rangle & \twoheadrightarrow & \langle s\ [k := \lambda x.\ \langle x\rangle]\rangle \\
\langle (\lambda x.\ \langle F[x]\rangle)\ s\rangle & \twoheadrightarrow & \langle F[s]\rangle
\end{array}
$$

where x is a fresh variable. We write $\twoheadrightarrow^*$ to denote the transitive closure of $\twoheadrightarrow$.

Lemma A.1.15. *If $\Gamma; \alpha \vdash s : A; \beta$ and $s \twoheadrightarrow t$, then $\Gamma; \alpha \vdash t : A; \beta$.*

Proof. By induction on the typing derivation. We mention only the cases where rules in Definition 10 are applied.

Case (T_OP): We are given $op(\overline{v_i}^i, \mathcal{S}k.\ t, \overline{t_j}^j) \twoheadrightarrow \mathcal{S}k'.\ t\ [k := \lambda x.\ \langle k'\ op(\overline{v_i}^i, x, \overline{t_j}^j)\rangle]$. By inversion and Lemma A.1.3, we have $ty(op) = \overline{\iota_i}^i \to \iota' \to \overline{\iota''_j}^j \to A$ and $\overline{\Gamma; \beta \vdash v_i : \iota_i; \beta}^i$ and $\Gamma; \gamma_0 \vdash \mathcal{S}k.\ t : \iota'; \beta$ and $\overline{\Gamma; \gamma_j \vdash t_j : \iota''_j; \gamma_{j-1}}^j$ and $\gamma_n = \alpha$ (note that here we assume that $\overline{t_j}^j = t_0, ..., t_n$).

By Lemmas A.1.4 and A.1.1, $\overline{\Gamma, x{:}\iota'; \gamma_0 \vdash v_i \,:\, \iota_i; \gamma_0}^{\,i}$, and by (T_VAR), $\Gamma, x{:}\iota'; \gamma_0 \vdash x \,:\, \iota'; \gamma_0$. Thus, by Lemma A.1.1 and (T_OP), $\Gamma, x{:}\iota'; \alpha \vdash op(\overline{v_i}^{\,i}, x, \overline{t_j}^{\,j}) \,:\, A; \gamma_0$. By Lemma A.1.1, (T_VAR), and (T_APP),

$$\Gamma, k'{:}A/\alpha \to \alpha/\alpha, x{:}\iota'; \alpha \vdash k'\; op(\overline{v_i}^{\,i}, x, \overline{t_j}^{\,j}) \,:\, \alpha; \gamma_0.$$

Here, by Lemma A.1.8, $\Gamma, k{:}\iota'/\gamma' \to \gamma_0/\gamma'; \delta' \vdash t \,:\, \delta'; \beta$ for some γ' and δ'. Since, by (T_RESET) and (T_ABS), $\Gamma, k'{:}A/\alpha \to \alpha/\alpha; \alpha \vdash \lambda x.\, \langle k'\; op(\overline{v_i}^{\,i}, x, \overline{t_j}^{\,j})\rangle \,:\, \iota'/\gamma' \to \gamma_0/\gamma'; \alpha$, we have

$$\Gamma, k'{:}A/\alpha \to \alpha/\alpha; \delta' \vdash t\,[k := \lambda x.\, \langle k'\; op(\overline{v_i}^{\,i}, x, \overline{t_j}^{\,j})\rangle] \,:\, \delta'; \beta$$

by Lemmas A.1.1 and A.1.5. By (T_SHIFT), $\Gamma; \alpha \vdash \mathcal{S}k'.\, t\,[k := \lambda x.\, \langle k'\; op(\overline{v_i}^{\,i}, x, \overline{t_j}^{\,j})\rangle] \,:\, A; \beta$.

Case (T_APP): By case analysis on the rule applied.

Case $(\mathcal{S}k.\, t)\; u \twoheadrightarrow \mathcal{S}k'.\, t\,[k := \lambda x.\, \langle k'\; (x\; u)\rangle]$: By inversion, we have $\Gamma; \gamma \vdash \mathcal{S}k.\, t \,:\, B/\alpha \to A/\beta'; \beta$ and $\Gamma; \beta' \vdash u \,:\, B; \gamma$ for some B, β', and γ.
By Lemma A.1.1, (T_VAR), and (T_APP), $\Gamma, x{:}B/\alpha \to A/\beta'; \alpha \vdash x\; u \,:\, B; \gamma$. Again, by Lemma A.1.1, (T_VAR), and (T_APP),

$$\Gamma, k'{:}A/\alpha \to \alpha/\alpha, x{:}B/\alpha \to A/\beta'; \alpha \vdash k'\; (x\; u) \,:\, \alpha; \gamma.$$

Here, by Lemma A.1.8, $\Gamma, k{:}(B/\alpha \to A/\beta')/\gamma' \to \gamma/\gamma'; \delta' \vdash t \,:\, \delta'; \beta$ for some γ' and δ'. Since, by (T_RESET) and (T_ABS), $\Gamma, k'{:}A/\alpha \to \alpha/\alpha; \alpha \vdash \lambda x.\, \langle k'\; (x\; u)\rangle \,:\, (B/\alpha \to A/\beta')/\gamma' \to \gamma/\gamma'; \alpha$, we have

$$\Gamma, k'{:}A/\alpha \to \alpha/\alpha; \delta' \vdash t\,[k := \lambda x.\, \langle k'\; (x\; u)\rangle] \,:\, \delta'; \beta$$

by Lemmas A.1.1 and A.1.5. By (T_SHIFT), $\Gamma; \alpha \vdash \mathcal{S}k'.\, t\,[k := \lambda x.\, \langle k'\; (x\; u)\rangle] \,:\, A; \beta$.

Case $v\; (\mathcal{S}k.\, t) \twoheadrightarrow \mathcal{S}k'.\, t\,[k := \lambda x.\, \langle k'\; (v\; x)\rangle]$: By inversion and Lemma A.1.3, we have $\Gamma; \beta \vdash v \,:\, B/\alpha \to A/\beta'; \beta$ and $\Gamma; \beta' \vdash \mathcal{S}k.\, t \,:\, B; \beta$ for some B and β'.
By Lemma A.1.1, (T_VAR), and (T_APP), $\Gamma, x{:}B; \alpha \vdash v\; x \,:\, A; \beta'$. Again, by Lemma A.1.1, (T_VAR), and (T_APP),

$$\Gamma, k'{:}A/\alpha \to \alpha/\alpha, x{:}B; \alpha \vdash k'\; (v\; x) \,:\, \alpha; \beta'.$$

Here, by Lemma A.1.8, $\Gamma, k{:}B/\gamma' \to \beta'/\gamma'; \delta' \vdash t \,:\, \delta'; \beta$ for some γ' and δ'. Since, by (T_RESET) and (T_ABS), $\Gamma, k'{:}A/\alpha \to \alpha/\alpha; \alpha \vdash \lambda x.\, \langle k'\; (v\; x)\rangle \,:\, B/\gamma' \to \beta'/\gamma'; \alpha$, we have

$$\Gamma, k'{:}A/\alpha \to \alpha/\alpha; \delta' \vdash t\,[k := \lambda x.\, \langle k'\; (v\; x)\rangle] \,:\, \delta'; \beta$$

by Lemmas A.1.1 and A.1.5. By (T_SHIFT), $\Gamma; \alpha \vdash \mathcal{S}k'.\, t\,[k := \lambda x.\, \langle k'\; (v\; x)\rangle] \,:\, A; \beta$.

Case (T_CAST): We are given $(\mathcal{S}k.\, t) \,:\, B \Rightarrow^p A \twoheadrightarrow \mathcal{S}k'.\, t\,[k := \lambda x.\, \langle k'\; (x \,:\, B \Rightarrow^p A)\rangle]$. By inversion, we have $\Gamma; \alpha \vdash \mathcal{S}k.\, t \,:\, B; \beta$.

By (T_VAR), and (T_CAST), $\Gamma, x{:}B; \alpha \vdash x \ : \ B \Rightarrow^p \ A \ : \ A; \alpha$. By Lemma A.1.1, (T_VAR), and (T_APP),

$$\Gamma, k'{:}A/\alpha \to \alpha/\alpha, x{:}B; \alpha \vdash k' \, (x \ : \ B \Rightarrow^p \ A) \ : \ \alpha; \alpha.$$

Here, by Lemma A.1.8, $\Gamma, k{:}B/\gamma' \to \alpha/\gamma'; \delta' \vdash t \ : \ \delta'; \beta$ for some γ' and δ'. Since, by (T_RESET) and (T_ABS), $\Gamma, k'{:}A/\alpha \to \alpha/\alpha; \alpha \vdash \lambda x. \langle k' \, (x \ : \ B \Rightarrow^p \ A) \rangle \ : \ B/\gamma' \to \alpha/\gamma'; \alpha$, we have

$$\Gamma, k'{:}A/\alpha \to \alpha/\alpha; \delta' \vdash t \, [k := \lambda x. \langle k' \, (x \ : \ B \Rightarrow^p \ A) \rangle] \ : \ \delta'; \beta$$

by Lemmas A.1.1 and A.1.5. By (T_SHIFT), $\Gamma; \alpha \vdash \mathcal{S}k'. \, t \, [k := \lambda x. \langle k' \, (x \ : \ B \Rightarrow^p A) \rangle] \ : \ A; \beta$.

Case (T_GROUND): We are given $(\mathcal{S}k. \, t) \ : \ G \Rightarrow \star \twoheadrightarrow \mathcal{S}k'. \, t \, [k := \lambda x. \langle k' \, (x \ : \ G \Rightarrow \star) \rangle]$. By inversion, we have $\Gamma; \alpha \vdash \mathcal{S}k. \, t \ : \ G; \beta$. Note that $A = \star$.

By (T_VAR), and (T_CAST), $\Gamma, x{:}G; \alpha \vdash x \ : \ G \Rightarrow \star \ : \ \star; \alpha$. By Lemma A.1.1, (T_VAR), and (T_APP),

$$\Gamma, k'{:}\star/\alpha \to \alpha/\alpha, x{:}G; \alpha \vdash k' \, (x \ : \ G \Rightarrow \star) \ : \ \alpha; \alpha.$$

Here, by Lemma A.1.8, $\Gamma, k{:}G/\gamma' \to \alpha/\gamma'; \delta' \vdash t \ : \ \delta'; \beta$ for some γ' and δ'. Since, by (T_RESET) and (T_ABS), $\Gamma, k'{:}\star/\alpha \to \alpha/\alpha; \alpha \vdash \lambda x. \langle k' \, (x \ : \ G \Rightarrow \star) \rangle \ : \ G/\gamma' \to \alpha/\gamma'; \alpha$, we have

$$\Gamma, k'{:}\star/\alpha \to \alpha/\alpha; \delta' \vdash t \, [k := \lambda x. \langle k' \, (x \ : \ G \Rightarrow \star) \rangle] \ : \ \delta'; \beta$$

by Lemmas A.1.1 and A.1.5. By (T_SHIFT), $\Gamma; \alpha \vdash \mathcal{S}k'. \, t \, [k := \lambda x. \langle k' \, (x \ : \ G \Rightarrow \star) \rangle] \ : \ \star; \beta$.

Case (T_IS): We are given $(\mathcal{S}k. \, t) \, \mathsf{is} \, G \twoheadrightarrow \mathcal{S}k'. \, t \, [k := \lambda x. \langle k' \, (x \, \mathsf{is} \, G) \rangle]$. By inversion, we have $\Gamma; \alpha \vdash \mathcal{S}k. \, t \ : \ \star; \beta$. Note that $A = \mathsf{bool}$.

By (T_VAR), and (T_CAST), $\Gamma, x{:}\star; \alpha \vdash x \, \mathsf{is} \, G \ : \ \mathsf{bool}; \alpha$. By Lemma A.1.1, (T_VAR), and (T_APP),

$$\Gamma, k'{:}\mathsf{bool}/\alpha \to \alpha/\alpha, x{:}G; \alpha \vdash k' \, (x \, \mathsf{is} \, G) \ : \ \alpha; \alpha.$$

Here, by Lemma A.1.8, $\Gamma, k{:}\star/\gamma' \to \alpha/\gamma'; \delta' \vdash t \ : \ \delta'; \beta$ for some γ' and δ'. Since, by (T_RESET) and (T_ABS), $\Gamma, k'{:}\mathsf{bool}/\alpha \to \alpha/\alpha; \alpha \vdash \lambda x. \langle k' \, (x \, \mathsf{is} \, G) \rangle \ : \ \star/\gamma' \to \alpha/\gamma'; \alpha$, we have

$$\Gamma, k'{:}\mathsf{bool}/\alpha \to \alpha/\alpha; \delta' \vdash t \, [k := \lambda x. \langle k' \, (x \, \mathsf{is} \, G) \rangle] \ : \ \delta'; \beta$$

by Lemmas A.1.1 and A.1.5. By (T_SHIFT), $\Gamma; \alpha \vdash \mathcal{S}k'. \, t \, [k := \lambda x. \langle k' \, (x \, \mathsf{is} \, G) \rangle] \ : \ \mathsf{bool}; \beta$.

Case (T_RESET): By case analysis on the rule applied.

Case $\langle \mathcal{S}k. \, t \rangle \twoheadrightarrow \langle t \, [k := \lambda x. \langle x \rangle] \rangle$: By inversion, we have $\Gamma; \gamma \vdash \mathcal{S}k. \, t \ : \ \gamma; A$ for some γ. By Lemma A.1.8, $\Gamma, k{:}\gamma/\gamma' \to \gamma/\gamma'; \delta' \vdash t \ : \ \delta'; A$ for some γ', δ'. Since $\Gamma; \alpha \vdash \lambda x. \langle x \rangle \ : \ \gamma/\gamma' \to \gamma/\gamma'; \alpha$ by (T_VAR) and (T_ABS), we have

$$\Gamma; \delta' \vdash t \, [k := \lambda x. \langle x \rangle] \ : \ \delta'; A$$

by Lemma A.1.5. By (T_RESET), $\Gamma; \alpha \vdash \langle t\,[k := \lambda x.\,\langle x\rangle]\rangle : A; \beta$.

Case $\langle(\lambda x.\,\langle F[x]\rangle)\,t\rangle \twoheadrightarrow \langle F[t]\rangle$: By inversion, we have $\Gamma; \gamma \vdash (\lambda x.\,\langle F[x]\rangle)\,t : \gamma; A$ for some γ, and $\alpha = \beta$. By Lemma A.1.9, $\Gamma; \gamma' \vdash \lambda x.\,\langle F[x]\rangle : B'/\gamma \to \gamma/\beta'; A$ and $\Gamma; \beta' \vdash t : B'; \gamma'$ for some B', β', and γ'. By Lemma A.1.3, $\gamma' = A$. By Lemmas A.1.6 and A.1.10, $\beta' = \gamma$ and $\Gamma, x{:}B'; \gamma'' \vdash F[x] : \gamma''; \gamma$ for some γ''. Since $\Gamma; \gamma \vdash t : B'; A$, we have $\Gamma; \gamma'' \vdash F[t] : \gamma''; A$ by Lemma A.1.14. By (T_RESET), $\Gamma; \alpha \vdash \langle F[t]\rangle : A; \beta$ (note that $\alpha = \beta$). □

Lemma A.1.16. *If $F \neq [\,]$, then $F[\mathcal{S}k.\,s] \twoheadrightarrow^* \mathcal{S}k'.\,s\,[k := \lambda x.\,\langle k'\,F[x]\rangle]$ where $x \notin fv(F)$.*

Proof. By structural induction on F.

Case $F = [\,]$: Contradictory.

Case $F = op(\overline{v_i}^i, F', \overline{t_j}^j)$: If $F' = [\,]$, then obvious; otherwise,

$$\begin{aligned} op(\overline{v_i}^i, F'[\mathcal{S}k.\,s], \overline{t_j}^j) &\twoheadrightarrow^* op(\overline{v_i}^i, \mathcal{S}k'.\,s\,[k := \lambda x.\,\langle k'\,F'[x]\rangle], \overline{t_j}^j) \quad \text{(by the IH)} \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle(\lambda y.\,\langle k''\,op(\overline{v_i}^i, y, \overline{t_j}^j)\rangle)\,F'[x]\rangle] \\ &\twoheadrightarrow^* \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle k''\,op(\overline{v_i}^i, F'[x], \overline{t_j}^j)\rangle]. \end{aligned}$$

Case $F = F'\,t$: If $F' = [\,]$, then obvious; otherwise,

$$\begin{aligned} F'[\mathcal{S}k.\,s]\,t &\twoheadrightarrow^* (\mathcal{S}k'.\,s\,[k := \lambda x.\,\langle k'\,F'[x]\rangle])\,t \quad \text{(by the IH)} \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle(\lambda y.\,\langle k''\,(y\,t)\rangle)\,F'[x]\rangle] \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle k''\,(F'[x]\,t)\rangle]. \end{aligned}$$

Case $F = v\,F'$: If $F' = [\,]$, then obvious; otherwise,

$$\begin{aligned} v\,F'[\mathcal{S}k.\,s] &\twoheadrightarrow^* v\,(\mathcal{S}k'.\,s\,[k := \lambda x.\,\langle k'\,F'[x]\rangle]) \quad \text{(by the IH)} \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle(\lambda y.\,\langle k''\,(v\,y)\rangle)\,F'[x]\rangle] \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle k''\,(v\,F'[x])\rangle]. \end{aligned}$$

Case $F = F' : A \Rightarrow^p B$: If $F' = [\,]$, then obvious; otherwise,

$$\begin{aligned} F'[\mathcal{S}k.\,s] : A \Rightarrow^p B &\twoheadrightarrow^* (\mathcal{S}k'.\,s\,[k := \lambda x.\,\langle k'\,F'[x]\rangle]) : A \Rightarrow^p B \quad \text{(by the IH)} \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle(\lambda y.\,\langle k''\,(y : A \Rightarrow^p B)\rangle)\,F'[x]\rangle] \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle k''\,(F'[x] : A \Rightarrow^p B)\rangle]. \end{aligned}$$

Case $F = F' : G \Rightarrow \star$: If $F' = [\,]$, then obvious; otherwise,

$$\begin{aligned} F'[\mathcal{S}k.\,s] : G \Rightarrow \star &\twoheadrightarrow^* (\mathcal{S}k'.\,s\,[k := \lambda x.\,\langle k'\,F'[x]\rangle]) : G \Rightarrow \star \quad \text{(by the IH)} \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle(\lambda y.\,\langle k''\,(y : G \Rightarrow \star)\rangle)\,F'[x]\rangle] \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle k''\,(F'[x] : G \Rightarrow \star)\rangle]. \end{aligned}$$

Case $F = F'$ is G: If $F' = [\,]$, then obvious; otherwise,

$$\begin{aligned} F'[\mathcal{S}k.\,s] \text{ is } G &\twoheadrightarrow^* (\mathcal{S}k'.\,s\,[k := \lambda x.\,\langle k'\,F'[x]\rangle]) \text{ is } G \quad \text{(by the IH)} \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle(\lambda y.\,\langle k''\,(y \text{ is } G)\rangle)\,F'[x]\rangle] \\ &\twoheadrightarrow \mathcal{S}k''.\,s\,[k := \lambda x.\,\langle k''\,(F'[x] \text{ is } G)\rangle]. \end{aligned}$$

□

Lemma A.1.17. $\langle F[\mathcal{S}k.\, s]\rangle \twoheadrightarrow^* \langle s\,[k := \lambda x.\, \langle F[x]\rangle]\rangle$.

Proof. If $F = [\,]$, $\langle \mathcal{S}k.\, s\rangle \twoheadrightarrow \langle s\,[k := \lambda x.\, \langle x\rangle]\rangle$, and so we finish. Otherwise, if $F \neq [\,]$, then

$$\begin{array}{lll} \langle F[\mathcal{S}k.\, s]\rangle & \twoheadrightarrow^* & \langle \mathcal{S}k'.\, s\,[k := \lambda x.\, \langle k'\ F[x]\rangle]\rangle \quad \text{(by Lemma A.1.16)} \\ & \twoheadrightarrow & \langle s\,[k := \lambda x.\, \langle (\lambda y.\, \langle y\rangle)\ F[x]\rangle]\rangle \\ & \twoheadrightarrow & \langle s\,[k := \lambda x.\, \langle F[x]\rangle]\rangle. \end{array}$$

□

Lemma A.1.18. *If $A/\alpha \to B/\beta \sim A'/\alpha' \to B'/\beta'$, then $A' \sim A$ and $B \sim B'$ and $\alpha' \sim \alpha$ and $\beta \sim \beta'$.*

Proof. Straightforward by case analysis on the compatibility rule applied last. □

Lemma 2 (Preservation). *Suppose that $\emptyset; \alpha \vdash s : A; \beta$.*

(1) If $s \longrightarrow t$, then $\emptyset; \alpha \vdash t : A; \beta$.

(2) If $s \longmapsto t$, then $\emptyset; \alpha \vdash t : A; \beta$.

Proof.

(1) By case analysis on the typing rule applied to s.

Case (T_CONST), (T_VAR), (T_ABS), (T_GROUND), (T_BLAME), (T_SHIFT): Contradictory.

Case (T_OP): We are given $\emptyset; \alpha \vdash op(\overline{t_i}^i) : A; \beta$ for some op and $\overline{t_i}^i$. The only reduction rule applicable to $op(\overline{t_i}^i)$ is (R_OP). By Lemma A.1.3 and the assumption on ζ, we finish.

Case (T_APP): We are given $\emptyset; \alpha \vdash t\ u : A; \beta$ for some t and u. By inversion, we have $\emptyset; \gamma \vdash t : B/\alpha \to A/\beta'; \beta$ and $\emptyset; \beta' \vdash u : B; \gamma$ for some β' and γ. The only reduction rule applicable to $t\ u$ is (R_BETA), so $t = \lambda x.\, t'$ for some x and t', and u is a value. By Lemma A.1.3, $\beta = \gamma = \beta'$. By Lemma A.1.6, $x{:}B; \alpha \vdash t : A; \beta$. By Lemma A.1.5, $\emptyset; \alpha \vdash t\,[x := u] : A; \beta$.

Case (T_CAST): We are given $\emptyset; \alpha \vdash t : B \Rightarrow^p A : A; \beta$ for some t, p and B. By inversion, we have $\emptyset; \alpha \vdash t : B; \beta$ and $B \sim A$. By case analysis on the reduction rule applicable to $t : B \Rightarrow^p A$.

Case (R_BASE) and (R_DYN): We are given $v : A \Rightarrow^p A \longrightarrow v$ where $v = t$ and $B = A$. Since $\emptyset; \alpha \vdash v : A; \beta$, we finish.

Case (R_WRAP): We are given

$$v : A'/\alpha' \to B'/\beta' \Rightarrow^p A''/\alpha'' \to B''/\beta'' \longrightarrow \lambda x.\, \mathcal{S}k.\, (\langle (k\ ((v\ (x : A'' \Rightarrow^{\bar{p}} A')) : B' \Rightarrow^p B'')) : \alpha'' \Rightarrow^{\bar{p}} \alpha' \rangle : \beta' \Rightarrow^p \beta'')$$

where $v = t$ and $B = A'/\alpha' \to B'/\beta'$ and $A = A''/\alpha'' \to B''/\beta''$. Since $B \sim A$, we have $A'' \sim A'$ and $B' \sim B''$ and $\alpha'' \sim \alpha'$ and $\beta' \sim \beta''$ by Lemma A.1.18.

Since $\alpha = \beta$ by Lemma A.1.3, we have

$$x{:}A''; \alpha' \vdash v\,(x : A'' \Rightarrow^{\bar{p}} A') : B' \Rightarrow^{p} B'' : B''; \beta'.$$

by Lemma A.1.1, (T_VAR), (T_CAST), and (T_APP). By Lemma A.1.1, (T_VAR), and (T_APP),

$$x{:}A'', k{:}B''/\alpha' \to \alpha''/\alpha'; \alpha' \vdash k\,(v\,(x : A'' \Rightarrow^{\bar{p}} A') : B' \Rightarrow^{p} B'') : \alpha''; \beta'.$$

By (T_CAST) and (T_RESET),

$$x{:}A'', k{:}B''/\alpha' \to \alpha''/\alpha'; \beta'' \vdash \langle (k\,(v\,(x : A'' \Rightarrow^{\bar{p}} A') : B' \Rightarrow^{p} B'')) : \alpha'' \Rightarrow^{\bar{p}} \alpha' \rangle : \beta'; \beta''.$$

By (T_CAST) and (T_SHIFT),

$$x{:}A''; \alpha'' \vdash \mathcal{S}k.\,(\langle (k\,(v\,(x : A'' \Rightarrow^{\bar{p}} A') : B' \Rightarrow^{p} B'')) : \alpha'' \Rightarrow^{\bar{p}} \alpha' \rangle : \beta' \Rightarrow^{p} \beta'') : B''; \beta''.$$

By (T_ABS),

$$\emptyset; \alpha \vdash \lambda x.\,\mathcal{S}k.\,(\langle (k\,(v\,(x : A'' \Rightarrow^{\bar{p}} A') : B' \Rightarrow^{p} B'')) : \alpha'' \Rightarrow^{\bar{p}} \alpha' \rangle : \beta' \Rightarrow^{p} \beta'') : A''/\alpha'' \to B''/\beta''; \beta$$

(note that $\alpha = \beta$).

Case (R_GROUND): We are given $v : B \Rightarrow^{p} \star \longrightarrow v : B \Rightarrow^{p} G \Rightarrow \star$ where $t = v$ and $A = \star$ and $B \sim G$. By (T_CAST) and (T_GROUND), we finish.

Case (R_COLLAPSE): We are given $v : G \Rightarrow \star \Rightarrow^{p} A \longrightarrow v : G \Rightarrow^{p} A$ where $t = v : G \Rightarrow \star$ and $B = \star$ and $G \sim A$ and $A \neq \star$. By Lemma A.1.7, $\emptyset; \alpha \vdash v : G; \beta$. Thus, we finish by (T_CAST).

Case (R_CONFLICT): We are given $v : G \Rightarrow \star \Rightarrow^{p} A \longrightarrow \mathsf{blame}\, p$. We finish by (T_BLAME).

Case (T_IS): We are given $\emptyset; \alpha \vdash t \,\mathsf{is}\, G : \mathsf{bool}; \beta$ for some t and G. By inversion, we have $\emptyset; \alpha \vdash t : \star; \beta$. By case analysis on the reduction rule applicable to $t \,\mathsf{is}\, G$.

Case (R_ISTRUE): We are given $(v : G \Rightarrow \star) \,\mathsf{is}\, G \longrightarrow \mathsf{true}$. By Lemma A.1.3 and (T_CONST), we finish.

Case (R_ISFALSE): We are given $(v : H \Rightarrow \star) \,\mathsf{is}\, G \longrightarrow \mathsf{false}$. By Lemma A.1.3 and (T_CONST), we finish.

Case (T_RESET): We are given $\emptyset; \alpha \vdash \langle t \rangle : A; \alpha$ for some t. By inversion, we have $\emptyset; \gamma \vdash t : \gamma; A$ for some γ. By case analysis on the reduction rule applicable to $\langle t \rangle$.

Case (R_RESET): We are given $\langle v \rangle \longrightarrow v$ where $t = v$. By Lemma A.1.3, $A = \gamma$, so $\emptyset; A \vdash t : A; A$. By Lemma A.1.4, we finish.

Case (R_SHIFT): We are given $\langle F[\mathcal{S}k.\,u] \rangle \longrightarrow \langle u\,[k := \lambda x.\,\langle F[x] \rangle] \rangle$ where $t = F[\mathcal{S}k.\,u]$ and $x \notin fv\,(F)$. By Lemmas A.1.17 and A.1.15.

(2) By case analysis on the evaluation rule applied.

Case (E_STEP): Straightforward by induction on the typing derivation of $\Gamma; \alpha \vdash s : A; \beta$ with case (1).

Case (E_ABORT): By (T_BLAME). □

A.2 Blame Theorem

In this section, we show Blame Theorem and Subtype Theorem (Theorem 2). We start with showing that naive and ordinal subtyping are characterized by using both positive and negative subtyping (Lemmas 4 and 5, respectively). Then, after proving Blame Progress (Lemma 6) and Blame Preservation (Lemma 7), we show Blame Theorem and Subtype Theorem.

Lemma 3. *If $A/\alpha \to B/\beta <:^- G$, then $A = \alpha = \star$ and $B <:^- \gamma$ and $\beta <:^- \gamma$ for any γ.*

Proof. By induction on the derivation of $A/\alpha \to B/\beta <:^- G$.

Case (S$^-$_REFL): Obvious.

Case (S$^-$_DYN): Contradictory.

Case (S$^-$_ANY): By the IH.

Case (S$^-$_FUN): We are given $G = \star/\star \to \star/\star$. By inversion, we have $\star <:^+ A$ and $B <:^- \star$ and $\star <:^+ \alpha$ and $\beta <:^- \star$. From (S$^+$_REFL) and (S$^+$_DYN), $A = \star$ and $\alpha = \star$. From (S$^-$_REFL), (S$^-$_DYN), and (S$^-$_ANY), $B = \star$ or $B <:^- G'$ for some G', and $\beta = \star$ or $\beta <:^- G''$ for some G''. Thus, (S$^-$_DYN) or (S$^-$_ANY), we finish.

□

Lemma A.2.1. *If $A <:_n B$, then*

(1) $A <:^+ B$ and

(2) $B <:^- A$.

Proof. By induction on the derivation of $A <:_n B$.

Case (SN_REFL): By (S$^+$_REFL) and (S$^-$_REFL).

Case (SN_DYN): By (S$^+$_DYN) and (S$^-$_DYN).

Case (SN_FUN): We are given $A'/\alpha' \to B'/\beta' <:_n A''/\alpha'' \to B''/\beta''$. By inversion, we have $A' <:_n A''$ and $B' <:_n B''$ and $\alpha' <:_n \alpha''$ and $\beta' <:_n \beta''$. By the IHs,

- $A' <:^+ A''$ and $A'' <:^- A'$,
- $B' <:^+ B''$ and $B'' <:^- B'$,
- $\alpha' <:^+ \alpha''$ and $\alpha'' <:^- \alpha'$, and
- $\beta' <:^+ \beta''$ and $\beta'' <:^- \beta'$.

Thus, by (S$^+$_FUN) and (S$^-$_FUN), we finish. □

Lemma A.2.2. *If $A <:^+ B$ and $B <:^- A$, then $A <:_n B$.*

Proof. By induction on the structure of A with case analysis on the rule applied last to derive $A <:^+ B$.

Case (S^+_REFL): By (SN_REFL).

Case (S^+_DYN): By (SN_DYN).

Case (S^+_FUN): We are given $A'/\alpha' \to B'/\beta' <:^+ A''/\alpha'' \to B''/\beta''$. By inversion, we have $A'' <:^- A'$ and $B' <:^+ B''$ and $\alpha'' <:^- \alpha'$ and $\beta' <:^+ \beta''$. By case analysis on the rule applied last to derive $A''/\alpha'' \to B''/\beta'' <:^- A'/\alpha' \to B'/\beta'$.

- Case (S^-_REFL): By (S^+_REFL) and (S^-_REFL), $A' <:^+ A''$ and $B'' <:^- B'$ and $\alpha' <:^+ \alpha''$ and $\beta'' <:^- \beta'$. Thus, by the IHs, $A' <:_n A''$ and $B' <:_n B''$ and $\alpha' <:_n \alpha''$ and $\beta' <:_n \beta''$. Thus, by (SN_FUN), we finish.
- Case (S^-_DYN): Contradictory.
- Case (S^-_ANY): By inversion, we have $A''/\alpha'' \to B''/\beta'' <:^- G$ for some G. By Lemma 3, $A'' = \alpha'' = \star$ and $B' <:^- B''$ and $\beta' <:^- \beta''$. By (S^+_DYN), $A' <:^+ A''$ and $\alpha' <:^+ \alpha''$. Thus, we finish by the IHs and (SN_FUN).
- Case (S^-_FUN): By the IHs and (SN_FUN). □

Lemma 4. *$A <:_n B$ iff $A <:^+ B$ and $B <:^- A$.*

Proof. By Lemmas A.2.1 and A.2.2. □

Lemma A.2.3. *If $A <: B$, then $A <:^+ B$ and $A <:^- B$.*

Proof. By induction on the derivation of $A <: B$.

Case (S_REFL): By (S^+_REFL) and (S^-_REFL).

Case (S_DYN): We are given $A <: \star$. By inversion, $A <: G$. By the IH, $A <:^- G$. Thus, by (S^-_ANY), $A <:^- \star$. By (S^+_DYN), $A <:^+ \star$.

Case (S_FUN): By the IHs, (S^+_FUN), and (S^-_FUN). □

Lemma A.2.4. *If $A <:^+ B$ and $A <:^- B$, then $A <: B$.*

Proof. By induction on the structure of A with case analysis on the rule applied last to derive $A <:^+ B$.

Case (S^+_REFL): By (S_REFL).

Case (S^+_DYN): We are given $A <:^+ \star$. Since $A <:^- \star$, $A = \star$ from (S^-_REFL) and (S^-_DYN), or $A <:^- G$ for some G from (S^-_ANY). If $A = \star$, then we finish by (S_REFL). Otherwise, if $A <:^- G$ for some G, by case analysis on A.

- Case $A = \star$: By (S_REFL).
- Case $A = \iota$: Since $\iota <: \iota$ by (S_REFL), we finish by (S_DYN).
- Case $A = A'/\alpha' \to B'/\beta'$: By Lemma 3, $A' = \alpha' = \star$ and $B' <:^- \star$ and $\beta' <:^- \star$. Since $B' <:^+ \star$ and $\beta' <:^+ \star$ by (S^+_DYN), we have $B' <: \star$ and $\beta' <: \star$ by the IHs. Since $\star <: A'$ and $\star <: \alpha'$ by (S_REFL), we have $A'/\alpha' \to B'/\beta' <: \star/\star \to \star/\star$ by (S_FUN), and so $A'/\alpha' \to B'/\beta' <: \star$ by (S_DYN).

Case (S^+_FUN): We are given $A'/\alpha' \to B'/\beta' <:^+ A''/\alpha'' \to B''/\beta''$ for some A', B', α', β', A'', B'', α'', and β''. By inversion, we have $A'' <:^- A'$ and $B' <:^+ B''$ and $\alpha'' <:^- \alpha'$ and $\beta' <:^+ \beta''$. By case analysis on the rule applied last to derive $A'/\alpha' \to B'/\beta' <:^- A''/\alpha'' \to B''/\beta''$.

Case (S^-_REFL): By (S_REFL).

Case (S^-_DYN): Contradictory.

Case (S^-_ANY): By inversion, we have $A'/\alpha' \to B'/\beta' <:^- G$ for some G. By Lemma 3, $A' = \alpha' = \star$ and $B' <:^- B''$ and $\beta' <:^- \beta''$. By the IHs, $B' <: B''$ and $\beta' <: \beta''$. Since $A'' <:^+ A'$ and $\alpha'' <:^+ \alpha'$ by (S^+_DYN), we have $A'' <: A'$ and $\alpha'' <: \alpha'$ by the IHs. Thus, we finish by (S_FUN).

Case (S^-_FUN): By the IHs and (S_FUN). □

Lemma 5. *$A <: B$ iff $A <:^+ B$ and $A <:^- B$.*

Proof. By Lemmas A.2.3 and A.2.4. □

Lemma 6 (Blame Progress)**.** *If s sf p, then $s \not\longmapsto$ blame p.*

Proof. By induction on the derivation of s sf p.

Case (SF_POS): We are given $t : A \Rightarrow^p B$ sf p for some t, A, and B. By inversion, we have t sf p and $A <:^+ B$. By case analysis on t.

Case $t \longmapsto u$: By the IH, and (E_STEP) or (E_ABORT).

Case $t =$ blame q: Obviously $q \neq p$, so we finish by (E_ABORT).

Case $t = v$: The reduction rule that implies blame p is only (R_CONFLICT), so we consider only it. In that case, we are given $A = \star$ and $t = v' : G \Rightarrow \star$ for some v' and G such that $G \not\sim B$. Since $\star <:^+ B$, $B = \star$ from (S^+_REFL) and (S^+_DYN). Here, $G \not\sim \star$ contradicts from the fact that $\alpha \sim \star$ for any type α.

Case otherwise: $t : A \Rightarrow^p B$ does not takes a step.

Case (SF_NEG): We are given $t : A \Rightarrow^{\bar{p}} B$ sf p for some t, A, and B. By inversion, we have t sf p and $A <:^- B$. By case analysis on t.

Case $t \longmapsto u$: By the IH, and (E_STEP) or (E_ABORT).

Case $t =$ blame q: Obviously $q \neq p$, so we finish.

Case $t = v$: There are no reduction rules that imply blame p.

Case otherwise: $t : A \Rightarrow^p B$ does not takes a step.

Case (SF_CONST), (SF_VAR), and (SF_ABS), (SF_BLAME), (SF_SHIFT): Obvious.

Case (SF_OP): By the IHs and (SF_CONST).

Case (SF_APP): By the IHs.

Case (SF_CAST), (SF_GROUND), (SF_IS): By the IH.

Case (SF_RESET): By the IH. □

Lemma A.2.5 (Substitution of Safety Value)**.** *If t sf p and v sf p, then $t\,[x := v]$ sf p.*

Proof. Straightforward by induction on the derivation of t sf p. □

Lemma A.2.6. *If $F[\mathcal{S}k.\, s]$ sf p, then $F[x]$ sf p for any x.*

Proof. Straightforward by induction on the derivation of $F[\mathcal{S}k.\, s]$ sf p. □

Lemma 7 (Blame Preservation)**.** *(1) If s sf p and $s \longrightarrow t$, then t sf p.*

(2) If s sf p *and* $s \longmapsto t$, *then* t sf p.

Proof. (1) By induction on the derivation of s sf p.

Case (SF_POS): We are given $t : A \Rightarrow^p B$ sf p for some t, A, and B. By inversion, we have t sf p and $A <:^+ B$. By case analysis on the reduction rule applied to $t : A \Rightarrow^p B$. In what follows, we suppose that $t = v$ for some value v.

Case (R_BASE) and (R_DYN): Obvious.

Case (R_WRAP): We are given

$$v : A'/\alpha' \to B'/\beta' \Rightarrow^p A''/\alpha'' \to B''/\beta'' \longrightarrow \lambda x.\mathcal{S}k.(\langle (k\,((v\,(x : A'' \Rightarrow^{\bar{p}} A')) : B' \Rightarrow^p B''))\rangle : \alpha'' \Rightarrow^{\bar{p}} \alpha'\rangle : \beta' \Rightarrow^p \beta'')$$

where $A = A'/\alpha' \to B'/\beta'$ and $B = A''/\alpha'' \to B''/\beta''$. Since $A'/\alpha' \to B'/\beta' <:^+ A''/\alpha'' \to B''/\beta''$, we have $A'' <:^- A'$ and $B' <:^+ B''$ and $\alpha'' <:^- \alpha'$ and $\beta' <:^+ \beta''$. By (SF_POS), (SF_NEG), and other rules, we finish.

Case (R_GROUND): We are given $v : A \Rightarrow^p \star \longrightarrow v : A \Rightarrow^p G \Rightarrow \star$ for some G such that $A \sim G$. Note that $A \neq \star$. From (SF_GROUND) and (SF_CAST), it suffices to show that $A <:^+ G$. By case analysis on A.

Case $A = \iota$: Then, $G = \iota$, and so we finish by (S^+_REFL).

Case $A = A'/\alpha' \to B'/\alpha'$: Then, $G = \star/\star \to \star/\star$. Since $\star <:^- A'$ and $B' <:^+ \star$ and $\star <:^- \alpha'$ and $\beta' <:^+ \star$ by (S^-_DYN) and (S^+_DYN), we have $A'/\alpha' \to B'/\beta' <:^+ \star/\star \to \star/\star$ by (S^+_FUN).

Case (R_COLLAPSE): We are given $v' : G \Rightarrow \star \Rightarrow^p B \longrightarrow v' : G \Rightarrow^p B$ for some v' and G such that $G \sim B$ and $B \neq \star$. Since $\star <:^+ B$, $B = \star$ from (S^+_REFL) and (S^+_DYN). Thus, contradictory.

Case (R_CONFLICT): We are given $v' : G \Rightarrow \star \Rightarrow^p B \longrightarrow \mathsf{blame}\, p$ for some v' and G such that $G \not\sim B$. Since $\star <:^+ B$, $B = \star$ from (S^+_REFL) and (S^+_DYN). However, it contradicts from $G \not\sim B$ since $G \sim \star$ by (C_DYNTO).

Case (SF_NEG): We are given $t : A \Rightarrow^{\bar{p}} B$ sf p for some t, A, and B. By inversion, we have t sf p and $A <:^- B$. By case analysis on the reduction rule applied to $t : A \Rightarrow^{\bar{p}} B$. In what follows, we suppose that $t = v$ for some value v.

Case (R_BASE) and (R_DYN): Obvious.

Case (R_WRAP): We are given

$$v : A'/\alpha' \to B'/\beta' \Rightarrow^{\bar{p}} A''/\alpha'' \to B''/\beta'' \longrightarrow \lambda x.\mathcal{S}k.(\langle (k\,((v\,(x : A'' \Rightarrow^p A')) : B' \Rightarrow^{\bar{p}} B''))\rangle : \alpha'' \Rightarrow^p \alpha'\rangle : \beta' \Rightarrow^{\bar{p}} \beta'')$$

where $A = A'/\alpha' \to B'/\beta'$ and $B = A''/\alpha'' \to B''/\beta''$. It suffices to show that $A'' <:^+ A'$ and $B' <:^- B''$ and $\alpha'' <:^+ \alpha'$ and $\beta' <:^- \beta''$. By case analysis on the rule applied last to derive $A'/\alpha' \to B'/\beta' <:^- A''/\alpha'' \to B''/\beta''$.

Case (S^-_FUN): Obvious.

Case (S^-_ANY): By inversion, $A'/\alpha' \to B'/\beta' <:^- G$ for some G. By Lemma 3, $A' = \alpha' = \star$ and $B' <:^- B''$ and $\beta' <:^- \beta''$. By (S^+_DYN), $A'' <:^+ A'$ and $\alpha'' <:^+ \alpha'$.

Case (R_GROUND): We are given $v : A \Rightarrow^{\bar{p}} \star \longrightarrow v : A \Rightarrow^{\bar{p}} G \Rightarrow \star$ for some G such that $A \sim G$. Note that $A \neq \star$. From (SF_GROUND) and (SF_CAST), it suffices to show that $A <:^{-} G$.
Since $A <:^{-} \star$, we have $A = \star$ from (S^{-}_REFL) and (S^{-}_DYN) or $A <:^{-} H$ for some H. If $A = \star$, then it contradicts from $A \neq \star$; otherwise, if $A <:^{-} H$, then we finish by (S^{-}_ANY).

Case (R_COLLAPSE): We are given $v' : G \Rightarrow \star \Rightarrow^{\bar{p}} B \longrightarrow v' : G \Rightarrow^{\bar{p}} B$ for some v' and G such that $G \sim B$ and $B \neq \star$. Since $G <:^{-} G$ by (S^{-}_REFL), we have $G <:^{-} B$ by (S^{-}_ANY).

Case (R_CONFLICT): We are given $v' : G \Rightarrow \star \Rightarrow^{\bar{p}} B \longrightarrow \mathsf{blame}\,\bar{p}$ for some v' and G such that $G \not\sim B$. By (SF_BLAME).

Case (SF_CONST), (SF_VAR), (SF_ABS), (SF_GROUND), (SF_BLAME), and (SF_SHIFT): Contradictory.

Case (SF_OP): By (SF_CONST).

Case (SF_APP): By Lemma A.2.5.

Case (SF_CAST): Straightforward by safety rules.

Case (SF_IS): By (SF_CONST).

Case (SF_RESET): By Lemmas A.2.6 and A.2.5.

(2) By case analysis on the evaluation rule applied.

Case (E_STEP): Straightforward by induction on the derivation of $s\ \mathsf{sf}\ p$ with case (1).

Case (E_ABORT): We are given $E[\mathsf{blame}\,q] \longmapsto \mathsf{blame}\,q$ for some E and q. Since $E[\mathsf{blame}\,q]\ \mathsf{sf}\ p$, $q \neq p$. Thus, we finish by (SF_BLAME). □

Theorem 2 (Blame Theorem and Subtype Theorem)**.** *Let s be a term with a subterm $t : A \Rightarrow^{p} B$ where cast is labeled by the only occurrence of p in s. Moreover, suppose that $\bar{p}$ does not appear in s.*

(1) If $A <:^{+} B$, then $s \mapsto\!\!\!\!\!\rightarrow^{*} \mathsf{blame}\,p$.

(2) If $A <:^{-} B$, then $s \mapsto\!\!\!\!\!\rightarrow^{*} \mathsf{blame}\,\bar{p}$.

(3) If $A <:_{n} B$, then $s \mapsto\!\!\!\!\!\rightarrow^{*} \mathsf{blame}\,p$; if $B <:_{n} A$, then $s \mapsto\!\!\!\!\!\rightarrow^{*} \mathsf{blame}\,\bar{p}$.

(4) If $A <: B$, then $s \mapsto\!\!\!\!\!\rightarrow^{*} \mathsf{blame}\,p$ and $s \mapsto\!\!\!\!\!\rightarrow^{*} \mathsf{blame}\,\bar{p}$.

Proof.

(1) Since $s\ \mathsf{sf}\ p$, we finish by Lemmas 6 and 7 (2).

(2) Since $s\ \mathsf{sf}\ \bar{p}$, we finish by Lemmas 6 and 7 (2).

(3) By cases (1) and (2) and Lemma 4.

(4) By cases (1) and (2) and Lemma 5. □

A.3 CPS Transformation

This sections shows two properties as a proof of correctness of our CPS transformation. The first property (Theorem 3), which states that the transformation preserves well-typedness, is easy to show, while the second (Theorem 4), which states that the transformation preserves term equality, is not.

Lemma A.3.1. *If* $A \sim B$, *then* $[\![A]\!] \sim [\![B]\!]$.

Proof. By induction on the derivation of $A \sim B$.

Case (C_DynTo), (C_DynFrom), and (C_Base): Obvious.

Case (C_Fun): We are given $A'/\alpha' \to B'/\beta' \sim A''/\alpha'' \to B''/\beta''$. By inversion, we have $A'' \sim A'$ and $B' \sim B''$ and $\alpha'' \sim \alpha'$ and $\beta' \sim \beta''$. By the IHs, $[\![A'']\!] \sim [\![A']\!]$ and $[\![B']\!] \sim [\![B'']\!]$ and $[\![\alpha'']\!] \sim [\![\alpha']\!]$ and $[\![\beta']\!] \sim [\![\beta'']\!]$. Here, $[\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!] \sim [\![A'']\!] \to ([\![B'']\!] \to [\![\alpha'']\!]) \to [\![\beta'']\!]$ holds.

□

Theorem 3 (Preservation of Type). *If* $\Gamma; \alpha \vdash s : A; \beta$, *then* $[\![\Gamma]\!] \vdash [\![s]\!] : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$.

Proof. By induction on the typing derivation. In the following, we use weakening and substitution lemmas for the target calculus; it is easy to prove the lemmas.

Case (T_Const): We are given $\Gamma; \alpha \vdash c : ty\,(c); \alpha$. By definition, $[\![c]\!] = \lambda\kappa.\, \kappa\, c$. Since $[\![ty\,(c)]\!] = ty\,(c)$, $[\![\Gamma]\!] \vdash \lambda\kappa.\, \kappa\, c : (ty\,(c) \to [\![\alpha]\!]) \to [\![\alpha]\!]$ holds obviously.

Case (T_Op): We are given $\Gamma; \alpha \vdash op(\overline{t_i}^{\,i}) : \iota; \beta$. By inversion, we have, for any i, $\Gamma; \alpha_i \vdash t_i : \iota_i; \alpha_{i-1}$, and $ty\,(op) = \overline{\iota_i}^{\,i} \to \iota$ and $\alpha = \alpha_n$ and $\beta = \alpha_0$. By the IHs, $[\![\Gamma]\!] \vdash [\![t_i]\!] : ([\![\iota_i]\!] \to [\![\alpha_i]\!]) \to [\![\alpha_{i-1}]\!]$, for any i. Thus, $[\![\Gamma]\!], \kappa{:}[\![\iota]\!] \to [\![\alpha]\!] \vdash [\![t_1]\!]\,(\lambda x_1. \ldots [\![t_n]\!]\,(\lambda x_n.\, \kappa\, op(\overline{x_i}^{\,i}))\ldots) : \beta$. By definition of CPS transformation, we finish.

Case (T_Var): We are given $\Gamma; \alpha \vdash x : A; \alpha$. By inversion, we have $x{:}A \in \Gamma$. Since $[\![\Gamma]\!] \vdash x : [\![A]\!]$ and $[\![x]\!] = \lambda\kappa.\, \kappa\, x$, we have $[\![\Gamma]\!] \vdash \lambda\kappa.\, \kappa\, x : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\alpha]\!]$.

Case (T_Abs): We are given $\Gamma; \alpha \vdash \lambda x.\, t : A'/\alpha' \to B'/\beta'; \alpha$. By inversion, we have $\Gamma, x{:}A'; \alpha' \vdash t : B'; \beta'$. By the IH, $[\![\Gamma]\!], x{:}[\![A']\!] \vdash [\![t]\!] : ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]$. Then, $[\![\Gamma]\!] \vdash \lambda x.\, [\![t]\!] : [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]$. Here, we have $[\![\Gamma]\!] \vdash \lambda\kappa.\, \kappa\, (\lambda x.\, [\![t]\!]) : (([\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \to [\![\alpha]\!]) \to [\![\alpha]\!]$. Since $[\![\lambda x.\, t]\!] = \lambda\kappa.\, \kappa\, (\lambda x.\, [\![t]\!])$, we finish.

Case (T_App): We are given $\Gamma; \alpha \vdash t\, u : A; \beta$. By inversion, we have $\Gamma; \gamma \vdash t : A'/\alpha \to A/\beta'; \beta$ and $\Gamma; \beta' \vdash u : A'; \gamma$. Here,

$$[\![\Gamma]\!], \kappa{:}[\![A]\!] \to [\![\alpha]\!], x{:}[\![A']\!] \to ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta']\!], y{:}[\![A']\!] \vdash x\, y\, \kappa : [\![\beta']\!].$$

Thus,

$$[\![\Gamma]\!], \kappa{:}[\![A]\!] \to [\![\alpha]\!], x{:}[\![A']\!] \to ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta']\!] \vdash \lambda y.\, x\, y\, \kappa : [\![A']\!] \to [\![\beta']\!].$$

Since $[\![\Gamma]\!] \vdash [\![u]\!] : ([\![A']\!] \to [\![\beta']\!]) \to [\![\gamma]\!]$ by the IH, we have

$$[\![\Gamma]\!], \kappa{:}[\![A]\!] \to [\![\alpha]\!] \vdash \lambda x.\, [\![u]\!]\,(\lambda y.\, x\, y\, \kappa) : ([\![A']\!] \to ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta']\!]) \to [\![\gamma]\!].$$

Since $[\![\Gamma]\!] \vdash [\![t]\!] : (([\![A']\!] \to ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta']\!]) \to ([\![\gamma]\!])) \to [\![\beta]\!]$ by the IH, we have

$$[\![\Gamma]\!] \vdash \lambda\kappa.\, [\![t]\!]\,(\lambda x.\, [\![u]\!]\,(\lambda y.\, x\; y\; \kappa)) : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta]\!].$$

Since $[\![t\; u]\!] = \lambda\kappa.\, [\![t]\!]\,(\lambda x.\, [\![u]\!]\,(\lambda y.\, x\; y\; \kappa))$, we finish.

Case (T_CAST): We are given $\Gamma; \alpha \vdash t : A' \Rightarrow^p A : A; \beta$. By inversion, we have $\Gamma; \alpha \vdash t : A'; \beta$ and $A' \sim A$. By the IH, $[\![\Gamma]\!] \vdash [\![t]\!] : ([\![A']\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$. By Lemma A.3.1, $[\![\Gamma]\!], x{:}[\![A']\!] \vdash x : [\![A']\!] \Rightarrow^p [\![A]\!] : [\![A]\!]$. Thus, $[\![\Gamma]\!], \kappa{:}[\![A]\!] \to [\![\alpha]\!] \vdash [\![t]\!]\,(\lambda x.\, \kappa\,(x : [\![A']\!] \Rightarrow^p [\![A]\!])) : [\![\beta]\!]$, and so $[\![\Gamma]\!] \vdash \lambda\kappa.\, [\![t]\!]\,(\lambda x.\, \kappa\,(x : [\![A']\!] \Rightarrow^p [\![A]\!])) : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$. We finish by definition of CPS transformation.

Case (T_GROUND): We are given $\Gamma; \alpha \vdash t : G \Rightarrow_p \star : \star; \beta$. By inversion, we have $\Gamma; \alpha \vdash t : G; \beta$. By the IH, $[\![\Gamma]\!] \vdash [\![t]\!] : ([\![G]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$. By case analysis on G.

- Case $G = \iota$: By definition, $[\![t : \iota \Rightarrow_p \star]\!] = \lambda\kappa.\, [\![t]\!]\,(\lambda x.\, \kappa\,(x : \iota \Rightarrow \star))$. Since $[\![\Gamma]\!], \kappa{:}\star \to [\![\alpha]\!] \vdash \lambda x.\, \kappa\,(x : \iota \Rightarrow \star) : [\![\iota]\!] \to [\![\alpha]\!]$, we have $[\![\Gamma]\!] \vdash \lambda\kappa.\, [\![t]\!]\,(\lambda x.\, \kappa\,(x : \iota \Rightarrow \star)) : (\star \to [\![\alpha]\!]) \to [\![\beta]\!]$.
- Case $G = \star/\star \to \star/\star$: By definition, $[\![t : \star/\star \to \star/\star \Rightarrow_p \star]\!] = \lambda\kappa.\, [\![t]\!]\,(\lambda x.\, \kappa\,((\lambda y.\, (x\; y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star))$. Since $[\![\Gamma]\!], \kappa{:}\star \to [\![\alpha]\!] \vdash \lambda x.\, \kappa\,((\lambda y.\, (x\; y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star) : [\![\star/\star \to \star/\star]\!] \to [\![\alpha]\!]$, we have $[\![\Gamma]\!] \vdash \lambda\kappa.\, [\![t]\!]\,(\lambda x.\, \kappa\,((\lambda y.\, (x\; y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)) : (\star \to [\![\alpha]\!]) \to [\![\beta]\!]$.

Case (T_IS): We are given $\Gamma; \alpha \vdash t \,\mathsf{is}\, G : \mathsf{bool}; \beta$. By inversion, we have $\Gamma; \alpha \vdash t : \star; \beta$. By the IH, $[\![\Gamma]\!] \vdash [\![t]\!] : (\star \to [\![\alpha]\!]) \to [\![\beta]\!]$. By case analysis on G.

- Case $G = \iota$: By definition, $[\![t \,\mathsf{is}\, \iota]\!] = \lambda\kappa.\, [\![t]\!]\,(\lambda x.\, \kappa\,(x \,\mathsf{is}\, \iota))$. Since $[\![\Gamma]\!], \kappa{:}\mathsf{bool} \to [\![\alpha]\!] \vdash \lambda x.\, \kappa\,(x \,\mathsf{is}\, \iota) : \star \to [\![\alpha]\!]$, we have $[\![\Gamma]\!] \vdash \lambda\kappa.\, [\![t]\!]\,(\lambda x.\, \kappa\,(x \,\mathsf{is}\, \iota)) : (\mathsf{bool} \to [\![\alpha]\!]) \to [\![\beta]\!]$.
- Case $G = \star/\star \to \star/\star$: Similarly to the above.

Case (T_BLAME): We are given $\Gamma; \alpha \vdash \mathsf{blame}\, p : A; \beta$. Since $[\![\mathsf{blame}\, p]\!] = \lambda\kappa.\, \mathsf{blame}\, p$ and $[\![\Gamma]\!] \vdash \lambda\kappa.\, \mathsf{blame}\, p : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$, we finish.

Case (T_SHIFT): We are given $\Gamma; \alpha \vdash \mathcal{S}k.\, t : A; \beta$. By inversion, $\Gamma, k{:}A/\gamma \to \alpha/\gamma; \delta \vdash t : \delta; \beta$. By the IH, $[\![\Gamma]\!], k{:}[\![A]\!] \to ([\![\alpha]\!] \to [\![\gamma]\!]) \to [\![\gamma]\!] \vdash [\![t]\!] : ([\![\delta]\!] \to [\![\delta]\!]) \to [\![\beta]\!]$. By definition, $[\![\mathcal{S}k.\, t]\!] = \lambda\kappa.\, ([\![t]\!]\,(\lambda x.\, x))\,[k := \lambda x.\, \lambda\kappa'.\, \kappa'\,(\kappa\, x)]$. Since $[\![\Gamma]\!], \kappa{:}[\![A]\!] \to [\![\alpha]\!] \vdash \lambda x.\, \lambda\kappa'.\, \kappa'\,(\kappa\, x) : [\![A]\!] \to ([\![\alpha]\!] \to [\![\gamma]\!]) \to [\![\gamma]\!]$, we have $[\![\Gamma]\!], \kappa{:}[\![A]\!] \to [\![\alpha]\!] \vdash ([\![t]\!]\,(\lambda x.\, x))\,[k := \lambda x.\, \lambda\kappa'.\, \kappa'\,(\kappa\, x)] : [\![\beta]\!]$. Thus, $[\![\Gamma]\!] \vdash \lambda\kappa.\, ([\![t]\!]\,(\lambda x.\, x))\,[k := \lambda x.\, \lambda\kappa'.\, \kappa'\,(\kappa\, x)] : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]$.

Case (T_RESET): We are given $\Gamma; \alpha \vdash \langle t \rangle : A; \alpha$. By inversion, $\Gamma; \gamma \vdash t : \gamma; A$. By the IH, $[\![\Gamma]\!] \vdash [\![t]\!] : ([\![\gamma]\!] \to [\![\gamma]\!]) \to [\![A]\!]$. By definition, $[\![\langle t \rangle]\!] = \lambda\kappa.\, \kappa\,([\![t]\!]\,(\lambda x.\, x))$ Since $[\![\Gamma]\!] \vdash [\![t]\!]\,(\lambda x.\, x) : [\![A]\!]$, we have $[\![\Gamma]\!] \vdash \lambda\kappa.\, \kappa\,([\![t]\!]\,(\lambda x.\, x)) : ([\![A]\!] \to [\![\alpha]\!]) \to [\![\alpha]\!]$. □

From now, we show that our CPS transformation preserves term equality. To clarify statements and proofs, we define term relations each of which is the least congruence satisfying one axiom in Definition 1. Recall that we use metavariables $\mathbb{e}$, $\mathbb{E}$, and $\mathbb{A}$ (and $\mathbb{B}$) to denote terms, evaluation contexts, and types in the target calculus, respectively, and $\Longrightarrow$ to denote the evaluation relation in the target calculus, and write $fv\,(\mathbb{v})$ and $fv\,(\mathbb{E})$ for the sets of free variables in $\mathbb{v}$ and $\mathbb{E}$, respectively.

Definition 11 (Term Equality). *Let*

- $\overset{\beta}{=}$ *be the least congruence that contains evaluation relation* $\Longrightarrow$,
- $\overset{\eta}{=}$ *be the least congruence that relates* $\lambda x.\, \mathbb{v}\, x$ *to* $\mathbb{v}$ *for any* x *and* v *such that* $x \notin fv(\mathbb{v})$,
- $\overset{\omega}{=}$ *be the least congruence that relates* $(\lambda x.\, \mathbb{E}[x])\, \mathbb{e}$ *to* $\mathbb{E}[\mathbb{e}]$ *for any* x, $\mathbb{E}$, *and* $\mathbb{e}$ *such that* $x \notin fv(\mathbb{E})$,
- $\overset{\xi}{=}$ *be the least congruence that relates* $\mathbb{e} : \star \Rightarrow^p \star \to \star \Rightarrow^p \mathbb{A} \to \mathbb{B}$ *to* $\mathbb{e} : \star \Rightarrow^p \mathbb{A} \to \mathbb{B}$ *for any* $\mathbb{e}$, p, $\mathbb{A}$, *and* $\mathbb{B}$, *and*
- $\overset{\upsilon}{=}$ *be the least congruence that relates* $\mathbb{e} : \star \Rightarrow^p \star$ *to* $\mathbb{e}$ *for any* $\mathbb{e}$ *and* p.

For any subset Σ *of* $\{\beta, \eta, \omega, \xi, \upsilon\}$, *we write* $\overset{\Sigma}{=}$ *to denote the transitive and symmetric closure of* $\bigcup_{\sigma \in \Sigma} \overset{\sigma}{=}$.

Note that the relation $\approx$ given in Definition 1 coincides with $\overset{\beta\eta\omega\xi\upsilon}{=}$.

Lemma A.3.2. *If* $G \sim \iota$, *then* $G = \iota$.

Proof. Straightforward by case analysis on the compatibility rule applied last to $G \sim \iota$. □

Lemma A.3.3. *If* $G \sim A/\alpha \to B/\beta$, *then* $G = \star/\star \to \star/\star$.

Proof. Straightforward by case analysis on the compatibility rule applied last to $G \sim A/\alpha \to B/\beta$. □

Lemma A.3.4.

(1) $[\![s\,[x := v]]\!] = [\![s]\!]\,[x := v^*]$.

(2) $(v'\,[x := v])^* = v'^*\,[x := v^*]$.

Proof. By mutual induction on structures of s and v'.

(1) By case analysis on s.

Case $s = v'$: $[\![v'\,[x := v]]\!] = \lambda\kappa.\, \kappa\,(v'\,[x := v])^*$. Similarly to case (2), we have $(v'\,[x := v])^* = v'^*\,[x := v^*]$. Thus, $[\![v'\,[x := v]]\!] = \lambda\kappa.\, \kappa\, v'^*\,[x := v^*] = [\![v']\!]\,[x := v^*]$.

Case $s = op(\overline{t_i}^{\,i})$: Similarly to the case for function applications.

Case $s = t\ u$:

$$
\begin{aligned}
& [\![(t\ u)\,[x := v]]\!] \\
= \ & [\![(t\,[x := v])\,(u\,[x := v])]\!] \\
= \ & \lambda\kappa.\, [\![t\,[x := v]]\!]\,(\lambda y.\, [\![u\,[x := v]]\!]\,(\lambda z.\, y\ z\ \kappa)) \\
= \ & \lambda\kappa.\, [\![t]\!]\,[x := v^*]\,(\lambda y.\, [\![u]\!]\,[x := v^*]\,(\lambda z.\, y\ z\ \kappa)) \quad \text{(by the IHs)} \\
= \ & (\lambda\kappa.\, [\![t]\!]\,(\lambda y.\, [\![u]\!]\,(\lambda z.\, y\ z\ \kappa)))\,[x := v^*] \\
= \ & [\![t\ u]\!]\,[x := v^*].
\end{aligned}
$$

Case $s = \langle t \rangle$:

$$\begin{array}{rcll}
[\![\langle t \rangle\,[x := v]]\!] & = & [\![\langle t\,[x := v] \rangle]\!] & \\
& = & \lambda\kappa.\,\kappa\,([\![t\,[x := v]]\!]\,(\lambda y.\,y)) & \\
& = & \lambda\kappa.\,\kappa\,([\![t]\!]\,[x := v^*]\,(\lambda y.\,y)) & \text{(by the IH)} \\
& = & (\lambda\kappa.\,\kappa\,([\![t]\!]\,(\lambda y.\,y)))\,[x := v^*] & \\
& = & [\![\langle t \rangle]\!]\,[x := v^*]. &
\end{array}$$

Case $s = \mathcal{S}k.\,t$: Without loss of generality, we can suppose that $k \neq x$ and $k \notin fv\,(v) \cup fv\,(v^*)$.

$$\begin{array}{rcll}
 & & [\![(\mathcal{S}k.\,t)\,[x := v]]\!] & \\
& = & [\![(\mathcal{S}k.\,t\,[x := v])]\!] & \\
& = & \lambda\kappa.\,[\![t\,[x := v]]\!]\,[k := \lambda y.\,\lambda\kappa'.\,\kappa'\,(\kappa\,y)]\,(\lambda z.\,z) & \\
& = & \lambda\kappa.\,[\![t]\!]\,[x := v^*]\,[k := \lambda y.\,\lambda\kappa'.\,\kappa'\,(\kappa\,y)]\,(\lambda z.\,z) & \text{(by the IH)} \\
& = & (\lambda\kappa.\,[\![t]\!]\,[k := \lambda y.\,\lambda\kappa'.\,\kappa'\,(\kappa\,y)])\,[x := v^*]\,(\lambda z.\,z) & \\
& = & [\![\mathcal{S}k.\,t]\!]\,[x := v^*]. &
\end{array}$$

Case $s = t : A \Rightarrow^p B$:

$$\begin{array}{rcll}
 & & [\![(t : A \Rightarrow^p B)\,[x := v]]\!] & \\
& = & [\![t\,[x := v] : A \Rightarrow^p B]\!] & \\
& = & \lambda\kappa.\,[\![t\,[x := v]]\!]\,(\lambda y.\,\kappa\,(y : [\![A]\!] \Rightarrow^p [\![B]\!])) & \\
& = & \lambda\kappa.\,[\![t]\!]\,[x := v^*]\,(\lambda y.\,\kappa\,(y : [\![A]\!] \Rightarrow^p [\![B]\!])) & \text{(by the IH)} \\
& = & (\lambda\kappa.\,[\![t]\!]\,(\lambda y.\,\kappa\,(y : [\![A]\!] \Rightarrow^p [\![B]\!])))\,[x := v^*] & \\
& = & [\![t : A \Rightarrow^p B]\!]\,[x := v^*]. &
\end{array}$$

Case $s = t : G \Rightarrow \star$:

$$\begin{array}{rcll}
 & & [\![(t : G \Rightarrow \star)\,[x := v]]\!] & \\
& = & [\![t\,[x := v] : G \Rightarrow \star]\!] & \\
& = & \lambda\kappa.\,[\![t\,[x := v]]\!]\,(\lambda y.\,\kappa\,(y : G \Rightarrow \star)^*) & \\
& = & \lambda\kappa.\,[\![t]\!]\,[x := v^*]\,(\lambda y.\,\kappa\,(y : G \Rightarrow \star)^*) & \text{(by the IH)} \\
& = & (\lambda\kappa.\,[\![t]\!]\,(\lambda y.\,\kappa\,(y : G \Rightarrow \star)^*))\,[x := v^*] & \\
& = & [\![t : G \Rightarrow \star]\!]\,[x := v^*]. &
\end{array}$$

Case $s = t\,\mathsf{is}\,\iota$:

$$\begin{array}{rcll}
[\![(t\,\mathsf{is}\,\iota)\,[x := v]]\!] & = & [\![t\,[x := v]\,\mathsf{is}\,\iota]\!] & \\
& = & \lambda\kappa.\,[\![t\,[x := v]]\!]\,(\lambda y.\,\kappa\,(y\,\mathsf{is}\,\iota)) & \\
& = & \lambda\kappa.\,[\![t]\!]\,[x := v^*]\,(\lambda y.\,\kappa\,(y\,\mathsf{is}\,\iota)) & \text{(by the IH)} \\
& = & (\lambda\kappa.\,[\![t]\!]\,(\lambda y.\,\kappa\,(y\,\mathsf{is}\,\iota)))\,[x := v^*] & \\
& = & [\![t\,\mathsf{is}\,\iota]\!]\,[x := v^*]. &
\end{array}$$

Case $s = t \,\mathsf{is}\, \star \,/ \star \to \star / \star$:

$$\begin{array}{rcl} & & [\![(t \,\mathsf{is}\, \star \,/ \star \to \star / \star)\,[x := v]]\!] \\ = & & [\![t\,[x := v] \,\mathsf{is}\, \star \,/ \star \to \star / \star]\!] \\ = & & \lambda\kappa.\, [\![t\,[x := v]]\!]\,(\lambda y.\, \kappa\,(y \,\mathsf{is}\, \star \to \star)) \\ = & & \lambda\kappa.\, [\![t]\!]\,[x := v^*]\,(\lambda y.\, \kappa\,(y \,\mathsf{is}\, \star \to \star)) \quad \text{(by the IH)} \\ = & & (\lambda\kappa.\, [\![t]\!]\,(\lambda y.\, \kappa\,(y \,\mathsf{is}\, \star \to \star)))\,[x := v^*] \\ = & & [\![t \,\mathsf{is}\, \star \,/ \star \to \star / \star]\!]\,[x := v^*]. \end{array}$$

Case $s = \mathsf{blame}\, p$:

$$\begin{array}{rcl} [\![(\mathsf{blame}\, p)\,[x := v]]\!] & = & [\![\mathsf{blame}\, p]\!] \\ & = & \lambda\kappa.\, \mathsf{blame}\, p \\ & = & (\lambda\kappa.\, \mathsf{blame}\, p)\,[x := v^*] \\ & = & [\![\mathsf{blame}\, p]\!]\,[x := v^*]. \end{array}$$

(2) By case analysis on v'.

Case $v' = y$: If $y = x$, then:

$$\begin{array}{rcl} (x\,[x := v])^* & = & v^* \\ & = & x\,[x := v^*] \\ & = & x^*\,[x := v^*]. \end{array}$$

Otherwise, if $y \neq x$, then:

$$\begin{array}{rcl} (y\,[x := v])^* & = & y^* \\ & = & y \\ & = & y\,[x := v^*] \\ & = & y^*\,[x := v^*]. \end{array}$$

Case $v' = c$:

$$\begin{array}{rcl} (c\,[x := v])^* & = & c^* \\ & = & c \\ & = & c\,[x := v^*] \\ & = & c^*\,[x := v^*]. \end{array}$$

Case $v' = \lambda y.\, t$: Without loss of generality, we can suppose that $y \neq x$ and $y \notin fv\,(v) \cup fv\,(v^*)$. Thus,

$$\begin{array}{rcl} ((\lambda y.\, t)\,[x := v])^* & = & (\lambda y.\, t\,[x := v])^* \\ & = & \lambda y.\, [\![t\,[x := v]]\!] \\ & = & \lambda y.\, [\![t]\!]\,[x := v^*] \quad \text{(by the IH)} \\ & = & (\lambda y.\, [\![t]\!])\,[x := v^*] \\ & = & (\lambda y.\, t)^*\,[x := v^*]. \end{array}$$

Case $v' = v'' : \iota \Rightarrow \star$:

$$\begin{array}{rcll}((v'' : \iota \Rightarrow \star)\,[x := v])^* & = & ((v''\,[x := v]) : \iota \Rightarrow \star)^* \\ & = & (v''\,[x := v])^* : \iota \Rightarrow \star \\ & = & (v''^*\,[x := v^*]) : \iota \Rightarrow \star & \text{(by the IH)} \\ & = & (v''^* : \iota \Rightarrow \star)\,[x := v^*] \\ & = & (v'' : \iota \Rightarrow \star)^*\,[x := v^*].\end{array}$$

Case $v' = v'' : \star/\star \to \star/\star \Rightarrow_p \star$:

$$\begin{array}{rcll} & ((v'' : \star/\star \to \star/\star \Rightarrow_p \star)\,[x := v])^* \\ = & (v''\,[x := v] : \star/\star \to \star/\star \Rightarrow_p \star)^* \\ = & (\lambda y.\,((v''\,[x := v])^*\,y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star \\ = & (\lambda y.\,(v''^*\,[x := v^*]\,y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star & \text{(by the IH)} \\ = & ((\lambda y.\,(v''^*\,y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)\,[x := v^*] \\ = & (v'' : \star/\star \to \star/\star \Rightarrow_p \star)^*\,[x := v^*].\end{array}$$

□

We show that the CPS-transformation result of term $E[s]$ is equivalent to the composition of the CPS-transformation results of E and s. Before proving it, we define a CPS transformation for evaluation contexts; note that it is also a transformation for pure evaluation contexts.

Definition 12 (CPS Transformation for Evaluation Contexts)**.** *We define CPS transformation for evaluation contexts as follows.*

$$\begin{array}{lcl}[\![[\,]]\!] & = & \lambda x.\,x \\ [\![op(\overline{v_i}^i, E, \overline{t_j}^j)]\!] & = & \lambda x.\,\lambda\kappa.\,[\![E]\!]\,x\,(\lambda y.\,[\![t_1]\!]\,(\lambda y_1.\,\ldots\,[\![t_n]\!]\,(\lambda y_n.\,\kappa\,op(\overline{v_i^*}^i, y, \overline{y_j}^j)))) \\ [\![E\;s]\!] & = & \lambda x.\,\lambda\kappa.\,[\![E]\!]\,x\,(\lambda y.\,[\![s]\!]\,(\lambda z.\,y\;z\;\kappa)) \\ [\![v\;E]\!] & = & \lambda x.\,\lambda\kappa.\,[\![v]\!]\,(\lambda y.\,[\![E]\!]\,x\,(\lambda z.\,y\;z\;\kappa)) \\ [\![\langle E\rangle]\!] & = & \lambda x.\,\lambda\kappa.\,\kappa\,([\![E]\!]\,x\,(\lambda y.\,y)) \\ [\![E : A \Rightarrow^p B]\!] & = & \lambda x.\,\lambda\kappa.\,[\![E]\!]\,x\,(\lambda y.\,\kappa\,(y : [\![A]\!] \Rightarrow^p [\![B]\!])) \\ [\![E : \iota \Rightarrow \star]\!] & = & \lambda x.\,\lambda\kappa.\,[\![E]\!]\,x\,(\lambda y.\,\kappa\,(y : \iota \Rightarrow \star)) \\ [\![E : \star/\star \to \star/\star \Rightarrow_p \star]\!] & = & \\ & & \lambda x.\,\lambda\kappa.\,[\![E]\!]\,x\,(\lambda y.\,\kappa\,((\lambda z.\,(y\;z) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)) \\ [\![E\;\mathsf{is}\;\iota]\!] & = & \lambda x.\,\lambda\kappa.\,[\![E]\!]\,x\,(\lambda y.\,\kappa\,(y\;\mathsf{is}\;\iota)) \\ [\![E\;\mathsf{is}\;\star/\star \to \star/\star]\!] & = & \lambda x.\,\lambda\kappa.\,[\![E]\!]\,x\,(\lambda y.\,\kappa\,(y\;\mathsf{is}\;\star \to \star))\end{array}$$

Lemma A.3.5. $[\![E]\!]\,[\![s]\!] \overset{\beta}{=} [\![E[s]]\!]$.

Proof. By structural induction on E. Note that $[\![s]\!]$ is a value.

Case $E = [\,]$:

$$[\![[\,]]\!]\,[\![s]\!] = (\lambda x.\,x)\,[\![s]\!] \overset{\beta}{=} [\![s]\!] = [\![[\,]\,[s]]\!].$$

Case $E = op(\overline{v_i}^i, E', \overline{t_j}^j)$: Similarly to the case for function applications.

Case $E = E'\ t$:

$$\begin{array}{rcl}
[\![E'\ t]\!]\,[\![s]\!] & = & (\lambda x.\,\lambda\kappa.\,[\![E']\!]\,x\,(\lambda y.\,[\![t]\!]\,(\lambda z.\,y\,z\,\kappa)))\,[\![s]\!] \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![E']\!]\,[\![s]\!]\,(\lambda y.\,[\![t]\!]\,(\lambda z.\,y\,z\,\kappa)) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![E'[s]]\!]\,(\lambda y.\,[\![t]\!]\,(\lambda z.\,y\,z\,\kappa)) \quad \text{(by the IH)} \\
& = & [\![E'[s]\ t]\!].
\end{array}$$

Case $E = v\ E'$:

$$\begin{array}{rcl}
[\![v\ E']\!]\,[\![s]\!] & = & (\lambda x.\,\lambda\kappa.\,[\![v]\!]\,(\lambda y.\,[\![E']\!]\,x\,(\lambda z.\,y\,z\,\kappa)))\,[\![s]\!] \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![v]\!]\,(\lambda y.\,[\![E']\!]\,[\![s]\!]\,(\lambda z.\,y\,z\,\kappa)) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![v]\!]\,(\lambda y.\,[\![E'[s]]\!]\,(\lambda z.\,y\,z\,\kappa)) \quad \text{(by the IH)} \\
& = & [\![v\ E'[s]]\!].
\end{array}$$

Case $E = \langle E'\rangle$:

$$\begin{array}{rcl}
[\![\langle E'\rangle]\!]\,[\![s]\!] & = & (\lambda x.\,\lambda\kappa.\,\kappa\,([\![E']\!]\,x\,(\lambda y.\,y)))\,[\![s]\!] \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,([\![E']\!]\,[\![s]\!]\,(\lambda y.\,y)) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,([\![E'[s]]\!]\,(\lambda y.\,y)) \quad \text{(by the IH)} \\
& = & [\![\langle E'[s]\rangle]\!]
\end{array}$$

Case $E = E' : A \Rightarrow^p B$:

$$\begin{array}{rcl}
[\![E' : A \Rightarrow^p B]\!]\,[\![s]\!] & = & (\lambda x.\,\lambda\kappa.\,[\![E']\!]\,x\,(\lambda y.\,\kappa\,(y : [\![A]\!] \Rightarrow^p [\![B]\!])))\,[\![s]\!] \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![E']\!]\,[\![s]\!]\,(\lambda y.\,\kappa\,(y : [\![A]\!] \Rightarrow^p [\![B]\!])) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![E'[s]]\!]\,(\lambda y.\,\kappa\,(y : [\![A]\!] \Rightarrow^p [\![B]\!])) \quad \text{(by the IH)} \\
& = & [\![E'[s] : A \Rightarrow^p B]\!].
\end{array}$$

Case $E = E' : \iota \Rightarrow \star$:

$$\begin{array}{rcl}
[\![E' : \iota \Rightarrow \star]\!]\,[\![s]\!] & = & (\lambda x.\,\lambda\kappa.\,[\![E']\!]\,x\,(\lambda y.\,\kappa\,(y : \iota \Rightarrow \star)))\,[\![s]\!] \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![E']\!]\,[\![s]\!]\,(\lambda y.\,\kappa\,(y : \iota \Rightarrow \star)) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![E'[s]]\!]\,(\lambda y.\,\kappa\,(y : \iota \Rightarrow \star)) \quad \text{(by the IH)} \\
& = & [\![E'[s] : \iota \Rightarrow \star]\!].
\end{array}$$

Case $E = E' : \star/\star \to \star/\star \Rightarrow_p \star$:

$$\begin{array}{rcl}
 & & [\![E' : \star/\star \to \star/\star \Rightarrow_p \star]\!]\,[\![s]\!] \\
& = & (\lambda x.\,\lambda\kappa.\,[\![E']\!]\,x\,(\lambda y.\,\kappa\,((\lambda z.\,(y\,z) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)))\,[\![s]\!] \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![E']\!]\,[\![s]\!]\,(\lambda y.\,\kappa\,((\lambda z.\,(y\,z) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,[\![E'[s]]\!]\,(\lambda y.\,\kappa\,((\lambda z.\,(y\,z) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)) \\
& = & [\![E'[s] : \star/\star \to \star/\star \Rightarrow \star]\!].
\end{array}$$

Case $E = E' \,\mathsf{is}\, \iota$:

$$\begin{array}{rcl}
[\![E' \,\mathsf{is}\, \iota]\!]\,[\![s]\!] & = & (\lambda x.\, \lambda\kappa.\, [\![E']\!]\, x\, (\lambda y.\, \kappa\, (y \,\mathsf{is}\, \iota)))\,[\![s]\!] \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![E']\!]\,[\![s]\!]\, (\lambda y.\, \kappa\, (y \,\mathsf{is}\, \iota)) \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![E'[s]]\!]\, (\lambda y.\, \kappa\, (y \,\mathsf{is}\, \iota)) \quad \text{(by the IH)} \\
& = & [\![E'[s] \,\mathsf{is}\, \iota]\!].
\end{array}$$

Case $E = E' \,\mathsf{is}\, \star \,/ \star \to \star / \star$:

$$\begin{array}{rcl}
[\![E' \,\mathsf{is}\, \star \,/ \star \to \star / \star]\!]\,[\![s]\!] & = & (\lambda x.\, \lambda\kappa.\, [\![E']\!]\, x\, (\lambda y.\, \kappa\, (y \,\mathsf{is}\, \star \to \star)))\,[\![s]\!] \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![E']\!]\,[\![s]\!]\, (\lambda y.\, \kappa\, (y \,\mathsf{is}\, \star \to \star)) \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![E'[s]]\!]\, (\lambda y.\, \kappa\, (y \,\mathsf{is}\, \star \to \star)) \quad \text{(by the IH)} \\
& = & [\![E'[s] \,\mathsf{is}\, \star \,/ \star \to \star / \star]\!].
\end{array}$$

□

Lemma A.3.6. $[\![F]\!]\,[\![s]\!] \overset{\beta\eta}{=} \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, [\![F]\!]\, (\lambda\kappa'.\, \kappa'\, x)\, \kappa)$ *where* $x \notin fv\,(F)$.

Proof. By structural induction on F.

Case $F = [\,]$:

$$\begin{array}{rcl}
[\![[\,]]\!]\,[\![s]\!] & \overset{\beta}{=} & [\![s]\!] \\
& \overset{\eta}{=} & \lambda\kappa.\, [\![s]\!]\, \kappa \\
& \overset{\eta}{=} & \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, \kappa\, x) \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, (\lambda\kappa'.\, \kappa'\, x)\, \kappa) \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![s]\!]\, (\lambda x.\, [\![[\,]]\!]\, (\lambda\kappa'.\, \kappa'\, x)\, \kappa).
\end{array}$$

Case $F = op(\overline{v_i}^{\,i}, F', \overline{t_j}^{\,j})$:

$$\begin{array}{rcl}
 & & [\![F]\!]\,[\![s]\!] \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![F']\!]\,[\![s]\!]\, (\lambda y.\, [\![t_1]\!]\, (\lambda y_1.\, \ldots\, [\![t_n]\!]\, (\lambda y_n.\, \kappa\, op(\overline{v_i^*}^{\,i}, y, \overline{y_j}^{\,j})))) \\
& \overset{\beta\eta}{=} & \lambda\kappa.\, (\lambda\kappa'.\, [\![s]\!]\, (\lambda z.\, [\![F']\!]\, (\lambda\kappa''.\, \kappa''\, z)\, \kappa')) \\
& & \qquad (\lambda y.\, [\![t_1]\!]\, (\lambda y_1.\, \ldots\, [\![t_n]\!]\, (\lambda y_n.\, \kappa\, op(\overline{v_i^*}^{\,i}, y, \overline{y_j}^{\,j})))) \quad \text{(by the IH)} \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![s]\!]\, (\lambda z.\, [\![F']\!]\, (\lambda\kappa''.\, \kappa''\, z)\, (\lambda y.\, [\![t_1]\!]\, (\lambda y_1.\, \ldots\, [\![t_n]\!]\, (\lambda y_n.\, \kappa\, op(\overline{v_i^*}^{\,i}, y, \overline{y_j}^{\,j})))))) \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![s]\!]\, (\lambda z.\, (\lambda x.\, \lambda\kappa'.\, [\![F']\!]\, x\, (\lambda y.\, [\![t_1]\!]\, (\lambda y_1.\, \ldots\, [\![t_n]\!]\, (\lambda y_n.\, \kappa'\, op(\overline{v_i^*}^{\,i}, y, \overline{y_j}^{\,j})))))) \\
& & \qquad\qquad (\lambda\kappa''.\, \kappa''\, z)\, \kappa) \\
& = & \lambda\kappa.\, [\![s]\!]\, (\lambda z.\, [\![F]\!]\, (\lambda\kappa''.\, \kappa''\, z)\, \kappa).
\end{array}$$

Case $F = F'\ t$:

$$\begin{array}{rcl}
[\![F]\!]\,[\![s]\!] & \overset{\beta}{=} & \lambda\kappa.\,[\![F']\!]\,[\![s]\!]\,(\lambda x.\,[\![t]\!]\,(\lambda y.\,x\ y\ \kappa)) \\
& \overset{\beta\eta}{=} & \lambda\kappa.\,(\lambda\kappa'.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa'))\,(\lambda x.\,[\![t]\!]\,(\lambda y.\,x\ y\ \kappa)) \quad \text{(by the IH)} \\
& \overset{\beta}{=} & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,(\lambda x.\,[\![t]\!]\,(\lambda y.\,x\ y\ \kappa))) \\
& \overset{\beta}{=} & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,(\lambda z.\,\lambda\kappa'.\,[\![F']\!]\,z\,(\lambda x.\,[\![t]\!]\,(\lambda y.\,x\ y\ \kappa')))\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa) \\
& = & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F]\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa).
\end{array}$$

Case $F = v\ F'$:

$$\begin{array}{rcl}
[\![F]\!]\,[\![s]\!] & \overset{\beta}{=} & \lambda\kappa.\,[\![F']\!]\,[\![s]\!]\,(\lambda x.\,v^*\ x\ \kappa) \\
& \overset{\beta\eta}{=} & \lambda\kappa.\,(\lambda\kappa'.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa'))\,(\lambda x.\,v^*\ x\ \kappa) \quad \text{(by the IH)} \\
& \overset{\beta}{=} & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,(\lambda x.\,v^*\ x\ \kappa)) \\
& \overset{\beta}{=} & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,(\lambda z.\,\lambda\kappa'.\,[\![F']\!]\,z\,(\lambda x.\,v^*\ x\ \kappa'))\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa) \\
& \overset{\beta}{=} & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F]\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa).
\end{array}$$

Case $F = F' : A \Rightarrow^p B$:

$$\begin{array}{rcl}
[\![F]\!]\,[\![s]\!] & \overset{\beta}{=} & \lambda\kappa.\,[\![F']\!]\,[\![s]\!]\,(\lambda x.\,\kappa\,(x\ :\ [\![A]\!] \Rightarrow^p\ [\![B]\!])) \\
& \overset{\beta\eta}{=} & \lambda\kappa.\,(\lambda\kappa'.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa'))\,(\lambda x.\,\kappa\,(x\ :\ [\![A]\!] \Rightarrow^p\ [\![B]\!])) \\
& & \quad \text{(by the IH)} \\
& \overset{\beta}{=} & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,(\lambda x.\,\kappa\,(x\ :\ [\![A]\!] \Rightarrow^p\ [\![B]\!]))) \\
& \overset{\beta}{=} & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,(\lambda z.\,\lambda\kappa'.\,[\![F']\!]\,z\,(\lambda x.\,\kappa'\,(x\ :\ [\![A]\!] \Rightarrow^p\ [\![B]\!])))\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa) \\
& = & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F]\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa).
\end{array}$$

Case $F = F' : \iota \Rightarrow \star$:

$$\begin{array}{rcl}
[\![F]\!]\,[\![s]\!] & \overset{\beta}{=} & \lambda\kappa.\,[\![F']\!]\,[\![s]\!]\,(\lambda x.\,\kappa\,(x\ :\ \iota \Rightarrow \star)) \\
& \overset{\beta\eta}{=} & \lambda\kappa.\,(\lambda\kappa'.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa'))\,(\lambda x.\,\kappa\,(x\ :\ \iota \Rightarrow \star)) \quad \text{(by the IH)} \\
& \overset{\beta}{=} & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,(\lambda x.\,\kappa\,(x\ :\ \iota \Rightarrow \star))) \\
& \overset{\beta}{=} & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,(\lambda z.\,\lambda\kappa'.\,[\![F']\!]\,z\,(\lambda x.\,\kappa'\,(x\ :\ \iota \Rightarrow \star)))\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa) \\
& = & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F]\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa).
\end{array}$$

Case $F = F' : \star/\star \to \star/\star \Rightarrow_p \star$:

$$
\begin{aligned}
& [\![F]\!]\,[\![s]\!] \\
\overset{\beta}{=}\ & \lambda\kappa.\,[\![F']\!]\,[\![s]\!]\,(\lambda x.\,\kappa\,((\lambda z.\,(y\ z)\ :\ (\star \to \star) \to \star \Rightarrow^p \star)\ :\ \star \to \star \Rightarrow \star)) \\
\overset{\beta\eta}{=}\ & \lambda\kappa.\,(\lambda\kappa'.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa'))\\
& \qquad (\lambda x.\,\kappa\,((\lambda z.\,(y\ z)\ :\ (\star \to \star) \to \star \Rightarrow^p \star)\ :\ \star \to \star \Rightarrow \star)) \\
& \qquad \text{(by the IH)} \\
\overset{\beta}{=}\ & \lambda\kappa.\,[\![s]\!] \\
& \quad (\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y) \\
& \qquad (\lambda x.\,\kappa\,((\lambda z.\,(y\ z)\ :\ (\star \to \star) \to \star \Rightarrow^p \star)\ :\ \star \to \star \Rightarrow \star))) \\
\overset{\beta}{=}\ & \lambda\kappa.\,[\![s]\!] \\
& \quad (\lambda y.\,(\lambda z.\,\lambda\kappa'.\,[\![F']\!]\ z \\
& \qquad\qquad (\lambda x.\,\kappa'\,((\lambda z.\,(y\ z)\ :\ (\star \to \star) \to \star \Rightarrow^p \star)\ :\ \star \to \star \Rightarrow \star))) \\
& \qquad (\lambda\kappa''.\,\kappa''\ y)\,\kappa) \\
=\ & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F]\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa).
\end{aligned}
$$

Case $F = F' \,\mathsf{is}\, \iota$:

$$
\begin{aligned}
[\![F]\!]\,[\![s]\!] \overset{\beta}{=}\ & \lambda\kappa.\,[\![F']\!]\,[\![s]\!]\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\,\iota)) \\
\overset{\beta\eta}{=}\ & \lambda\kappa.\,(\lambda\kappa'.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa'))\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\,\iota)) \quad \text{(by the IH)} \\
\overset{\beta}{=}\ & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\,\iota))) \\
\overset{\beta}{=}\ & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,(\lambda z.\,\lambda\kappa'.\,[\![F']\!]\ z\,(\lambda x.\,\kappa'\,(x\,\mathsf{is}\,\iota)))\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa) \\
=\ & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F]\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa).
\end{aligned}
$$

Case $F = F' \,\mathsf{is}\, \star\ /\star \to \star/\star$:

$$
\begin{aligned}
& [\![F]\!]\,[\![s]\!] \\
\overset{\beta}{=}\ & \lambda\kappa.\,[\![F']\!]\,[\![s]\!]\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\star \to \star)) \\
\overset{\beta\eta}{=}\ & \lambda\kappa.\,(\lambda\kappa'.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa'))\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\star \to \star)) \quad \text{(by the IH)} \\
\overset{\beta}{=}\ & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F']\!]\,(\lambda\kappa''.\,\kappa''\ y)\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\star \to \star))) \\
\overset{\beta}{=}\ & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,(\lambda z.\,\lambda\kappa'.\,[\![F']\!]\ z\,(\lambda x.\,\kappa'\,(x\,\mathsf{is}\star \to \star)))\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa) \\
=\ & \lambda\kappa.\,[\![s]\!]\,(\lambda y.\,[\![F]\!]\,(\lambda\kappa''.\,\kappa''\ y)\,\kappa).
\end{aligned}
$$

□

Lemma A.3.7. $[\![E]\!]\,[\![\mathsf{blame}\ p]\!] \overset{\beta\eta}{=} [\![\mathsf{blame}\ p]\!]$.

Proof. We first show

$$[\![E]\!]\,[\![\mathsf{blame}\ p]\!] \overset{\beta\eta}{=} \lambda\kappa.\,[\![\mathsf{blame}\ p]\!]\,(\lambda x.\,[\![E]\!]\,(\lambda\kappa'.\,\kappa'\ x)\,\kappa),$$

where $x \notin fv\,(E)$, by structural induction on E. We show only the case for $E = \langle E' \rangle$; other cases can be proven similarly to Lemma A.3.6. Suppose that $E = \langle E' \rangle$ for some

E'. Then,

$$\begin{array}{ll} & [\![\langle E' \rangle]\!]\,[\![\mathsf{blame}\ p]\!] \\ \overset{\beta}{=} & \lambda\kappa.\,\kappa\,([\![E']\!]\,[\![\mathsf{blame}\ p]\!]\,(\lambda x.\,x)) \\ \overset{\beta\eta}{=} & \lambda\kappa.\,\kappa\,((\lambda\kappa'.\,[\![\mathsf{blame}\ p]\!]\,(\lambda x.\,[\![E]\!]\,(\lambda\kappa''.\,\kappa''\,x)\,\kappa'))\,(\lambda x.\,x)) \quad \text{(by the IH)} \\ \overset{\beta}{=} & \lambda\kappa.\,\kappa\,(\mathsf{blame}\ p) \\ \overset{\beta}{=} & \lambda\kappa.\,\mathsf{blame}\ p \\ \overset{\beta}{=} & \lambda\kappa.\,[\![\mathsf{blame}\ p]\!]\,(\lambda x.\,[\![E]\!]\,(\lambda\kappa'.\,\kappa'\,x)\,\kappa). \end{array}$$

Next, we show $[\![E]\!]\,[\![\mathsf{blame}\ p]\!] \overset{\beta\eta}{=} [\![\mathsf{blame}\ p]\!]$. By the proof above,

$$\begin{array}{rcl} [\![E]\!]\,[\![\mathsf{blame}\ p]\!] & \overset{\beta\eta}{=} & \lambda\kappa.\,[\![\mathsf{blame}\ p]\!]\,(\lambda x.\,[\![E]\!]\,(\lambda\kappa'.\,\kappa'\,x)\,\kappa) \\ & \overset{\beta}{=} & \lambda\kappa.\,\mathsf{blame}\ p \\ & = & [\![\mathsf{blame}\ p]\!]. \end{array}$$

□

Lemma A.3.8. $\mathbb{e} : (\mathbb{A} \to \mathbb{B}) \to \mathbb{C} \Rightarrow^{p} (\star \to \star) \to \star \Rightarrow^{p} \star \overset{\beta\omega\xi\upsilon}{=} \mathbb{e} : (\mathbb{A} \to \mathbb{B}) \to \mathbb{C} \Rightarrow^{p} \star$

Proof.

$$\begin{array}{ll} & \mathbb{e} : (\mathbb{A} \to \mathbb{B}) \to \mathbb{C} \Rightarrow^{p} (\star \to \star) \to \star \Rightarrow^{p} \star \\ \overset{\omega}{=} & (\lambda x.\,x : (\mathbb{A} \to \mathbb{B}) \to \mathbb{C} \Rightarrow^{p} (\star \to \star) \to \star \Rightarrow^{p} \star)\,\mathbb{e} \\ \overset{\beta}{=} & (\lambda x.\,(\lambda y.\,x\,(y : \star \to \star \Rightarrow^{\bar{p}} \mathbb{A} \to \mathbb{B}) : \mathbb{C} \Rightarrow^{p} \star) : (\star \to \star) \to \star \Rightarrow^{p} \star)\,\mathbb{e} \\ \overset{\beta}{=} & (\lambda x.\,(\lambda y.\,x\,(y : \star \to \star \Rightarrow^{\bar{p}} \mathbb{A} \to \mathbb{B}) : \mathbb{C} \Rightarrow^{p} \star) : (\star \to \star) \to \star \Rightarrow^{p} \star \to \star \Rightarrow \star)\,\mathbb{e} \\ \overset{\beta}{=} & (\lambda x.\,(\lambda z.\,((\lambda y.\,x\,(y : \star \to \star \Rightarrow^{\bar{p}} \mathbb{A} \to \mathbb{B}) : \mathbb{C} \Rightarrow^{p} \star) \\ & \qquad (z : \star \Rightarrow^{\bar{p}} \star \to \star)) : \star \Rightarrow^{p} \star) : \star \to \star \Rightarrow \star)\,\mathbb{e} \\ \overset{\omega}{=} & (\lambda x.\,(\lambda z.\,x\,(z : \star \Rightarrow^{\bar{p}} \star \to \star \Rightarrow^{\bar{p}} \mathbb{A} \to \mathbb{B}) : \mathbb{C} \Rightarrow^{p} \star \Rightarrow^{p} \star) : \star \to \star \Rightarrow \star)\,\mathbb{e} \\ \overset{\upsilon}{=} & (\lambda x.\,(\lambda z.\,x\,(z : \star \Rightarrow^{\bar{p}} \star \to \star \Rightarrow^{\bar{p}} \mathbb{A} \to \mathbb{B}) : \mathbb{C} \Rightarrow^{p} \star) : \star \to \star \Rightarrow \star)\,\mathbb{e} \\ \overset{\xi}{=} & (\lambda x.\,(\lambda z.\,x\,(z : \star \Rightarrow^{\bar{p}} \mathbb{A} \to \mathbb{B}) : \mathbb{C} \Rightarrow^{p} \star) : \star \to \star \Rightarrow \star)\,\mathbb{e} \\ \overset{\beta}{=} & (\lambda x.\,x : (\mathbb{A} \to \mathbb{B}) \to \mathbb{C} \Rightarrow^{p} \star \to \star \Rightarrow \star)\,\mathbb{e} \\ \overset{\beta}{=} & (\lambda x.\,x : (\mathbb{A} \to \mathbb{B}) \to \mathbb{C} \Rightarrow^{p} \star)\,\mathbb{e} \\ \overset{\omega}{=} & \mathbb{e} : (\mathbb{A} \to \mathbb{B}) \to \mathbb{C} \Rightarrow^{p} \star \end{array}$$

□

Lemma A.3.9. $\mathbb{e} : (\star \to \star) \to \star \Rightarrow^{p} \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C} \overset{\beta\omega\upsilon}{=} \mathbb{e} : (\star \to \star) \to \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C}$

Proof.

$$
\begin{array}{ll}
 & \mathbb{e} : (\star \to \star) \to \star \Rightarrow^{p} \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C} \\
\overset{\omega}{=} & (\lambda x.\, x : (\star \to \star) \to \star \Rightarrow^{p} \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C})\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, x : (\star \to \star) \to \star \Rightarrow^{p} \star \to \star \Rightarrow \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C})\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda y.\, x\, (y : \star \Rightarrow^{\bar{p}} \star \to \star) : \star \Rightarrow^{p} \star) : \star \to \star \Rightarrow \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C})\, \mathbb{e} \\
\overset{\upsilon}{=} & (\lambda x.\, (\lambda y.\, x\, (y : \star \Rightarrow^{\bar{p}} \star \to \star)) : \star \to \star \Rightarrow \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C})\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda y.\, x\, (y : \star \Rightarrow^{\bar{p}} \star \to \star)) : \star \to \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C})\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda z.\, (\lambda y.\, x\, (y : \star \Rightarrow^{\bar{p}} \star \to \star))\, (z : (\mathbb{A} \to \mathbb{B}) \Rightarrow^{\bar{q}} \star) : \star \Rightarrow^{q} \mathbb{C}))\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda z.\, x\, (z : (\mathbb{A} \to \mathbb{B}) \Rightarrow^{\bar{q}} \star \Rightarrow^{\bar{p}} \star \to \star) : \star \Rightarrow^{q} \mathbb{C}))\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda z.\, x\, (z : (\mathbb{A} \to \mathbb{B}) \Rightarrow^{\bar{q}} \star \to \star \Rightarrow \star \Rightarrow^{\bar{p}} \star \to \star) : \star \Rightarrow^{q} \mathbb{C}))\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda z.\, x \\
 & \quad ((\lambda x'.\, z\, (x' : \star \Rightarrow^{q} \mathbb{A}) : \mathbb{B} \Rightarrow^{\bar{q}} \star) : \star \to \star \Rightarrow \star \Rightarrow^{\bar{p}} \star \to \star) : \star \Rightarrow^{q} \mathbb{C}))\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda z.\, x \\
 & \quad ((\lambda x'.\, z\, (x' : \star \Rightarrow^{q} \mathbb{A}) : \mathbb{B} \Rightarrow^{\bar{q}} \star) : \star \to \star \Rightarrow^{\bar{p}} \star \to \star) : \star \Rightarrow^{q} \mathbb{C}))\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda z.\, x \\
 & \quad (\lambda y'.\, (\lambda x'.\, z\, (x' : \star \Rightarrow^{q} \mathbb{A}) : \mathbb{B} \Rightarrow^{\bar{q}} \star)\, (y' : \star \Rightarrow^{p} \star) : \star \Rightarrow^{\bar{p}} \star) : \star \Rightarrow^{q} \mathbb{C}))\, \mathbb{e} \\
\overset{\upsilon}{=} & (\lambda x.\, (\lambda z.\, x\, (\lambda y'.\, (\lambda x'.\, z\, (x' : \star \Rightarrow^{q} \mathbb{A}) : \mathbb{B} \Rightarrow^{\bar{q}} \star)\, y') : \star \Rightarrow^{q} \mathbb{C}))\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda z.\, x\, (\lambda y'.\, z\, (y' : \star \Rightarrow^{q} \mathbb{A}) : \mathbb{B} \Rightarrow^{\bar{q}} \star) : \star \Rightarrow^{q} \mathbb{C}))\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, (\lambda z.\, x\, (z : \mathbb{A} \to \mathbb{B} \Rightarrow^{\bar{q}} \star \to \star) : \star \Rightarrow^{q} \mathbb{C}))\, \mathbb{e} \\
\overset{\beta}{=} & (\lambda x.\, x : (\star \to \star) \to \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C})\, \mathbb{e} \\
\overset{\omega}{=} & \mathbb{e} : (\star \to \star) \to \star \Rightarrow^{q} (\mathbb{A} \to \mathbb{B}) \to \mathbb{C}
\end{array}
$$

□

Lemma A.3.10. *If $s \longrightarrow t$, then $[\![s]\!] \overset{\beta\eta\omega\xi\upsilon}{=} [\![t]\!]$.*

Proof. By case analysis on the reduction rule applied to s.

Case (R_Op): We are given $op(\overline{v_i}^{\,i}) \longrightarrow \zeta\,(op, \overline{v_i}^{\,i})$ for some op and $\overline{v_i}^{\,i}$. We show that $[\![op(\overline{v_i}^{\,i})]\!] \overset{\beta}{=} [\![\zeta\,(op, \overline{v_i}^{\,i})]\!]$. By definition, $[\![op(\overline{v_i}^{\,i})]\!] \overset{\beta}{=} (\lambda\kappa.\, [\![v_1]\!]\, (\lambda x_1.\, \ldots\, [\![v_n]\!]\, (\lambda x_n.\, \kappa\, op(\overline{x_i}^{\,i}))))$. Since $op(\overline{v_i}^{\,i})$ takes a step, all values $\overline{v_i}^{\,i}$ are constants. Thus, for each v_i, $[\![v_i]\!] = \lambda\kappa'.\, \kappa'\, v_i$, and so

$$
\begin{array}{rcl}
[\![op(\overline{v_i}^{\,i})]\!] & \overset{\beta}{=} & \lambda\kappa.\, \kappa\, op(\overline{v_i}^{\,i}) \\
 & \overset{\beta}{=} & \lambda\kappa.\, \kappa\, \zeta\,(op, \overline{v_i}^{\,i}) \\
 & \overset{\beta}{=} & [\![\zeta\,(op, \overline{v_i}^{\,i})]\!] \quad \text{(since } \zeta\,(op, \overline{v_i}^{\,i}) \text{ is a constant).}
\end{array}
$$

Case (R_BETA): We are given $(\lambda x.\, u)\, v \longrightarrow u\,[x := v]$. We show that $[\![(\lambda x.\, u)\, v]\!] \overset{\beta}{=} [\![u\,[x := v]]\!]$. Here,

$$\begin{array}{rcl}
[\![(\lambda x.\, u)\, v]\!] & = & \lambda\kappa.\, [\![\lambda x.\, u]\!]\, (\lambda y.\, [\![v]\!]\, (\lambda z.\, y\, z\, \kappa)) \\
& = & \lambda\kappa.\, (\lambda\kappa'.\, \kappa'\, (\lambda x.\, [\![u]\!]))\, (\lambda y.\, [\![v]\!]\, (\lambda z.\, y\, z\, \kappa)) \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![v]\!]\, (\lambda z.\, (\lambda x.\, [\![u]\!])\, z\, \kappa) \\
& = & \lambda\kappa.\, (\lambda\kappa''.\, \kappa''\, v^*)\, (\lambda z.\, (\lambda x.\, [\![u]\!])\, z\, \kappa) \\
& \overset{\beta}{=} & \lambda\kappa.\, (\lambda x.\, [\![u]\!])\, v^*\, \kappa \\
& \overset{\beta}{=} & \lambda\kappa.\, [\![u]\!]\, [x := v^*]\, \kappa \\
& = & \lambda\kappa.\, [\![u\,[x := v]]\!]\, \kappa \quad \text{(by Lemma A.3.4)} \\
& \overset{\eta}{=} & [\![u\,[x := v]]\!].
\end{array}$$

Case (R_WRAP): We are given

$$v : A/\alpha \to B/\beta \Rightarrow^p A'/\alpha' \to B'/\beta' \longrightarrow \lambda x.\, \mathcal{S}k.\, (\langle (k\, ((v\, (x : A' \Rightarrow^{\bar{p}} A)) : B \Rightarrow^p B')) : \alpha' \Rightarrow^{\bar{p}} \alpha \rangle : \beta \Rightarrow^p \beta').$$

$$\begin{array}{rcl}
& & [\![v : A/\alpha \to B/\beta \Rightarrow^p A'/\alpha' \to B'/\beta']\!] \\
& = & \lambda\kappa.\, [\![v]\!]\, (\lambda x.\, \kappa\, (x : [\![A/\alpha \to B/\beta]\!] \Rightarrow^p [\![A'/\alpha' \to B'/\beta']\!])) \\
& \overset{\beta}{=} & \lambda\kappa.\, \kappa\, (v^* : ([\![A]\!] \to ([\![B]\!] \to [\![\alpha]\!]) \to [\![\beta]\!]) \Rightarrow^p [\![[\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]]\!]) \\
& \overset{\beta}{=} & \lambda\kappa.\, \kappa\, (\lambda x. \\
& & \quad (v^*\, (x : [\![A']\!] \Rightarrow^{\bar{p}} [\![A]\!])) : ([\![B]\!] \to [\![\alpha]\!]) \to [\![\beta]\!] \Rightarrow^p ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \\
& \overset{\omega}{=} & \lambda\kappa.\, \kappa\, (\lambda x. \\
& & \quad (\lambda y.\, y : ([\![B]\!] \to [\![\alpha]\!]) \to [\![\beta]\!] \Rightarrow^p ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \\
& & \quad (v^*\, (x : [\![A']\!] \Rightarrow^{\bar{p}} [\![A]\!]))) \\
& \overset{\beta}{=} & \lambda\kappa.\, \kappa\, (\lambda x. \\
& & \quad (\lambda y.\, \lambda\kappa'.\, (y\, (\kappa' : [\![B']\!] \to [\![\alpha']\!] \Rightarrow^{\bar{p}} [\![B]\!] \to [\![\alpha]\!])) : [\![\beta]\!] \Rightarrow^p [\![\beta']\!]) \\
& & \quad (v^*\, (x : [\![A']\!] \Rightarrow^{\bar{p}} [\![A]\!]))) \\
& \overset{\beta}{=} & \lambda\kappa.\, \kappa\, (\lambda x. \\
& & \quad (\lambda y.\, \lambda\kappa'.\, (y\, (\lambda z.\, (\kappa'\, (z : [\![B]\!] \Rightarrow^p [\![B']\!])) : [\![\alpha']\!] \Rightarrow^{\bar{p}} [\![\alpha]\!])) : [\![\beta]\!] \Rightarrow^p [\![\beta']\!]) \\
& & \quad (v^*\, (x : [\![A']\!] \Rightarrow^{\bar{p}} [\![A]\!]))) \\
& \overset{\omega}{=} & \lambda\kappa.\, \kappa\, (\lambda x. \\
& & \quad \lambda\kappa'.\, (v^*\, (x : [\![A']\!] \Rightarrow^{\bar{p}} [\![A]\!]) \\
& & \qquad (\lambda z.\, (\kappa'\, (z : [\![B]\!] \Rightarrow^p [\![B']\!])) : [\![\alpha']\!] \Rightarrow^{\bar{p}} [\![\alpha]\!])) : [\![\beta]\!] \Rightarrow^p [\![\beta']\!])
\end{array}$$

Moreover,

$$\begin{array}{rcl}
& & [\![\lambda x.\, \mathcal{S}k.\, (\langle (k\, ((v\, (x : A' \Rightarrow^{\bar{p}} A)) : B \Rightarrow^p B')) : \alpha' \Rightarrow^{\bar{p}} \alpha \rangle : \beta \Rightarrow^p \beta')]\!] \\
& = & \lambda\kappa.\, \kappa\, (\lambda x. \\
& & \quad [\![\mathcal{S}k.\, (\langle (k\, ((v\, (x : A' \Rightarrow^{\bar{p}} A)) : B \Rightarrow^p B')) : \alpha' \Rightarrow^{\bar{p}} \alpha \rangle : \beta \Rightarrow^p \beta')]\!]) \\
& = & \lambda\kappa.\, \kappa\, (\lambda x. \\
& & \quad \lambda\kappa'.\, ([\![\langle (k\, ((v\, (x : A' \Rightarrow^{\bar{p}} A)) : B \Rightarrow^p B')) : \alpha' \Rightarrow^{\bar{p}} \alpha \rangle : \beta \Rightarrow^p \beta']\!] \\
& & \qquad (\lambda y.\, y))\, [k := \lambda z.\, \lambda\kappa''.\, \kappa''\, (\kappa'\, z)])
\end{array}$$

Here,

$$
\begin{array}{rl}
 & [\![\langle (k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B'))\,:\,\alpha' \Rightarrow^{\bar{p}}\ \alpha\rangle\,:\,\beta \Rightarrow^{p}\ \beta']\!]\,(\lambda y.\,y) \\
= & (\lambda \kappa'''.\,[\![\langle (k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B'))\,:\,\alpha' \Rightarrow^{\bar{p}}\ \alpha\rangle]\!] \\
 & \qquad (\lambda x'.\,\kappa'''\,(x'\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!]))) \\
 & (\lambda y.\,y) \\
\overset{\beta}{=} & [\![\langle (k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B'))\,:\,\alpha' \Rightarrow^{\bar{p}}\ \alpha\rangle]\!] \\
 & (\lambda x'.\,(\lambda y.\,y)\,(x'\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!])) \\
\overset{\omega}{=} & [\![\langle (k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B'))\,:\,\alpha' \Rightarrow^{\bar{p}}\ \alpha\rangle]\!] \\
 & (\lambda x'.\,x'\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!]) \\
= & (\lambda \kappa'''.\,\kappa'''\,([\![(k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B'))\,:\,\alpha' \Rightarrow^{\bar{p}}\ \alpha]\!]\,(\lambda y.\,y))) \\
 & (\lambda x'.\,x'\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!]) \\
\overset{\beta}{=} & (\lambda x'.\,x'\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!]) \\
 & ([\![(k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B'))\,:\,\alpha' \Rightarrow^{\bar{p}}\ \alpha]\!]\,(\lambda y.\,y)) \\
\overset{\omega}{=} & ([\![(k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B'))\,:\,\alpha' \Rightarrow^{\bar{p}}\ \alpha]\!]\,(\lambda y.\,y))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
= & ((\lambda \kappa'''.\,[\![k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B')]\!]\,(\lambda x'.\,\kappa'''\,(x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!]))) \\
 & (\lambda y.\,y))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
\overset{\beta}{=} & ([\![k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B')]\!] \\
 & (\lambda x'.\,(\lambda y.\,y)\,(x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!])))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
\overset{\omega}{=} & ([\![k\,((v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B')]\!] \\
 & (\lambda x'.\,x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!]))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
= & ((\lambda \kappa'''.\,[\![k]\!]\,(\lambda x''.\,[\![(v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B']\!]\,(\lambda y''.\,x''\,y''\,\kappa'''))) \\
 & (\lambda x'.\,x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!]))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
\overset{\beta}{=} & ([\![k]\!]\,(\lambda x''.\,[\![(v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B']\!] \\
 & \qquad (\lambda y''.\,x''\,y''\,(\lambda x'.\,x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!])))))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
\overset{\beta}{=} & ([\![(v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A))\,:\,B \Rightarrow^{p}\ B']\!] \\
 & (\lambda y''.\,k\,y''\,(\lambda x'.\,x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!])))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
= & ((\lambda \kappa'''.\,[\![v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A)]\!]\,(\lambda x''.\,\kappa'''\,(x''\,:\,[\![B]\!] \Rightarrow^{p}\ [\![B']\!]))) \\
 & (\lambda y''.\,k\,y''\,(\lambda x'.\,x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!])))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
\overset{\beta}{=} & ([\![v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A)]\!] \\
 & (\lambda x''.\,(\lambda y''.\,k\,y''\,(\lambda x'.\,x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!])) \\
 & \qquad (x''\,:\,[\![B]\!] \Rightarrow^{p}\ [\![B']\!])))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
\overset{\omega}{=} & ([\![v\,(x\,:\,A' \Rightarrow^{\bar{p}}\ A)]\!] \\
 & (\lambda x''.\,k\,(x''\,:\,[\![B]\!] \Rightarrow^{p}\ [\![B']\!])\,(\lambda x'.\,x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!])))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
= & ((\lambda \kappa'''.\,[\![v]\!]\,(\lambda x''.\,[\![x\,:\,A' \Rightarrow^{\bar{p}}\ A]\!]\,(\lambda y''.\,x''\,y''\,\kappa'''))) \\
 & (\lambda x''.\,k\,(x''\,:\,[\![B]\!] \Rightarrow^{p}\ [\![B']\!])\,(\lambda x'.\,x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!])))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!] \\
\overset{\beta}{=} & ([\![v]\!]\,(\lambda x'''.\,[\![x\,:\,A' \Rightarrow^{\bar{p}}\ A]\!]\,(\lambda y''.\,x'''\,y'' \\
 & (\lambda x''.\,k\,(x''\,:\,[\![B]\!] \Rightarrow^{p}\ [\![B']\!])\,(\lambda x'.\,x'\,:\,[\![\alpha']\!] \Rightarrow^{\bar{p}}\ [\![\alpha]\!])))))\,:\,[\![\beta]\!] \Rightarrow^{p}\ [\![\beta']\!]
\end{array}
$$

$$
\begin{array}{rl}
\overset{\beta}{=} & (\llbracket x \,:\, A' \Rightarrow^{\bar{p}} \; A \rrbracket \, (\lambda y''.\, v^*\, y'' \\
& \quad (\lambda x''.\, k\, (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket)\, (\lambda x'.\, x' \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket)))) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket \\
= & ((\lambda \kappa'''.\, \llbracket x \rrbracket \, (\lambda x'''.\, \kappa'''\, (x''' \,:\, \llbracket A' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket A \rrbracket)))\, (\lambda y''.\, v^*\, y'' \\
& \quad (\lambda x''.\, k\, (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket)\, (\lambda x'.\, x' \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket)))) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket \\
\overset{\beta}{=} & ((\llbracket x \rrbracket \, (\lambda x'''.\, (\lambda y''.\, v^*\, y''\, (\lambda x''.\, k\, (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket)\, (\lambda x'.\, x' \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket))) \\
& \qquad\qquad (x''' \,:\, \llbracket A' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket A \rrbracket)))) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket \\
\overset{\beta}{=} & ((\lambda y''.\, v^*\, y''\, (\lambda x''.\, k\, (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket)\, (\lambda x'.\, x' \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket))) \\
& \quad (x \,:\, \llbracket A' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket A \rrbracket)) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket \\
\overset{\omega}{=} & (v^*\, (x \,:\, \llbracket A' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket A \rrbracket) \\
& \quad (\lambda x''.\, k\, (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket)\, (\lambda x'.\, x' \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket))) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket.
\end{array}
$$

Thus,

$$
\begin{array}{rl}
& (\llbracket \langle (k\, ((v\, (x \,:\, A' \Rightarrow^{\bar{p}} \; A)) \,:\, B \Rightarrow^{p} \; B')) \,:\, \alpha' \Rightarrow^{\bar{p}} \; \alpha \rangle \,:\, \beta \Rightarrow^{p} \; \beta' \rrbracket \\
& \quad (\lambda y.\, y))\, [k := \lambda z.\, \lambda \kappa''.\, \kappa''\, (\kappa'\, z)] \\
\overset{\beta\omega}{=} & (v^*\, (x \,:\, \llbracket A' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket A \rrbracket) \\
& \quad (\lambda x''.\, (\lambda z.\, \lambda \kappa''.\, \kappa''\, (\kappa'\, z)) \\
& \qquad\quad (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket) \\
& \qquad\quad (\lambda x'.\, x' \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket))) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket \\
\overset{\omega}{=} & (v^*\, (x \,:\, \llbracket A' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket A \rrbracket) \\
& \quad (\lambda x''.\, (\lambda \kappa''.\, \kappa''\, (\kappa'\, (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket))) \\
& \qquad\quad (\lambda x'.\, x' \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket))) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket \\
\overset{\beta}{=} & (v^*\, (x \,:\, \llbracket A' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket A \rrbracket) \\
& \quad (\lambda x''.\, (\lambda x'.\, x' \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket)\, (\kappa'\, (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket)))) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket \\
\overset{\omega}{=} & (v^*\, (x \,:\, \llbracket A' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket A \rrbracket) \\
& \quad (\lambda x''.\, (\kappa'\, (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket)) \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket)) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket
\end{array}
$$

Therefore,

$$
\begin{array}{rl}
& \llbracket \lambda x.\, \mathcal{S} k.\, (\langle (k\, ((v\, (x \,:\, A' \Rightarrow^{\bar{p}} \; A)) \,:\, B \Rightarrow^{p} \; B')) \,:\, \alpha' \Rightarrow^{\bar{p}} \; \alpha \rangle \,:\, \beta \Rightarrow^{p} \; \beta') \rrbracket \\
\overset{\beta\omega}{=} & \lambda \kappa.\, \kappa\, (\lambda x.\, \lambda \kappa'. \\
& \quad (v^*\, (x \,:\, \llbracket A' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket A \rrbracket) \\
& \qquad (\lambda x''.\, (\kappa'\, (x'' \,:\, \llbracket B \rrbracket \Rightarrow^{p} \; \llbracket B' \rrbracket)) \,:\, \llbracket \alpha' \rrbracket \Rightarrow^{\bar{p}} \; \llbracket \alpha \rrbracket)) \,:\, \llbracket \beta \rrbracket \Rightarrow^{p} \; \llbracket \beta' \rrbracket) \\
\overset{\beta\omega}{=} & \llbracket v \,:\, A/\alpha \to B/\beta \Rightarrow^{p} \; A'/\alpha' \to B'/\beta' \rrbracket
\end{array}
$$

Case (R_RESET): We are given $\langle v \rangle \longrightarrow v$ for some v. We show that $\llbracket \langle v \rangle \rrbracket \overset{\beta}{=} \llbracket v \rrbracket$. Here,

$$
\begin{array}{rcl}
\llbracket \langle v \rangle \rrbracket & \overset{\beta}{=} & \lambda \kappa.\, \kappa\, (\llbracket v \rrbracket\, (\lambda x.\, x)) \\
& \overset{\beta}{=} & \lambda \kappa.\, \kappa\, v^* \\
& \overset{\beta}{=} & \llbracket v \rrbracket.
\end{array}
$$

Case (R_SHIFT): We are given $\langle F[\mathcal{S}k.\, s]\rangle \longrightarrow \langle s\,[k := \lambda x.\, \langle F[x]\rangle]\rangle$ where $x \notin fv\,(F)$. We show that $[\![\langle F[\mathcal{S}k.\, s]\rangle]\!] \overset{\beta\eta}{=} [\![\langle s\,[k := \lambda x.\, \langle F[x]\rangle]\rangle]\!]$. Here,

$$
\begin{aligned}
& [\![\langle F[\mathcal{S}k.\, s]\rangle]\!] \\
= \;& \lambda\kappa.\, \kappa\,([\![F[\mathcal{S}k.\, s]]\!]\,(\lambda y.\, y)) \\
\overset{\beta}{=} \;& \lambda\kappa.\, \kappa\,([\![F]\!]\,[\![\mathcal{S}k.\, s]\!]\,(\lambda y.\, y)) \quad \text{(by Lemma A.3.5)} \\
\overset{\beta\eta}{=} \;& \lambda\kappa.\, \kappa\,((\lambda\kappa'.\, [\![\mathcal{S}k.\, s]\!]\,(\lambda x.\, [\![F]\!]\,(\lambda\kappa''.\, \kappa''\, x)\,\kappa'))\,(\lambda y.\, y)) \quad \text{(by Lemma A.3.6)} \\
\overset{\beta}{=} \;& \lambda\kappa.\, \kappa\,([\![\mathcal{S}k.\, s]\!]\,(\lambda x.\, [\![F]\!]\,(\lambda\kappa''.\, \kappa''\, x)\,(\lambda y.\, y))) \\
= \;& \lambda\kappa.\, \kappa \\
& \quad ((\lambda\kappa'''.\, [\![s]\!]\,[k := \lambda z.\, \lambda\kappa''''.\, \kappa''''\,(\kappa'''\, z)]\,(\lambda z.\, z))\,(\lambda x.\, [\![F]\!]\,(\lambda\kappa''.\, \kappa''\, x)\,(\lambda y.\, y))) \\
\overset{\beta}{=} \;& \lambda\kappa.\, \kappa\,([\![s]\!]\,[k := \lambda z.\, \lambda\kappa''''.\, \kappa''''\,((\lambda x.\, [\![F]\!]\,(\lambda\kappa''.\, \kappa''\, x)\,(\lambda y.\, y))\, z)]\,(\lambda z.\, z)) \\
\overset{\beta}{=} \;& \lambda\kappa.\, \kappa\,([\![s]\!]\,[k := \lambda z.\, \lambda\kappa''''.\, \kappa''''\,([\![F]\!]\,(\lambda\kappa''.\, \kappa''\, z)\,(\lambda y.\, y))]\,(\lambda z.\, z)) \\
= \;& \lambda\kappa.\, \kappa\,([\![s]\!]\,[k := \lambda z.\, \lambda\kappa''''.\, \kappa''''\,([\![F]\!]\,[\![z]\!]\,(\lambda y.\, y))]\,(\lambda z.\, z)) \\
\overset{\beta}{=} \;& \lambda\kappa.\, \kappa\,([\![s]\!]\,[k := \lambda z.\, \lambda\kappa''''.\, \kappa''''\,([\![F[z]]\!]\,(\lambda y.\, y))]\,(\lambda z.\, z)) \quad \text{(by Lemma A.3.5)} \\
= \;& \lambda\kappa.\, \kappa\,([\![s]\!]\,[k := \lambda z.\, [\![\langle F[z]\rangle]\!]]\,(\lambda z.\, z)) \\
= \;& \lambda\kappa.\, \kappa\,([\![s]\!]\,[k := (\lambda z.\, \langle F[z]\rangle)^*]\,(\lambda z.\, z)) \\
= \;& \lambda\kappa.\, \kappa\,([\![s\,[k := \lambda z.\, \langle F[z]\rangle]]\!]\,(\lambda z.\, z)) \quad \text{(by Lemma A.3.4)} \\
= \;& [\![\langle s\,[k := \lambda z.\, \langle F[z]\rangle]\rangle]\!].
\end{aligned}
$$

Case (R_BASE): We are given $v : \iota \Rightarrow^p \iota \longrightarrow v$ for some v and ι. We show that $[\![v : \iota \Rightarrow^p \iota]\!] \overset{\beta}{=} [\![v]\!]$. Here,

$$
\begin{aligned}
[\![v : \iota \Rightarrow^p \iota]\!] &\overset{\beta}{=} \lambda\kappa.\, \kappa\,(v^* : \iota \Rightarrow^p \iota) \\
&\overset{\beta}{=} \lambda\kappa.\, \kappa\, v^* \\
&= [\![v]\!].
\end{aligned}
$$

Case (R_DYN): We are given $v : \star \Rightarrow^p \star \longrightarrow v$ for some v. We show that $[\![v : \star \Rightarrow^p \star]\!] \overset{v}{=} [\![v]\!]$. Here,

$$
\begin{aligned}
[\![v : \star \Rightarrow^p \star]\!] &= \lambda\kappa.\, \kappa\,(v^* : \star \Rightarrow^p \star) \\
&\overset{v}{=} \lambda\kappa.\, \kappa\, v^* \\
&= [\![v]\!].
\end{aligned}
$$

Case (R_GROUND): We are given $v : A \Rightarrow^p \star \longrightarrow v : A \Rightarrow^p G \Rightarrow_p \star$ for some v, A, p and G such that $A \sim G$ and $A \neq \star$. We show that $[\![v : A \Rightarrow^p \star]\!] \overset{\beta\omega\xi v}{=} [\![v : A \Rightarrow^p G \Rightarrow_p \star]\!]$. By case analysis on A.

Case $A = \iota$: Then, we have $G = \iota$ by Lemma A.3.2. On one hand:

$$
\begin{aligned}
[\![v : \iota \Rightarrow^p \star]\!] &\overset{\beta}{=} \lambda\kappa.\, \kappa\,(v^* : \iota \Rightarrow^p \star) \\
&\overset{\beta}{=} \lambda\kappa.\, \kappa\,(v^* : \iota \Rightarrow^p \iota \Rightarrow \star) \\
&\overset{\beta}{=} \lambda\kappa.\, \kappa\,(v^* : \iota \Rightarrow \star).
\end{aligned}
$$

On the other hand:

$$\begin{aligned}
[\![v : \iota \Rightarrow^p \iota \Rightarrow_p \star]\!] &= \lambda\kappa.\, [\![v : \iota \Rightarrow^p \iota]\!]\, (\lambda x.\, \kappa\, (x : \iota \Rightarrow \star)) \\
&\overset{\beta}{=} \lambda\kappa.\, (\lambda x.\, \kappa\, (x : \iota \Rightarrow \star))\, (v^* : \iota \Rightarrow^p \iota) \\
&\overset{\beta}{=} \lambda\kappa.\, \kappa\, (v^* : \iota \Rightarrow \star).
\end{aligned}$$

Therefore, we finish.

Case $A = A'/\alpha' \to B'/\beta'$: Then, we have $G = \star/\star \to \star/\star$ by Lemma A.3.3. On one hand:

$$\begin{aligned}
& [\![v : A'/\alpha' \to B'/\beta' \Rightarrow^p \star]\!] \\
\overset{\beta}{=}\ & \lambda\kappa.\, \kappa\, (v^* : [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!] \Rightarrow^p \star) \\
\overset{\beta}{=}\ & \lambda\kappa.\, \kappa\, (v^* : [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!] \Rightarrow^p \star \to \star \Rightarrow \star) \\
\overset{\beta}{=}\ & \lambda\kappa.\, \kappa \\
& \quad ((\lambda x.\, v^*\, (x : \star \Rightarrow^{\bar{p}} [\![A']\!]) : ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!] \Rightarrow^p \star) : \star \to \star \Rightarrow \star).
\end{aligned}$$

On the other hand:

$$\begin{aligned}
& [\![v : A'/\alpha' \to B'/\beta' \Rightarrow^p \star/\star \to \star/\star \Rightarrow_p \star]\!] \\
=\ & \lambda\kappa.\, [\![v : A'/\alpha' \to B'/\beta' \Rightarrow^p \star/\star \to \star/\star]\!] \\
& \quad (\lambda x.\, \kappa\, (x : \star/\star \to \star/\star \Rightarrow_p \star)^*) \\
=\ & \lambda\kappa.\, [\![v : A'/\alpha' \to B'/\beta' \Rightarrow^p \star/\star \to \star/\star]\!] \\
& \quad (\lambda x.\, \kappa\, ((\lambda y.\, (x\ y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)) \\
=\ & \lambda\kappa.\, (\lambda\kappa'.\, \kappa'\, (v^* : [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!] \Rightarrow^p \star \to (\star \to \star) \to \star)) \\
& \quad (\lambda x.\, \kappa\, ((\lambda y.\, (x\ y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)) \\
\overset{\beta}{=}\ & \lambda\kappa.\, (\lambda\kappa'.\, \kappa'\, (\lambda z.\, v^*\, (z : \star \Rightarrow^{\bar{p}} [\![A']\!]) \\
& \qquad\qquad : ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!] \Rightarrow^p (\star \to \star) \to \star)) \\
& \quad (\lambda x.\, \kappa\, ((\lambda y.\, (x\ y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)) \\
\overset{\beta}{=}\ & \lambda\kappa.\, \kappa\, ((\lambda y.\, v^*\, (y : \star \Rightarrow^{\bar{p}} [\![A']\!]) \\
& \qquad : ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!] \Rightarrow^p (\star \to \star) \to \star \Rightarrow^p \star) \\
& \quad : \star \to \star \Rightarrow \star) \\
\overset{\beta\omega\xi\upsilon}{=}\ & \lambda\kappa.\, \kappa\, ((\lambda y.\, v^*\, (y : \star \Rightarrow^{\bar{p}} [\![A']\!]) \\
& \qquad : ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!] \Rightarrow^p \star) : \star \to \star \Rightarrow \star) \\
& \quad \text{(by Lemma A.3.8).}
\end{aligned}$$

Case (R_COLLAPSE): We are given $v : G \Rightarrow \star \Rightarrow^p A \longrightarrow v : G \Rightarrow^p A$ for some v, G, p and A such that $G \sim A$ and $A \neq \star$. We show that $[\![v : G \Rightarrow \star \Rightarrow^p A]\!] \overset{\beta\omega\xi\upsilon}{=} [\![v : G \Rightarrow^p A]\!]$ by case analysis on A.

Case $A = \iota$: Then, $G = \iota$ by Lemma A.3.2. Thus,

$$\begin{aligned}
[\![v : \iota \Rightarrow \star \Rightarrow^p \iota]\!] &= \lambda\kappa.\, [\![v : \iota \Rightarrow \star]\!]\, (\lambda x.\, \kappa\, (x : \star \Rightarrow^p \iota)) \\
&\overset{\beta}{=} \lambda\kappa.\, \kappa\, (v^* : \iota \Rightarrow \star \Rightarrow^p \iota) \\
&\overset{\beta}{=} \lambda\kappa.\, \kappa\, (v^* : \iota \Rightarrow^p \iota) \\
&\overset{\beta}{=} \lambda\kappa.\, [\![v]\!]\, (\lambda x.\, \kappa\, (x : \iota \Rightarrow^p \iota)) \\
&\overset{\beta}{=} [\![v : \iota \Rightarrow^p \iota]\!].
\end{aligned}$$

Case $A = A'/\alpha' \to B'/\beta'$: Then, $G = \star/\star \to \star/\star$ by Lemma A.3.3. Thus,

$$
\begin{array}{rl}
 & [\![v : \star/\star \to \star/\star \Rightarrow_q \star \Rightarrow^p A'/\alpha' \to B'/\beta']\!] \\
= & \lambda\kappa.\, [\![v : \star/\star \to \star/\star \Rightarrow_q \star]\!] \\
 & \quad (\lambda x.\, \kappa\, (x : \star \Rightarrow^p [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!])) \\
= & \lambda\kappa.\, (\lambda\kappa'.\, \kappa'\, ((\lambda y.\, (v^*\, y) : (\star \to \star) \to \star \Rightarrow^q \star) : \star \to \star \Rightarrow \star)) \\
 & \quad (\lambda x.\, \kappa\, (x : \star \Rightarrow^p [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!])) \\
\overset{\beta}{=} & \lambda\kappa.\, \kappa\, ((\lambda y.\, (v^*\, y) : (\star \to \star) \to \star \Rightarrow^q \star) \\
 & \qquad : \star \to \star \Rightarrow \star \Rightarrow^p [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \\
\overset{\beta}{=} & \lambda\kappa.\, \kappa\, ((\lambda y.\, (v^*\, y) : (\star \to \star) \to \star \Rightarrow^q \star) \\
 & \qquad : \star \to \star \Rightarrow^p [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \\
\overset{\beta}{=} & \lambda\kappa.\, \kappa\, (\lambda z.\, ((\lambda y.\, (v^*\, y) : (\star \to \star) \to \star \Rightarrow^q \star) \\
 & \qquad\quad (z : [\![A']\!] \Rightarrow^{\bar{p}} \star)) : \star \Rightarrow^p ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \\
\overset{\omega}{=} & \lambda\kappa.\, \kappa\, (\lambda z.\, (v^*\, (z : [\![A']\!] \Rightarrow^{\bar{p}} \star)) \\
 & \qquad : (\star \to \star) \to \star \Rightarrow^q \star \Rightarrow^p ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \\
\overset{\beta\omega v}{=} & \lambda\kappa.\, \kappa\, (\lambda z.\, (v^*\, (z : [\![A']\!] \Rightarrow^{\bar{p}} \star)) \\
 & \qquad : (\star \to \star) \to \star \Rightarrow^p ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \\
 & \quad \text{(by Lemma A.3.9)} \\
\overset{\beta}{=} & \lambda\kappa.\, \kappa\, (v^* : \star \to (\star \to \star) \to \star \Rightarrow^p [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \\
\overset{\beta}{=} & [\![v : \star/\star \to \star/\star \Rightarrow^p A'/\alpha' \to B'/\beta']\!]
\end{array}
$$

Case (R_CONFLICT): We are given $v : G \Rightarrow \star \Rightarrow^p A \longrightarrow \mathsf{blame}\, p$ for some v, G, p and A such that $G \not\sim A$. We show that $[\![v : G \Rightarrow \star \Rightarrow^p A]\!] \overset{\beta}{=} [\![\mathsf{blame}\, p]\!]$ by case analysis on A.

Case $A = \star$: Contradictory since $\star$ is compatible with any type.

Case $A = \iota$: Then, $G = \star/\star \to \star/\star$ from $G \not\sim A$. Thus,

$$
\begin{array}{rl}
 & [\![v : \star/\star \to \star/\star \Rightarrow_q \star \Rightarrow^p \iota]\!] \\
= & \lambda\kappa.\, [\![v : \star/\star \to \star/\star \Rightarrow_q \star]\!]\, (\lambda x.\, \kappa\, (x : \star \Rightarrow^p \iota)) \\
= & \lambda\kappa.\, (\lambda\kappa'.\, \kappa'\, ((\lambda y.\, (v^*\, y) : (\star \to \star) \to \star \Rightarrow^q \star) : \star \to \star \Rightarrow \star)) \\
 & \quad (\lambda x.\, \kappa\, (x : \star \Rightarrow^p \iota)) \\
\overset{\beta}{=} & \lambda\kappa.\, \kappa\, ((\lambda y.\, (v^*\, y) : (\star \to \star) \to \star \Rightarrow^q \star) : \star \to \star \Rightarrow \star \Rightarrow^p \iota) \\
\overset{\beta}{=} & \lambda\kappa.\, \kappa\, (\mathsf{blame}\, p) \\
\overset{\beta}{=} & [\![\mathsf{blame}\, p]\!].
\end{array}
$$

Case $A = A'/\alpha' \to B'/\beta'$: Then, $G = \iota$ for some ι from $G \not\sim A$. Thus,

$$
\begin{array}{rl}
 & [\![v : \iota \Rightarrow \star \Rightarrow^p A'/\alpha' \to B'/\beta']\!] \\
= & \lambda\kappa.\, [\![v : \iota \Rightarrow \star]\!]\, (\lambda x.\, \kappa\, (x : \star \Rightarrow^p [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!])) \\
= & \lambda\kappa.\, (\lambda\kappa'.\, \kappa'\, (v^* : \iota \Rightarrow \star)) \\
 & \quad (\lambda x.\, \kappa\, (x : \star \Rightarrow^p [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!])) \\
\overset{\beta}{=} & \lambda\kappa.\, \kappa\, (v^* : \iota \Rightarrow \star \Rightarrow^p [\![A']\!] \to ([\![B']\!] \to [\![\alpha']\!]) \to [\![\beta']\!]) \\
\overset{\beta}{=} & \lambda\kappa.\, \kappa\, (\mathsf{blame}\, p) \\
\overset{\beta}{=} & [\![\mathsf{blame}\, p]\!].
\end{array}
$$

Case (R_IsTrue): We are given $(v : G \Rightarrow \star)\,\mathsf{is}\,G \longrightarrow \mathsf{true}$ for some v and G. We show that $[\![(v : G \Rightarrow \star)\,\mathsf{is}\,G]\!] \stackrel{\beta}{=} [\![\mathsf{true}]\!]$ by case analysis on G.

Case $G = \iota$: Then,

$$\begin{array}{rcl}
[\![(v : \iota \Rightarrow \star)\,\mathsf{is}\,\iota]\!] & = & \lambda\kappa.\,[\![v : \iota \Rightarrow \star]\!]\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\,\iota)) \\
& = & \lambda\kappa.\,(\lambda\kappa'.\,\kappa'\,(v^* : \iota \Rightarrow \star))\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\,\iota)) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,((v^* : \iota \Rightarrow \star)\,\mathsf{is}\,\iota) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,\mathsf{true} \\
& = & [\![\mathsf{true}]\!].
\end{array}$$

Case $G = \star/\star \to \star/\star$: Then,

$$\begin{array}{rcl}
 & & [\![(v : \star/\star \to \star/\star \Rightarrow_p \star)\,\mathsf{is}\,\star/\star \to \star/\star]\!] \\
& = & \lambda\kappa.\,[\![v : \star/\star \to \star/\star \Rightarrow_p \star]\!]\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\,\star \to \star)) \\
& = & \lambda\kappa.\,(\lambda\kappa'.\,\kappa'\,((\lambda y.\,(v^*\,y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)) \\
& & \quad (\lambda x.\,\kappa\,(x\,\mathsf{is}\,\star \to \star)) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,(((\lambda y.\,(v^*\,y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)\,\mathsf{is}\,\star \to \star) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,\mathsf{true} \\
& = & [\![\mathsf{true}]\!].
\end{array}$$

Case (R_IsFalse): We are given $(v : H \Rightarrow \star)\,\mathsf{is}\,G \longrightarrow \mathsf{false}$ for some v, H and G such that $H \not\sim G$. We show that $[\![(v : H \Rightarrow \star)\,\mathsf{is}\,G]\!] \stackrel{\beta}{=} [\![\mathsf{false}]\!]$ by case analysis on G.

Case $G = \iota$: By case analysis on H.

Case $H = \iota'$ for some $\iota' \neq \iota$:

$$\begin{array}{rcl}
 & & [\![(v : \iota' \Rightarrow \star)\,\mathsf{is}\,\iota]\!] \\
& = & \lambda\kappa.\,[\![v : \iota' \Rightarrow \star]\!]\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\,\iota)) \\
& = & \lambda\kappa.\,(\lambda\kappa'.\,\kappa'\,(v^* : \iota' \Rightarrow \star))\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\,\iota)) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,((v^* : \iota' \Rightarrow \star)\,\mathsf{is}\,\iota) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,\mathsf{false} \\
& = & [\![\mathsf{false}]\!].
\end{array}$$

Case $H = \star/\star \to \star/\star$:

$$\begin{array}{rcl}
 & & [\![(v : \star/\star \to \star/\star \Rightarrow_p \star)\,\mathsf{is}\,\iota]\!] \\
& = & \lambda\kappa.\,[\![v : \star/\star \to \star/\star \Rightarrow_p \star]\!]\,(\lambda x.\,\kappa\,(x\,\mathsf{is}\,\iota)) \\
& = & \lambda\kappa.\,(\lambda\kappa'.\,\kappa'\,((\lambda y.\,(v^*\,y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)) \\
& & \quad (\lambda x.\,\kappa\,(x\,\mathsf{is}\,\iota)) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,(((\lambda y.\,(v^*\,y) : (\star \to \star) \to \star \Rightarrow^p \star) : \star \to \star \Rightarrow \star)\,\mathsf{is}\,\iota) \\
& \stackrel{\beta}{=} & \lambda\kappa.\,\kappa\,\mathsf{false} \\
& = & [\![\mathsf{false}]\!].
\end{array}$$

Case $G = \star/\star \to \star/\star$: Then, $H = \iota$ for some ι from $H \not\sim G$. Thus,

$$\begin{array}{ll} & [\![(v : \iota \Rightarrow \star) \,\mathsf{is}\, \star/\star \to \star/\star]\!] \\ = & \lambda\kappa.\, [\![v : \iota \Rightarrow \star]\!]\, (\lambda x.\, \kappa\, (x \,\mathsf{is}\star \to \star)) \\ = & \lambda\kappa.\, (\lambda\kappa'.\, \kappa'\, (v^* : \iota \Rightarrow \star))\, (\lambda x.\, \kappa\, (x \,\mathsf{is}\star \to \star)) \\ \overset{\beta}{=} & \lambda\kappa.\, \kappa\, ((v^* : \iota \Rightarrow \star) \,\mathsf{is}\star \to \star) \\ \overset{\beta}{=} & \lambda\kappa.\, \kappa\, \mathsf{false} \\ = & [\![\mathsf{false}]\!]. \end{array}$$

□

Theorem 4 (Preservation of Equality). *If $s \longmapsto t$, then $[\![s]\!] \approx [\![t]\!]$.*

Proof. By case analysis on the evaluation rule applied to s and t.

Case (E_STEP): We are given $E[s'] \longmapsto E[t']$ for some E, s' and t' such that $s' \longrightarrow t'$. By Lemmas A.3.5 and A.3.10,

$$[\![E[s']]\!] \overset{\beta}{=} [\![E]\!]\, [\![s']\!] \overset{\beta\eta\omega\xi\upsilon}{=} [\![E]\!]\, [\![t']\!] \overset{\beta}{=} [\![E[t']]\!].$$

Case (E_ABORT): We are given $E[\mathsf{blame}\, p] \longmapsto \mathsf{blame}\, p$ for some p. By Lemmas A.3.5 and A.3.7,

$$[\![E[\mathsf{blame}\, p]]\!] \overset{\beta}{=} [\![E]\!]\, [\![\mathsf{blame}\, p]\!] \overset{\beta\eta}{=} [\![\mathsf{blame}\, p]\!].$$

□

Appendix B

Proofs of Manifest Contracts with Parametric Polymorphism

This chapter gives the proofs of (syntactic) type soundness (Theorem 7) and parametricity (Theorem 8) of $\mathrm{F}^{\sigma}_{\mathrm{H}}$, without conjectures. We start with proving standard properties about free variables and substitution (Section B.1) because they are nonstandard and slightly tricky. Section B.2 shows cotermination, a key property to ensure that our type conversion relates only types equivalent "semantically" (in particular, Lemma 12 deals with the case for refinement types). Using cotermination, we show type soundness via progress (Theorem 5) and preservation (Theorem 6) in Section B.3. Finally, Section B.4 shows parametricity (Theorem 8), which also depends on cotermination.

B.1 Properties of substitution

Lemma B.1.1 (Free Term Variables After Substitution)**.** *Let σ be a substitution.*

1. *For any term e, $\mathrm{FV}(\sigma(e)) = (\mathrm{FV}(e) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(e)})$.*
2. *For any type T, $\mathrm{FV}(\sigma(T)) = (\mathrm{FV}(T) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(T)})$.*

Proof. By structural induction on e and T. We mention only the case of casts in the first case. We are given $e = \langle T_1 \Rightarrow T_2 \rangle^{l}_{\sigma_1}$. Let $\sigma_2 = \sigma(\sigma_1) \uplus \sigma|_{(\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1)}$. By definition, $\sigma(e) = \langle T_1 \Rightarrow T_2 \rangle^{l}_{\sigma_2}$ and

$$
\begin{aligned}
& \mathrm{FV}(\sigma(e)) \\
= \; & ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma_2)) \cup \mathrm{FV}(\sigma_2) \\
= \; & ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus (\mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma|_{(\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1)}))) \cup \mathrm{FV}(\sigma_2) \\
= \; & ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus (\mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma))) \cup \mathrm{FV}(\sigma_2).
\end{aligned}
$$

We have $\mathrm{FV}(e) = ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma_1)) \cup \mathrm{FV}(\sigma_1)$, and so

$$\mathrm{FV}(e) \setminus \mathrm{dom}(\sigma) = ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus (\mathrm{dom}(\sigma_1) \cup \mathrm{dom}(\sigma))) \cup (\mathrm{FV}(\sigma_1) \setminus \mathrm{dom}(\sigma)).$$

Thus, it suffices to show that

$$\mathrm{FV}(\sigma_2) = (\mathrm{FV}(\sigma_1) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(e)}).$$

Here, we have $\mathrm{FV}(\sigma_2) = \mathrm{FV}(\sigma(\sigma_1)) \cup \mathrm{FV}(\sigma|_{(\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2))\setminus\mathrm{dom}(\sigma_1)})$. By the IHs,

$$\begin{aligned}
\mathrm{FV}(\sigma(\sigma_1)) &= \bigcup_{x \in \mathrm{dom}(\sigma_1)} \mathrm{FV}(\sigma(\sigma_1(x))) \cup \bigcup_{\alpha \in \mathrm{dom}(\sigma_1)} \mathrm{FV}(\sigma(\sigma_1(\alpha))) \\
&= \bigcup_{x \in \mathrm{dom}(\sigma_1)} ((\mathrm{FV}(\sigma_1(x)) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1(x))})) \cup \\
&\quad \bigcup_{\alpha \in \mathrm{dom}(\sigma_1)} ((\mathrm{FV}(\sigma_1(\alpha)) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1(\alpha))})) \\
&= (\mathrm{FV}(\sigma_1) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1)}).
\end{aligned}$$

Thus,

$$\mathrm{FV}(\sigma_2) = (\mathrm{FV}(\sigma_1) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1)}) \cup \mathrm{FV}(\sigma|_{(\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2))\setminus\mathrm{dom}(\sigma_1)}),$$

and so it suffices to show that

$$\mathrm{FV}(\sigma|_{\mathrm{AFV}(e)}) = \mathrm{FV}(\sigma|_{\mathrm{AFV}(\sigma_1)}) \cup \mathrm{FV}(\sigma|_{(\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2))\setminus\mathrm{dom}(\sigma_1)}).$$

Since $\mathrm{AFV}(e) = ((\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma_1)) \cup \mathrm{AFV}(\sigma_1)$, we finish. □

Lemma B.1.2 (Free Type Variables After Substitution)**.** *Let σ be a substitution.*

1. *For any term e, $\mathrm{FTV}(\sigma(e)) = (\mathrm{FTV}(e) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FTV}(\sigma|_{\mathrm{AFV}(e)})$.*
2. *For any type T, $\mathrm{FTV}(\sigma(T)) = (\mathrm{FTV}(T) \setminus \mathrm{dom}(\sigma)) \cup \mathrm{FTV}(\sigma|_{\mathrm{AFV}(T)})$.*

Proof. Similarly to Lemma B.1.1; by structural induction on e and T. □

Lemma B.1.3. *Let σ be a substitution.*

1. *If $\mathrm{AFV}(e) \cap \mathrm{dom}(\sigma) = \emptyset$, then $\sigma(e) = e$.*
2. *If $\mathrm{AFV}(T) \cap \mathrm{dom}(\sigma) = \emptyset$, then $\sigma(T) = T$.*

Proof. By structural induction on e and T. We mention only the case of casts. We are given $e = \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma'}$. By definition:

$$\begin{aligned}
\mathrm{FV}(e) &= ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma')) \cup \mathrm{FV}(\sigma') \\
\mathrm{FTV}(e) &= ((\mathrm{FTV}(T_1) \cup \mathrm{FTV}(T_2)) \setminus \mathrm{dom}(\sigma')) \cup \mathrm{FTV}(\sigma')
\end{aligned}$$

Since $(\mathrm{FV}(e) \cup \mathrm{FTV}(e)) \cap \mathrm{dom}(\sigma) = \emptyset$, we have:

$$\begin{aligned}
\mathrm{dom}(\sigma) \cap ((\mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)) \setminus \mathrm{dom}(\sigma')) &= \emptyset \\
\mathrm{dom}(\sigma) \cap ((\mathrm{FTV}(T_1) \cup \mathrm{FTV}(T_2)) \setminus \mathrm{dom}(\sigma')) &= \emptyset
\end{aligned}$$

Thus, $\sigma(\langle T_1 \Rightarrow T_2 \rangle^l_{\sigma'}) = \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma(\sigma')}$. Since $(\mathrm{FV}(e) \cup \mathrm{FTV}(e)) \cap \mathrm{dom}(\sigma) = \emptyset$, we have $(\mathrm{FV}(\sigma') \cup \mathrm{FTV}(\sigma')) \cap \mathrm{dom}(\sigma) = \emptyset$, and thus $\sigma(\sigma') = \sigma'$ by the IHs. Thus, $\sigma(\langle T_1 \Rightarrow T_2 \rangle^l_{\sigma'}) = \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma'}$. □

Lemma B.1.4. *Let σ_1 and σ_2 be substitutions. Suppose that $\mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset$ and $\mathrm{AFV}(\sigma_2) \cap \mathrm{dom}(\sigma_1) = \emptyset$.*

1. *For any term e, $\sigma_2(\sigma_1(e)) = (\sigma_2(\sigma_1))(\sigma_2(e))$.*

2. *For any type* T, $\sigma_2(\sigma_1(T)) = (\sigma_2(\sigma_1))(\sigma_2(T))$.

Proof. By structural induction on e and T. We mention only the case of casts. We are given $e = \langle T_1 \Rightarrow T_2 \rangle^l_\sigma$. Let $S_1 = \mathrm{FV}(T_1) \cup \mathrm{FV}(T_2)$, $S_2 = \mathrm{FTV}(T_1) \cup \mathrm{FTV}(T_2)$, and $S = \mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)$. By definition, $\sigma_1(e) = \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma'_1}$ where $\sigma'_1 = \sigma_1(\sigma) \uplus \sigma_1|_{S \setminus \mathrm{dom}(\sigma)}$. Thus, $\sigma_2(\sigma_1(e)) = \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma''_1}$ where

$$\begin{aligned} \sigma''_1 &= \sigma_2(\sigma'_1) \uplus \sigma_2|_{S \setminus \mathrm{dom}(\sigma'_1)} \\ &= \sigma_2(\sigma_1(\sigma)) \uplus \sigma_2(\sigma_1)|_{S \setminus \mathrm{dom}(\sigma)} \uplus \sigma_2|_{S \setminus \mathrm{dom}(\sigma'_1)}. \end{aligned}$$

Also, we have $\sigma_2(e) = \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma'_2}$ where $\sigma'_2 = \sigma_2(\sigma) \uplus \sigma_2|_{S \setminus \mathrm{dom}(\sigma)}$, and $\sigma_2(\sigma_1)(\sigma_2(e)) = \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma''_2}$ where

$$\begin{aligned} \sigma''_2 &= (\sigma_2(\sigma_1))(\sigma'_2) \uplus \sigma_2(\sigma_1)|_{S \setminus \mathrm{dom}(\sigma'_2)} \\ &= (\sigma_2(\sigma_1))(\sigma_2(\sigma)) \uplus (\sigma_2(\sigma_1))(\sigma_2)|_{S \setminus \mathrm{dom}(\sigma)} \uplus \sigma_2(\sigma_1)|_{S \setminus \mathrm{dom}(\sigma'_2)}. \end{aligned}$$

We show that $\sigma''_1 = \sigma''_2$ as follows.

1. We have $\sigma_2(\sigma_1(\sigma)) = (\sigma_2(\sigma_1))(\sigma_2(\sigma))$ because, for any $x \in \mathsf{dom}(\sigma)$,

$$\begin{aligned} \sigma_2(\sigma_1(\sigma))(x) &= \sigma_2(\sigma_1(\sigma(x))) \\ &= (\sigma_2(\sigma_1))(\sigma_2(\sigma(x))) \quad \text{(by the IH)} \\ &= (\sigma_2(\sigma_1))(\sigma_2(\sigma))(x), \end{aligned}$$

   and for any $\alpha \in \mathrm{dom}(\sigma)$, $\sigma_2(\sigma_1(\sigma))(\alpha) = (\sigma_2(\sigma_1))(\sigma_2(\sigma))(\alpha)$, which can be proven similarly to term variables by the IH.

2. We show that $\sigma_2(\sigma_1)|_{S \setminus \mathrm{dom}(\sigma)} = \sigma_2(\sigma_1)|_{S \setminus \mathrm{dom}(\sigma'_2)}$, that is, we show that

$$\mathrm{dom}(\sigma_1) \cap (S \setminus \mathrm{dom}(\sigma)) = \mathrm{dom}(\sigma_1) \cap (S \setminus \mathrm{dom}(\sigma'_2)).$$

   Here, we have

$$\begin{aligned} \mathsf{dom}(\sigma'_2) &= \mathrm{dom}(\sigma) \cup (\mathrm{dom}(\sigma_2) \cap (S \setminus \mathrm{dom}(\sigma))) \\ &= (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma_2)) \cap (\mathrm{dom}(\sigma) \cup (S \setminus \mathrm{dom}(\sigma))) \\ &= (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma_2)) \cap (\mathrm{dom}(\sigma) \cup S) \\ &= \mathrm{dom}(\sigma) \cup (\mathrm{dom}(\sigma_2) \cap S). \end{aligned}$$

   Thus,

$$\begin{aligned} \mathrm{dom}(\sigma_1) \cap (S \setminus \mathrm{dom}(\sigma'_2)) &= \mathrm{dom}(\sigma_1) \cap (S \setminus (\mathrm{dom}(\sigma) \cup (\mathrm{dom}(\sigma_2) \cap S))) \\ &= \mathrm{dom}(\sigma_1) \cap (S \setminus (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma_2))) \\ &= (\mathrm{dom}(\sigma_1) \cap S) \setminus (\mathrm{dom}(\sigma) \cup \mathrm{dom}(\sigma_2)) \\ &= (\mathrm{dom}(\sigma_1) \cap S) \setminus \mathrm{dom}(\sigma) \\ &\qquad \text{(since } \mathrm{dom}(\sigma_1) \cap \mathrm{dom}(\sigma_2) = \emptyset\text{)} \\ &= \mathrm{dom}(\sigma_1) \cap (S \setminus \mathrm{dom}(\sigma)). \end{aligned}$$

3. We show that $\sigma_2|_{S\setminus\mathrm{dom}(\sigma_1')} = (\sigma_2(\sigma_1))(\sigma_2)|_{S\setminus\mathrm{dom}(\sigma)}$. Since $\mathrm{AFV}(\sigma_2) \cap \mathrm{dom}(\sigma_1) = \emptyset$, we have $(\sigma_2(\sigma_1))(\sigma_2) = \sigma_2$ by Lemma B.1.3. Thus, it suffices to show that

$$\mathrm{dom}(\sigma_2) \cap (S \setminus \mathrm{dom}(\sigma_1')) = \mathrm{dom}(\sigma_2) \cap (S \setminus \mathrm{dom}(\sigma)),$$

which can be shown similarly to the above.

□

B.2 Cotermination

The key observation in proving cotermination is that the relation $\{([e_1/x]e, [e_2/x]e) \mid e_1 \longrightarrow e_2\}$ is weak bisimulation. Lemmas 8 and 9 show it for cases that left- and right-hand terms first evaluate, respectively; the cases of term and type applications (without reducible subterms) are shown in Lemmas B.2.3 and B.2.5, respectively. We show cotermination in the case that substitutions map only one term variable (Lemma 10), and then show general cases (Lemma 11).

Throughout the proof, we implicitly make use of the determinism of the semantics.

Lemma B.2.1 (Determinism). *If $e \longrightarrow e_1$ and $e \longrightarrow e_2$ then $e_1 = e_2$.*

Proof. By case analysis for $\rightsquigarrow$ and induction on $e \longrightarrow e_1$.

□

Lemma B.2.2. *Suppose that e_1 and e_2 are closed terms and that e_1', $[e_1/x]e_2'$ and $[e_2/x]e_2'$ are values. If $[e_1/x](e_1'\ e_2') \longrightarrow e$, then $[e_2/x](e_1'\ e_2') \longrightarrow [e_2/x]e'$ for some e' such that $e = [e_1/x]e'$.*

Proof. By case analysis on e_1'. Here e_1' takes the form of either lambda abstraction or cast since the application term $[e_1/x](e_1'\ e_2')$ takes a step.

Case $e_1' = \lambda y{:}T.\ e'$: Without loss of generality, we can suppose that y is fresh. By (E_REDUCE)/(E_BETA), $[e_1/x](e_1'\ e_2') = [e_1/x]((\lambda y{:}T.\ e')\ e_2') \longrightarrow [[e_1/x]e_2'/y][e_1/x]e'$ and $[e_2/x](e_1'\ e_2') = [e_2/x]((\lambda y{:}T.\ e')\ e_2') \longrightarrow [[e_2/x]e_2'/y][e_2/x]e'$. Since, for $i \in \{1,2\}$, $[[e_i/x]e_2'/y][e_i/x]e' = [e_i/x][e_2'/y]e'$ by Lemma B.1.4, we finish.

Case $e_1' = \langle T \Rightarrow T\rangle_\sigma^l$: By (E_REDUCE)/(E_REFL), $[e_1/x](e_1'\ e_2') = [e_1/x](\langle T \Rightarrow T\rangle_\sigma^l\ e_2') \longrightarrow [e_1/x]e_2'$ and $[e_2/x](e_1'\ e_2') = [e_2/x](\langle T \Rightarrow T\rangle_\sigma^l\ e_2') \longrightarrow [e_2/x]e_2'$, and thus we finish.

Case $e_1' = \langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22}\rangle_\sigma^l$ where $y{:}T_{11} \to T_{12} \neq y{:}T_{21} \to T_{22}$: Without loss of generality, we can suppose that y and variables of $\mathrm{dom}(\sigma)$ are fresh. Let z be a fresh variable and $i, j \in \{1,2\}$. Moreover, let σ_i be

$$[e_i/x]\sigma \uplus ([e_i/x]|_{(\mathrm{AFV}(y:T_{11}\to T_{12})\cup\mathrm{AFV}(y:T_{21}\to T_{22}))\setminus\mathrm{dom}(\sigma)})$$

and σ_{ij} be $\sigma_i|_{\mathrm{AFV}(T_{1j})\cup\mathrm{AFV}(T_{2j})}$. Then, $[e_i/x]e_1' = \langle y{:}T_{11} \to T_{12} \Rightarrow y{:}T_{21} \to T_{22}\rangle_{\sigma_i}^l$ and, by (E_REDUCE)/(E_FUN), $[e_i/x](e_1'\ e_2') \longrightarrow e_i''$ where e_i'' takes the form:

$$\lambda y{:}\sigma_i(T_{21}).\ \mathsf{let}\ z : \sigma_i(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle_{\sigma_{i1}}^l\ y\ \mathsf{in}\ \langle [z/y]T_{12} \Rightarrow T_{22}\rangle_{\sigma_{i2}}^l\ ([e_i/x]e_2'\ z).$$

Here, let $\sigma'_j = \sigma|_{\mathrm{AFV}(T_{1j})\cup\mathrm{AFV}(T_{2j})}$ and e' be

$$\lambda y{:}\sigma(T_{21}).\ \mathsf{let}\ z : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma'_1}\ y\ \mathsf{in}\ \langle [z/y]T_{12} \Rightarrow T_{22}\rangle^l_{\sigma'_2}\ (e'_2\ z)$$

for some fresh variable z.

We show $[e_i/x]e' = e''_i$. By Lemma B.1.4, $[e_i/x]\sigma(T_{21}) = ([e_i/x]\sigma)([e_i/x]T_{21}) = \sigma_i(T_{21})$ and, similarly, $[e_i/x]\sigma(T_{11}) = \sigma_i(T_{11})$. Also, letting $S_j = \mathrm{AFV}(T_{1j}) \cup \mathrm{AFV}(T_{2j})$,

$$\begin{array}{lll} & [e_i/x]\sigma'_j \uplus ([e_i/x]|_{S_j\setminus\mathrm{dom}(\sigma'_j)}) & \\ = & [e_i/x]\sigma'_j \uplus ([e_i/x]|_{S_j\setminus\mathrm{dom}(\sigma)}) & (\text{because } S_j \setminus \mathrm{dom}(\sigma'_j) = S_j \setminus \mathrm{dom}(\sigma)) \\ = & ([e_i/x]\sigma|_{S_j}) \uplus ([e_i/x]|_{S_j\setminus\mathrm{dom}(\sigma)}) & \\ = & (\sigma_i|_{\mathrm{dom}(\sigma)\cap S_j}) \uplus ([e_i/x]|_{S_j\setminus\mathrm{dom}(\sigma)}) & \\ = & (\sigma_{ij}|_{\mathrm{dom}(\sigma)}) \uplus ([e_i/x]|_{S_j\setminus\mathrm{dom}(\sigma)}) & \\ = & \sigma_{ij}. & \end{array}$$

The last equation is derived from the fact that

$$\begin{array}{ll} & x \in \mathsf{dom}(\sigma_{ij}) \\ \iff & x \in S_j \cap \mathrm{dom}(\sigma_i) \\ \iff & x \in S_j \cap ((\mathrm{AFV}(y{:}T_{11} \to T_{12}) \cup \mathrm{AFV}(y{:}T_{21} \to T_{22})) \setminus \mathrm{dom}(\sigma)) \\ \iff & x \in (S_j \cap (\mathrm{AFV}(y{:}T_{11} \to T_{12}) \cup \mathrm{AFV}(y{:}T_{21} \to T_{22}))) \setminus \mathrm{dom}(\sigma) \\ \iff & x \in S_j \setminus \mathrm{dom}(\sigma). \end{array}$$

Case $e'_1 = \langle \forall\alpha.T_1 \Rightarrow \forall\alpha.T_2\rangle^l_\sigma$ where $\forall\alpha.T_1 \neq \forall\alpha.T_2$: Without loss of generality, we can suppose that α is fresh. Let $i \in \{1,2\}$ and $\sigma_i = [e_i/x]\sigma \uplus ([e_i/x]|_{(\mathrm{AFV}(\forall\alpha.T_1)\cup\mathrm{AFV}(\forall\alpha.T_2))\setminus\mathrm{dom}(\sigma)})$. Then, by (E_REDUCE)/(E_FORALL), $[e_i/x](e'_1\ e'_2) \longrightarrow \Lambda\alpha.\ \langle [\alpha/\alpha]T_1 \Rightarrow T_2\rangle^l_{\sigma_i}\ ([e_i/x]e'_2\ \alpha)$. Because

$$\begin{array}{l} x \in (\mathrm{AFV}(\forall\alpha.T_1) \cup \mathrm{AFV}(\forall\alpha.T_2)) \setminus \mathrm{dom}(\sigma) \\ \quad \text{if and only if } x \in (\mathrm{AFV}(T_1) \cup \mathrm{AFV}(T_2)) \setminus \mathrm{dom}(\sigma), \end{array}$$

we have $\langle [\alpha/\alpha]T_1 \Rightarrow T_2\rangle^l_{\sigma_i} = [e_i/x](\langle [\alpha/\alpha]T_1 \Rightarrow T_2\rangle^l_\sigma)$. Thus, we finish.

Case $e'_1 = \langle T \Rightarrow \{y{:}T \mid e\}\rangle^l_\sigma$: Without loss of generality, we can suppose that y and variables of $\mathsf{dom}(\sigma)$ are fresh. Let $i \in \{1,2\}$ and $\sigma_i = [e_i/x]\sigma \uplus ([e_i/x]|_{\mathrm{AFV}(\{y:T|e\})\setminus\mathrm{dom}(\sigma)})$. By (E_REDUCE)/(E_CHECK), $[e_i/x](e'_1\ e'_2) \longrightarrow e''_i$ where $e''_i = \langle \sigma_i(\{y{:}T \mid e\}), \sigma_i([[e_i/x]e'_2/y]e), [e_i/x]e'_2\rangle^l$. Since e_i and $[e_i/x]e'_2$ are closed (recall evaluation relation is defined over closed terms), we have

$$\begin{array}{lll} \sigma_i([[e_i/x]e'_2/y]e) & = & ([e_i/x]\sigma)([e_i/x][[e_i/x]e'_2/y]e) \\ & = & ([e_i/x]\sigma)([[e_i/x]e'_2/y][e_i/x]e) \\ & = & ([e_i/x]\sigma)([e_i/x][e'_2/y]e) \\ & = & [e_i/x](\sigma([e'_2/y]e)) \end{array}$$

by Lemmas B.1.3 and B.1.4. Because $\sigma_i(\{y{:}T \mid e\}) = [e_i/x]\sigma(\{y{:}T \mid e\})$ similarly, we finish.

Case $e_1' = \langle \{y{:}T_1 \mid e\} \Rightarrow T_2 \rangle^l_\sigma$ where $T_2 \neq \{y{:}T_1 \mid e_1\}$ and $T_2 \neq \{z{:}\{y{:}T_1 \mid e\} \mid e'\}$ for any z and e': Let $i \in \{1,2\}$ and

$$\begin{array}{rcl} \sigma_i & = & [e_i/x]\sigma \uplus ([e_i/x]|_{(\mathrm{AFV}(\{y:T_1|e_1\})\cup\mathrm{AFV}(T_2))\setminus\mathrm{dom}(\sigma)}) \\ \sigma_i' & = & \sigma_i|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)}. \end{array}$$

Then, by (E_REDUCE)/(E_FORGET), $[e_i/x](e_1'\ e_2') \longrightarrow \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma_i'}\ [e_i/x]e_2'$.

Letting $\sigma' = \sigma|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)}$, it suffices to show that $[e_i/x](\langle T_1 \Rightarrow T_2 \rangle^l_{\sigma'}) = \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma_i'}$. We can show that $[e_i/x]\sigma' \uplus ([e_i/x]|_{(\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2))\setminus\mathrm{dom}(\sigma')}) = \sigma_i'$ similarly to the case of casts between function types, and so we finish.

Case $e_1' = \langle T_1 \Rightarrow \{y{:}T_2 \mid e\} \rangle^l_\sigma$ where $T_1 \neq T_2$ and $T_1 \neq \{z{:}T' \mid e'\}$ for any z, T' and e': Let $i \in \{1,2\}$ and

$$\begin{array}{rcl} \sigma_i & = & [e_i/x]\sigma \uplus ([e_i/x]|_{(\mathrm{AFV}(T_1)\cup\mathrm{AFV}(\{y:T_2|e\}))\setminus\mathrm{dom}(\sigma)}) \\ \sigma_{i1} & = & \sigma_i|_{\mathrm{AFV}(\{y:T_2|e\})} \\ \sigma_{i2} & = & \sigma_i|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)}. \end{array}$$

Then, by (E_REDUCE)/(E_PRECHECK), $[e_i/x](e_1'\ e_2') \longrightarrow e_i''$ where

$$e_i'' = \langle T_2 \Rightarrow \{y{:}T_2 \mid e\} \rangle^l_{\sigma_{i1}}\ (\langle T_1 \Rightarrow T_2 \rangle^l_{\sigma_{i2}}\ [e_i/x]e_2').$$

Let

$$\begin{array}{rcl} \sigma_1' & = & \sigma|_{\mathrm{AFV}(\{y:T_2|e\})} \\ \sigma_2' & = & \sigma|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)} \\ e' & = & \langle T_2 \Rightarrow \{y{:}T_2 \mid e\} \rangle^l_{\sigma_1'}\ (\langle T_1 \Rightarrow T_2 \rangle^l_{\sigma_2'}\ e_2'). \end{array}$$

Then, it suffices to show that $[e_i/x]e' = e_i''$. We can show that $[e_i/x]\sigma_1' \uplus ([e_i/x]|_{\mathrm{AFV}(\{y:T_2|e\})\setminus\mathrm{dom}(\sigma_1')}) = \sigma_{i1}$ and $[e_i/x]\sigma_2' \uplus ([e_i/x]|_{(\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2))\setminus\mathrm{dom}(\sigma_2')}) = \sigma_{i2}$ similarly to the above, and so we finish.

□

Lemma B.2.3. *Suppose that $e_1 \longrightarrow e_2$ and that $[e_1/x]e_1'$, $[e_1/x]e_2'$ and $[e_2/x]e_2'$ are values.*

1. *If $[e_1/x](e_1'\ e_2') \longrightarrow e$, then $[e_2/x](e_1'\ e_2') \longrightarrow [e_2/x]e'$ for some e' such that $e = [e_1/x]e'$.*
2. *If $[e_2/x](e_1'\ e_2') \longrightarrow e$, then $[e_1/x](e_1'\ e_2') \longrightarrow [e_1/x]e'$ for some e' such that $e = [e_2/x]e'$.*

Proof. Since $[e_1/x]e_1'$ is a value, and e_1 is not a value from $e_1 \longrightarrow e_2$, we have e_1' is not a variable, and thus e_1' is a value from the assumption that so is $[e_1/x]e_1'$. Since evaluation relation is defined over closed terms, we finish by Lemma B.2.2. □

Lemma B.2.4. *Suppose that e_1 and e_2 are closed terms and that e is a value. If $[e_1/x](e\ T) \longrightarrow e'$, then $[e_2/x](e\ T) \longrightarrow [e_2/x]e''$ for some e'' such that $e' = [e_1/x]e''$.*

Proof. Since the type application term $[e_1/x](e\ T)$ takes a step, e takes the form of type abstraction. Let $e = \Lambda\alpha.\ e'$. Without loss of generality, we can suppose that α is fresh.

Let $i \in \{1, 2\}$. By (E_REDUCE)/(E_TBETA), $[e_i/x](e\ T) \longrightarrow [[e_i/x]T/\alpha][e_i/x]e'$. Since e_i is closed, we have $[[e_i/x]T/\alpha][e_i/x]e' = [e_i/x][T/\alpha]e'$ by Lemma B.1.4, and thus we finish. □

Lemma B.2.5. *Suppose that $e_1 \longrightarrow e_2$ and that $[e_1/x]e$ is a value.*

1. *If $[e_1/x](e\ T) \longrightarrow e'$, then $[e_2/x](e\ T) \longrightarrow [e_2/x]e''$ for some e'' such that $e' = [e_1/x]e''$.*
2. *If $[e_2/x](e\ T) \longrightarrow e'$, then $[e_1/x](e\ T) \longrightarrow [e_1/x]e''$ for some e'' such that $e' = [e_2/x]e''$.*

Proof. By Lemma B.2.4 because it is found that e is a value and that e_1 and e_2 are closed terms (evaluation relation is defined over closed terms). □

Lemma B.2.6. *If $e_1 \longrightarrow^* e_2$, then $E\,[e_1] \longrightarrow^* E\,[e_2]$.*

Proof. By induction on the number of evaluation steps of e_1 with (E_COMPAT). □

Lemma 8 (Weak bisimulation, left side)**.** *Suppose that $e_1 \longrightarrow e_2$. If $[e_1/x]e \longrightarrow e'$, then $[e_2/x]e \longrightarrow^* [e_2/x]e''$ for some e'' such that $e' = [e_1/x]e''$.*

Proof. By structural induction on e. Here e_1 is not a value, since $e_1 \longrightarrow e_2$.

Case $e = x$: Since $[e_1/x]e = e_1$ and $[e_2/x]e = e_2$, we finish by Lemma B.1.3 when letting $e'' = e_2$ because e_2 is closed (recall that the evaluation relation is a relation over closed terms).

Case $e = v$, y where $x \neq y$ or $\Uparrow l$: Contradiction from $[e_1/x]e \longrightarrow e'$.

Case $e = op\,(e'_1, .., e'_n)$: If all terms $[e_1/x]e'_i$ are values, then they are constants since $[e_1/x]op\,(e'_1, \ldots, e'_n)$ takes a step. Since e_1 is not a value, $e'_i = k_i$ for some k_i. Thus, $[e_1/x]e = [e_2/x]e = op\,(k_1, \ldots, k_n)$ and so we finish.

Otherwise, we suppose that some $[e_1/x]e'_i$ is not a value and all terms to the left of $[e_1/x]e'_i$ are values. From that, we can show that all terms to the left of $[e_2/x]e'_i$ are values since e_1 is not a value. If $[e_1/x]e'_i$ gets stuck, then contradiction because $[e_1/x]e$ takes a step. If $[e_1/x]e'_i \longrightarrow e''$, then, by the IH, $[e_2/x]e'_i \longrightarrow^* [e_2/x]e''_i$ for some e''_i such that $e'' = [e_1/x]e''_i$. Thus, we finish by Lemma B.2.6. Otherwise, if $[e_1/x]e'_i = \Uparrow l$, then $[e_2/x]e'_i = \Uparrow l$ because $e'_i = \Uparrow l$ by $e_1 \neq \Uparrow l$, which follows from $e_1 \longrightarrow e_2$. Thus, we finish by (E_BLAME).

Case $e = e'_1\ e'_2$: We can show the case where either $[e_1/x]e'_1$ or $[e_1/x]e'_2$ is not a value similarly to the above. Otherwise, if they are values, we can find that so are $[e_2/x]e'_1$ and $[e_2/x]e'_2$, and thus we finish by Lemma B.2.3 (1).

Case $e = e'_1\ T_2$: Similarly to the case of function application, with Lemma B.2.5 (1).

Case $e = \langle \{y{:}T \mid e'_1\}, e'_2, v\rangle^l$: Similarly to the above.

□

Lemma B.2.7. *If $e_1 \longrightarrow e_2$, and $[e_2/x]e$ is a value, then there exists some e' such that*

- *$[e_1/x]e \longrightarrow^* [e_1/x]e'$,*
- *$[e_1/x]e'$ is a value, and*

- $[e_2/x]e = [e_2/x]e'$.

Proof. By case analysis on e. □

Lemma B.2.8. *If* $e_1 \longrightarrow e_2$, *and* $[e_2/x]e = \Uparrow l$, *then* $[e_1/x]e \longrightarrow^* \Uparrow l$.

Proof. By case analysis on e. □

Lemma 9 (Weak bisimulation, right side)**.** *Suppose that* $e_1 \longrightarrow e_2$. *If* $[e_2/x]e \longrightarrow e'$, *then* $[e_1/x]e \longrightarrow^* [e_1/x]e''$ *for some* e'' *such that* $e' = [e_2/x]e''$.

Proof. By structural induction on e.

Case $e = x$: Since $[e_1/x]e = e_1$ and $[e_2/x]e = e_2$, we finish by Lemma B.1.3 when letting $e'' = e'$.

Case $e = v$, y where $x \neq y$ or $\Uparrow l$: Contradiction from $[e_2/x]e \longrightarrow e'$.

Case $e = op\,(e'_1, .., e'_n)$: If all terms $[e_2/x]e'_i$ are values, then they are constants since $[e_2/x]op\,(e'_1, \ldots, e'_n)$ takes a step. By Lemma B.2.7, $[e_1/x]op\,(e'_1, \ldots, e'_n) \longrightarrow^* [e_1/x]op\,(e''_1, \ldots, e''_2)$ for some $e''_1, ..., e''_n$ such that $[e_2/x]op\,(e'_1, \ldots, e'_n) = [e_2/x]op\,(e''_1, \ldots, e''_n)$. Since e_1 is not a value from $e_1 \longrightarrow e_2$, $e''_i = k_i$ for some k_i. Thus, we finish.

Otherwise, we suppose that some $[e_2/x]e'_i$ is not a value and all terms to the left of $[e_2/x]e'_i$ are values. By Lemma B.2.7, each term $[e_1/x]e'_j$ to the left of $[e_1/x]e'_i$ evaluates to a value $[e_1/x]e''_j$ for some e''_j such that $[e_2/x]e'_j = [e_2/x]e''_j$. If $[e_2/x]e'_i$ gets stuck, then contradiction because $[e_2/x]e$ takes a step. If $[e_2/x]e'_i = \Uparrow l$, then $[e_1/x]e'_i \longrightarrow^* \Uparrow l$ by Lemma B.2.8. Thus, we finish by (E_BLAME). Otherwise, if $[e_2/x]e'_i \longrightarrow e''$, then we finish by the IH and (E_COMPAT).

Case $e = e'_1\ e'_2$: We can show the case where either $[e_2/x]e'_1$ or $[e_2/x]e'_2$ is not a value similarly to the above. Otherwise, if they are values, we can find, by Lemma B.2.7, that $[e_1/x]e'_1$ and $[e_1/x]e'_2$ evaluates to values $[e_1/x]e''_1$ and $[e_1/x]e''_2$ for some e''_1 and e''_2 such that $[e_2/x]e'_1 = [e_2/x]e''_1$ and $[e_2/x]e'_2 = [e_2/x]e''_2$, respectively. Then, we finish by Lemma B.2.3 (2).

Case $e = e'_1\ T_2$: Similarly to the case of function application, with Lemma B.2.5 (2).

Case $e = \langle \{y{:}T \mid e'_1\}, e'_2, v \rangle^l$: Similarly to the above.

□

Lemma 10 (Cotermination, one variable)**.**

1. *Suppose that* $e_1 \longrightarrow e_2$.
   (a) *If* $[e_1/x]e \longrightarrow^* \mathsf{true}$, *then* $[e_2/x]e \longrightarrow^* \mathsf{true}$.
   (b) *If* $[e_2/x]e \longrightarrow^* \mathsf{true}$, *then* $[e_1/x]e \longrightarrow^* \mathsf{true}$.
2. *Suppose that* $e_1 \longrightarrow^* e_2$.
   (a) *If* $[e_1/x]e \longrightarrow^* \mathsf{true}$, *then* $[e_2/x]e \longrightarrow^* \mathsf{true}$.
   (b) *If* $[e_2/x]e \longrightarrow^* \mathsf{true}$, *then* $[e_1/x]e \longrightarrow^* \mathsf{true}$.

Proof.

1. By induction on the number of evaluation steps of $[e_1/x]e$ and $[e_2/x]e$ with Lemma 8 and Lemmas B.2.7 and 9, respectively.

2. By induction on the number of evaluation steps of e_1 with the first case.

□

Lemma 11 (Cotermination). *Suppose that* $\sigma_1 \longrightarrow^* \sigma_2$.

1. *If* $\sigma_1(e) \longrightarrow^* \mathsf{true}$, *then* $\sigma_2(e) \longrightarrow^* \mathsf{true}$.

2. *If* $\sigma_2(e) \longrightarrow^* \mathsf{true}$, *then* $\sigma_1(e) \longrightarrow^* \mathsf{true}$.

Proof. By induction on the size of $\mathsf{dom}(\sigma_1)$ with Lemma 10. □

B.3 Type soundness

We show type soundness in a syntactic manner: progress (Theorem 5) and preservation (Theorem 6). Cotermination is used to show value inversion (Lemma 13), which implies consistency of the contract system of $\mathrm{F}^{\sigma}_{\mathrm{H}}$ and is used to show progress in the case for (T_OP). After proving properties of convertibility (Lemmas B.3.1–B.3.6) and compatibility (Lemmas B.3.7–B.3.10), we show progress and preservation, using standard lemmas: weakening lemmas (Lemmas 16 and 17), substitution lemmas (Lemmas 18 and 19), inversion lemmas (Lemmas 20, B.3.11, and B.3.12), and canonical forms lemma (Lemma 21).

Lemma 12 (Cotermination of refinement types). *If* $\{x{:}T_1 \mid e_1\} \equiv \{x{:}T_2 \mid e_2\}$ *then* $T_1 \equiv T_2$ *and* $[v/x]e_1 \longrightarrow^* \mathsf{true}$ *iff* $[v/x]e_2 \longrightarrow^* \mathsf{true}$, *for any closed value* v.

Proof. By induction on the equivalence. There are three cases.

Case (C_REFINE): We have $T_1 \equiv T_2$ by assumption. We know that $e_1 = \sigma_1(e)$ and $e_2 = \sigma_2(e)$ for $\sigma_1 \longrightarrow^* \sigma_2$. It is trivially true that $v \longrightarrow^* v$, so $[v/x]\sigma_1 \longrightarrow^* [v/x]\sigma_2$. By cotermination (Lemma 11), we know that $[v/x]\sigma_1(e) \longrightarrow^* \mathsf{true}$ iff $[v/x]\sigma_2(e) \longrightarrow^* \mathsf{true}$.

Case (C_SYM): By the IH.

Case (C_TRANS): By the IHs and transitivity of $\equiv$ and cotermination. □

Lemma 13 (Value inversion). *If* $\emptyset \vdash v : T$ *and* $\mathrm{unref}_n(T) = \{x{:}T_n \mid e_n\}$ *then* $[v/x]e_n \longrightarrow^* \mathsf{true}$.

Proof. By induction on the height of the typing derivation; we list all the cases that could type values.

Case (T_CONST): By assumption of valid typing of constants.

Case (T_ABS): Contradictory—the type is wrong.

Case (T_TABS): Contradictory—the type is wrong.

Case (T_CAST): Contradictory—the type is wrong.

Case (T_CONV): By applying Lemma 12 on the stack of refinements on T.

Case (T_FORGET): By the IH on $\emptyset \vdash v : \{x{:}T \mid e\}$, adjusting each of the n down by one to cover the stack of refinements on T.

Case (T_EXACT): By assumption for the outermost refinement; by the IH on $\emptyset \vdash v : T$ for the rest. □

Lemma B.3.1 (Reflexivity of conversion).
$T \equiv T$ for all T.

Proof. By induction on T. □

Lemma B.3.2 (Like-type arrow conversion). *If $x{:}T_{11} \to T_{12} \equiv T$ then $T = x{:}T_{21} \to T_{22}$.*

Proof. By induction on the conversion relation. Only (C_FUN) applies, and (C_SYM) and (C_TRANS) are resolved by the IH. □

Lemma B.3.3 (Conversion arrow inversion). *If $x{:}T_{11} \to T_{12} \equiv x{:}T_{21} \to T_{22}$ then $T_{11} \equiv T_{21}$ and $T_{12} \equiv T_{22}$.*

Proof. By induction on the conversion derivation, using Lemma B.3.2. □

Lemma B.3.4 (Like-type forall conversion). *If $\forall\alpha.\, T_1 \equiv T$ then $T = \forall\alpha.\, T_2$.*

Proof. By induction on the conversion relation. Only (C_FORALL) applies, and (C_SYM) and (C_TRANS) are resolved by the IH. □

Lemma B.3.5 (Conversion forall inversion). *If $\forall\alpha.\, T_1 \equiv \forall\alpha.\, T_2$ then $T_1 \equiv T_2$.*

Proof. By induction on the conversion derivation, using Lemma B.3.4. □

Lemma 14 (Term substitutivity of conversion).
If $T_1 \equiv T_2$ and $e_1 \longrightarrow^* e_2$ then $[e_1/x]\,T_1 \equiv [e_2/x]\,T_2$.

Proof. By induction on $T_1 \equiv T_2$.

Case (C_VAR): By (C_VAR).

Case (C_BASE): By (C_BASE).

Case (C_REFINE): $T_1 = \{y{:}T_1' \mid \sigma_1(e)\}$ and $T_2 = \{y{:}T_2' \mid \sigma_2(e)\}$ such that $T_1' \equiv T_2'$ and $\sigma_1 \longrightarrow^* \sigma_2$. By the IH on $T_1' \equiv T_2'$, we know that $[e_1/x]\,T_1' \equiv [e_2/x]\,T_2'$. Since $e_1 \longrightarrow^* e_2$, we know that $\sigma_1 \uplus [e_1/x] \longrightarrow^* \sigma_2 \uplus [e_2/x]$, and we are done by (C_REFINE).

Case (C_FUN): By the IHs and (C_FUN).

Case (C_FORALL): By the IH and (C_FORALL).

Case (C_TRANS): By the IHs and (C_TRANS).

Case (C_SYM): By the IHs and (C_SYM). □

Lemma 15 (Type substitutivity of conversion).
If $T_1 \equiv T_2$ then $[T/\alpha]\,T_1 \equiv [T/\alpha]\,T_2$.

Proof. By induction on $T_1 \equiv T_2$.

Case (C_VAR): If $T_1 = T_2 = \alpha$, then by reflexivity (Lemma B.3.1). Otherwise, by (C_VAR).

Case (C_BASE): By (C_BASE).

Case (C_REFINE): $T_1 = \{y{:}T_1' \mid \sigma_1(e)\}$ and $T_2 = \{y{:}T_2' \mid \sigma_2(e)\}$ such that $T_1' \equiv T_2'$ and $\sigma_1 \longrightarrow^* \sigma_2$. By the IH on $T_1' \equiv T_2'$, we know that $[T/\alpha]\,T_1' \equiv [T/\alpha]\,T_2'$. Since $[T/\alpha]\sigma_1 = \sigma_1$ and $[T/\alpha]\sigma_2 = \sigma_2$, so we are done by (C_REFINE).

Case (C_FUN): By the IHs and (C_FUN).

Case (C_FORALL): By the IH and (C_FORALL), possibly varying the bound variable name.

Case (C_SYM): By the IH and (C_SYM).

Case (C_TRANS): By the IHs and (C_TRANS). □

Lemma B.3.6 (Conversion of unrefined types). *If* $T_1 \equiv T_2$ *then* $\mathrm{unref}(T_1) \equiv \mathrm{unref}(T_2)$.

Proof. By induction on the derivation of $T_1 \equiv T_2$. □

Lemma B.3.7 (Compatibility is symmetric). $T_1 \parallel T_2$ *iff* $T_2 \parallel T_1$.

Proof. By induction on $T_1 \parallel T_2$.

Case (SIM_VAR): By (SIM_VAR).

Case (SIM_BASE): By (SIM_BASE).

Case (SIM_REFINEL): By (SIM_REFINER) and the IH.

Case (SIM_REFINER): By (SIM_REFINEL) and the IH.

Case (SIM_FUN): By (SIM_FUN) and the IHs.

Case (SIM_FORALL): By the IH and (SIM_FORALL). □

Lemma B.3.8 (Substitution preserves compatibility).
If $T_1 \parallel T_2$, *then* $[e/x]T_1 \parallel T_2$.

Proof. By induction on the compatibility relation.

Case (SIM_VAR): By (SIM_VAR).

Case (SIM_BASE): By (SIM_BASE).

Case (SIM_REFINEL): By (SIM_REFINEL) and the IH.

Case (SIM_REFINER): By (SIM_REFINER) and the IH.

Case (SIM_FUN): By (SIM_FUN) and the IHs.

Case (SIM_FORALL): By (SIM_FORALL) and the IH. □

Lemma B.3.9 (Type substitution preserves compatibility). *If* $T_1 \parallel T_2$ *then* $[T'/\alpha]T_1 \parallel [T'/\alpha]T_2$.

Proof. By induction on the compatibility relation.

Case (SIM_VAR): By (SIM_VAR) or reflexivity of the compatibility (proved easily).

Case (SIM_BASE): By (SIM_BASE).

Case (SIM_REFINEL): By (SIM_REFINEL) and the IH.

Case (SIM_REFINER): By (SIM_REFINER) and the IH.

Case (SIM_FUN): By (SIM_FUN) and the IHs.

Case (SIM_FORALL): By (SIM_FORALL) and the IH.

□

Lemma B.3.10 (Identity type substitution on one side preserves compatibility). *If* $T_1 \parallel T_2$ *then* $[\alpha/\alpha]T_1 \parallel T_2$.

Proof. Similar to Lemma B.3.9. □

Lemma 16 (Term weakening). *If x is fresh and $\Gamma \vdash T'$ then*

1. *$\Gamma, \Gamma' \vdash e : T$ implies $\Gamma, x{:}T', \Gamma \vdash e : T$,*
2. *$\Gamma, \Gamma' \vdash T$ implies $\Gamma, x{:}T', \Gamma' \vdash T$, and*
3. *$\vdash \Gamma, \Gamma'$ implies $\vdash \Gamma, x{:}T', \Gamma'$.*

Proof. By induction on e, T, and Γ'. The only interesting case is for terms where a run-time rule applies:

Case ((T_CONV),(T_EXACT),(T_FORGET)): The argument is the same for all terms, so: since $\vdash \Gamma, x{:}T', \Gamma'$, we can reapply (T_CONV), (T_EXACT), or (T_FORGET), respectively. In the rest of this proof, we will not bother considering these rules. □

Lemma 17 (Type weakening). *If α is fresh then*

1. *$\Gamma, \Gamma' \vdash e : T$ implies $\Gamma, \alpha, \Gamma \vdash e : T$,*
2. *$\Gamma, \Gamma' \vdash T$ implies $\Gamma, \alpha, \Gamma' \vdash T$, and*
3. *$\vdash \Gamma, \Gamma'$ implies $\vdash \Gamma, \alpha, \Gamma'$.*

Proof. By induction on e, T, and Γ'. The proof is similar to term weakening, Lemma 16. □

Lemma 18 (Term substitution). *If $\Gamma \vdash e' : T'$, then*

1. *if $\Gamma, x{:}T', \Gamma' \vdash e : T$ then $\Gamma, [e'/x]\Gamma' \vdash [e'/x]e : [e'/x]T$,*
2. *if $\Gamma, x{:}T', \Gamma' \vdash T$ then $\Gamma, [e'/x]\Gamma' \vdash [e'/x]T$, and*
3. *if $\vdash \Gamma, x{:}T', \Gamma'$ then $\vdash \Gamma, [e'/x]\Gamma'$.*

Proof. By induction on e, T, and Γ'. In the first two clauses, we are careful to leave Γ' as long as it is well formed. □

Lemma 19 (Type substitution). *If $\Gamma \vdash T'$ then*

1. *if $\Gamma, \alpha, \Gamma' \vdash e : T$, then $\Gamma, [T'/\alpha]\Gamma' \vdash [T'/\alpha]e : [T'/\alpha]T$,*
2. *if $\Gamma, \alpha, \Gamma' \vdash T$, then $\Gamma, [T'/\alpha]\Gamma' \vdash [T'/\alpha]T$, and*
3. *if $\vdash \Gamma, \alpha, \Gamma'$, then $\vdash \Gamma, [T'/\alpha]\Gamma'$.*

Proof. By induction on e, T, and Γ'. □

Lemma 20 (Lambda inversion). *If $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$, then there exists some T_2 such that*

1. *$\Gamma \vdash T_1$,*
2. *$\Gamma, x{:}T_1 \vdash e_{12} : T_2$, and*
3. *$x{:}T_1 \to T_2 \equiv \mathrm{unref}(T)$.*

Proof. By induction on the typing derivation. Cases not mentioned only apply to terms which are not lambdas.

Case (T_ABS): By inversion, we have $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. We find conversion immediately by reflexivity (Lemma B.3.1), since $\mathrm{unref}(T) = T = x{:}T_1 \to T_2$.

Case (T_CONV): We have $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$; by inversion, $T \equiv T'$ and $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T'$. By the IH on this second derivation, we find $\emptyset \vdash T_1$ and $x{:}T_1 \vdash e_{12} : T_2$ where, $\mathrm{unref}(T') \equiv x{:}T_1 \to T_2$. By weakening, we have $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $T' \equiv T$, we have $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T') \equiv \mathrm{unref}(T)$ by (C_TRANS).

Case (T_EXACT): $T = \{x{:}T' \mid e\}$, and we have $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : \{x{:}T' \mid e\}$; by inversion, $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T'$. By the IH, $\emptyset \vdash T_1$ and $x{:}T_1 \vdash e_{12} : T_2$, where $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T')$. By weakening, $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $\mathrm{unref}(T') = \mathrm{unref}(\{x{:}T' \mid e\})$, we have the conversion by (C_TRANS)): $x{:}T_1 \to T_2 \equiv \mathrm{unref}(T') = \mathrm{unref}(\{x{:}T' \mid e\})$.

Case (T_FORGET): We have $\Gamma \vdash \lambda x{:}T_1.\ e_{12} : T$; by inversion, $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : \{x{:}T \mid e\}$. By the IH on this latter derivation, we $\emptyset \vdash T_1$ and $x{:}T_1 \vdash e_{12} : T_2$, where $x{:}T_1 \to T_2 \equiv \mathrm{unref}(\{x{:}T \mid e\})$. By weakening, $\Gamma \vdash T_1$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Since $\mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$, we have by (C_TRANS) that $x{:}T_1 \to T_{12} \equiv \mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$. □

Lemma B.3.11 (Cast inversion). *If* $\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : T$, *then*

1. $\Gamma \vdash \sigma(T_1)$,
2. $\Gamma \vdash \sigma(T_2)$,
3. $T_1 \parallel T_2$
4. $\_{:}\sigma(T_1) \to \sigma(T_2) \equiv \mathrm{unref}(T)$ *(i.e., T_2 does not mention the dependent variable), and*
5. $\mathrm{AFV}(\sigma) \subseteq \mathrm{dom}(\Gamma)$.

Proof. By induction on the typing derivation, as for 20. □

Lemma B.3.12 (Type abstraction inversion). *If* $\Gamma \vdash \Lambda\alpha.\ e : T$, *then*

1. $\Gamma, \alpha \vdash e : T'$ *and*
2. $\forall\alpha.T' \equiv \mathrm{unref}(T)$.

Proof. By induction on the typing derivation, as for 20. □

Lemma 21 (Canonical forms). *If* $\emptyset \vdash v : T$, *then:*

1. *If* $\mathrm{unref}(T) = B$ *then* v *is* $k \in \mathcal{K}_B$ *for some* k.
2. *If* $\mathrm{unref}(T) = x{:}T_1 \to T_2$ *then*
   (a) v *is* $\lambda x{:}T_1'.\ e_{12}$ *and* $T_1' \equiv T_1$ *for some* x, T_1', *and* e_{12}, *or*
   (b) v *is* $\langle T_1' \Rightarrow T_2' \rangle^l_\sigma$ *and* $\sigma(T_1') \equiv T_1$ *and* $\sigma(T_2') \equiv T_2$ *for some* T_1', T_2', σ, *and* l.
3. *If* $\mathrm{unref}(T) = \forall\alpha.T'$ *then* v *is* $\Lambda\alpha.\ e$ *for some* e.

Proof. By induction on the typing derivation.

Case (T_VAR): Contradictory: variables are not values.

Case (T_CONST): $\emptyset \vdash k : T$ and $\mathrm{unref}(T) = B$; we are in case 1. By assumption, $k \in \mathcal{K}_B$.

Case (T_OP): Contradictory: $op\,(e_1, \ldots, e_n)$ is not a value.

Case (T_ABS): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : T$ and $T = \mathrm{unref}(T) = x{:}T_1 \to T_2$; we are in case 2a. Conversion is by reflexivity (Lemma B.3.1).

Case (T_APP): Contradictory: $e_1\ e_2$ is not a value.

Case (T_TABS): $\emptyset \vdash \Lambda\alpha.\ e : \forall\alpha.T$; we are in case 3. It is immediate that $v = \Lambda\alpha.\ e$, and conversion is by reflexivity (Lemma B.3.1).

Case (T_TAPP): Contradictory: $e\ T$ is not a value.

Case (T_CAST): $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : \_{:}\sigma(T_1) \to \sigma(T_2)$; we are in case 2b. It is immediate that $v = \langle T_1 \Rightarrow T_2 \rangle^l_\sigma$. Conversion is by reflexivity (Lemma B.3.1).

Case (T_CHECK): Contradictory: $\langle \{x{:}T \mid e_1\}, e_2, v \rangle^l$ is not a value.

Case (T_BLAME): Contradictory: $\Uparrow l$ is not a value.

Case (T_CONV): $\emptyset \vdash v : T$; by inversion, $\emptyset \vdash v : T'$ and $T' \equiv T$. We find an appropriate form for $\mathrm{unref}(T')$ by the IH on $\emptyset \vdash v : T'$. We go by cases, in each case reproving whatever case was found in the IH and finding conversions by (C_TRANS).

Case Case 1: $\mathrm{unref}(T) = B$ and $v = k \in \mathcal{K}_B$. Since $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$, we know that $\mathrm{unref}(T') = B$, which is all we needed to show.

Case Case 2a: $\mathrm{unref}(T) = x{:}T_1 \to T_2$ and $v = \lambda x{:}T_1''.\ e_{12}$ and $T_1'' \equiv T_1$. Since $T' \equiv T$, we have $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$ (Lemma B.3.6) and so $\mathrm{unref}(T') = x{:}T_1' \to T_2'$ for some T_1' and T_2' such that $T_1' \equiv T_1$ (Lemma B.3.3); by (C_TRANS), we have $T_1'' \equiv T_1'$.

Case Case 2b: $\mathrm{unref}(T) = x{:}T_1 \to T_2$ and $v = \langle T_1' \Rightarrow T_2' \rangle^l$ and $T_1' \equiv T_1$ and $T_2' \equiv T_2$. Since $T' \equiv T$, we have $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$ (Lemma B.3.6) and so $\mathrm{unref}(T') = x{:}T_1'' \to T_2''$ for some T_1'' and T_2'' such that $T_1'' \equiv T_1$ and $T_2'' \equiv T_2$ (Lemmas B.3.2 and B.3.3); by (C_TRANS), we have $T_1' \equiv T_1''$ and $T_2' \equiv T_2''$ as required.

Case Case 3: $\mathrm{unref}(T) = \forall\alpha.T_0$ and v is $\Lambda\alpha.\ e$. Since $T' \equiv T$, then $\mathrm{unref}(T') \equiv \mathrm{unref}(T)$ (Lemma B.3.6).

Case (T_EXACT): $\emptyset \vdash v : \{x{:}T \mid e\}$; by inversion, $\emptyset \vdash v : T$. Noting that $\mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$, we apply the IH. Unlike the previous case, we need not change the conversion—it is in terms of the unrefined type.

Case (T_FORGET): $\emptyset \vdash v : T$; by inversion $\emptyset \vdash v : \{x{:}T \mid e\}$. By the IH (noting $\mathrm{unref}(\{x{:}T \mid e\}) = \mathrm{unref}(T)$), so we use the IH's conversion directly. □

Theorem 5 (Progress). *If $\emptyset \vdash e : T$, then either*

1. *$e \longrightarrow e'$, or*
2. *e is a result r, i.e., a value or blame.*

Proof. By induction on the typing derivation.

Case (T_VAR): Contradictory: there is no derivation $\emptyset \vdash x : T$.

Case (T_CONST): $\emptyset \vdash k : \mathrm{ty}(k)$. In this case, $e = k$ is a result.

Case (T_OP): $\emptyset \vdash op\,(e_1, \ldots, e_n) : \sigma(T)$, where $\mathrm{ty}(op) = x_1 : T_1 \to \ldots \to x_n : T_n \to T$. By inversion, $\emptyset \vdash e_i : \sigma(T_i)$. Applying the IH from left to right, each of the e_i either steps or is a result.

Suppose everything to the left of e_i is a value. Then either e_i steps or is a result. If $e_i \longrightarrow e_i'$, then $op(v_1, \ldots, v_{i-1}, e_i, \ldots, e_n) \longrightarrow op(v_1, \ldots, v_{i-1}, e_i', \ldots, e_n)$ by (E_COMPAT). One the other hand, if e_i is a result, there are two cases. If $e_i = \Uparrow l$, then the original expression steps to $\Uparrow l$ by (E_BLAME). If e_i is a value, we can continue this process

for each of the operation's arguments. Eventually, all of the operations arguments are values. By value inversion (Lemma 13), we know that we can type each of these values at the exact refinement types we need by (T_EXACT). We assume that if $op\,(v_1, \ldots, v_n)$ is well defined on values satisfying the refinements in its type, so (E_OP) applies.

Case (T_ABS): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1 \to T_2)$. In this case, $e = \lambda x{:}T_1.\ e_{12}$ is a result.

Case (T_APP): $\emptyset \vdash e_1\ e_2 : [e_2/x]\,T_2$; by inversion, $\emptyset \vdash e_1 : (x{:}T_1 \to T_2)$ and $\emptyset \vdash e_2 : T_1$.

By the IH on the first derivation, e_1 steps or is a result. If e_1 steps, then the entire term steps by (E_COMPAT). In the latter case, if e_1 is blame, we step by (E_BLAME). So e_1 is a value, v_1.

By the IH on the second derivation, e_2 steps or is a result. If e_2 steps, then by (E_COMPAT). Otherwise, if e_2 is blame, we step by (E_BLAME). So e_2 is a value, v_2.

By canonical forms (Lemma 21) on $\emptyset \vdash e_1 : (x{:}T_1 \to T_2)$, there are two cases:

Case ($e_1 = \lambda x{:}T_1'.\ e_{12}$ and $T_1' \equiv T_1$): In this case, $(\lambda x{:}T_1'.\ e_{12})\,v_2 \longrightarrow [v_2/x]e_{12}$ by (E_BETA).

Case ($e_1 = \langle T_1' \Rightarrow T_2' \rangle^l_\sigma$ and $\sigma(T_1') \equiv T_1$ and $\sigma(T_2') \equiv T_2$): We know that $T_1' \parallel T_2'$ by cast inversion (Lemma B.3.11). We determine which step is taken by cases on T_1' and T_2'.

Case ($T_1' = B$):

Case ($T_2' = B'$): It must be the case that $B = B'$, since $B \parallel B'$. By (E_REFL), $\langle B \Rightarrow B \rangle^l_\sigma\ v_2 \longrightarrow v_2$.

Case ($T_2' = \alpha$ or $x{:}T_{21} \to T_{22}$ or $\forall\alpha.\,T_{22}$): Incompatible; contradictory.

Case ($T_2' = \{x{:}T_2'' \mid e\}$): If $T_2'' = B$, then by (E_CHECK), $\langle B \Rightarrow \{x{:}B \mid e\} \rangle^l_\sigma\ v_2 \longrightarrow \langle \sigma(\{x{:}B \mid e\}), \sigma([v_2/x]e), v_2 \rangle^l$. Otherwise, by (E_PRECHECK), we have:

$$\langle B \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l_\sigma\ v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l_{\sigma_1}\ (\langle B \Rightarrow T_2'' \rangle^l_{\sigma_2}\ v_2)$$

where $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2''|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_2'')}$.

Case ($T_1' = \alpha$):

Case ($T_2' = \alpha'$): It must be the case that $\alpha = \alpha'$, since $\alpha \parallel \alpha'$. By (E_REFL), $\langle \alpha \Rightarrow \alpha \rangle^l_\sigma\ v_2 \longrightarrow v_2$.

Case ($T_2' = B$ or $x{:}T_{21} \to T_{22}$ or $\forall\alpha.\,T_{22}$): Incompatible; contradictory.

Case ($T_2' = \{x{:}T_2'' \mid e\}$): If $T_2'' = \alpha$, then by (E_CHECK), $\langle \alpha \Rightarrow \{x{:}\alpha \mid e\} \rangle^l_\sigma\ v_2 \longrightarrow \langle \sigma(\{x{:}\alpha \mid e\}), \sigma([v_2/x]e), v_2 \rangle^l$. Otherwise,

$$\langle \alpha \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l_\sigma\ v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\} \rangle^l_{\sigma_1}\ (\langle \alpha \Rightarrow T_2'' \rangle^l_{\sigma_2}\ v_2)$$

where $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2''|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_2'')}$, by (E_PRECHECK).

Case ($T_1' = x{:}T_{11} \to T_{12}$):

Case ($T_2' = B$ or α or $\forall\alpha.\,T_{22}$): Incompatible; contradictory.

Case ($T_2' = x{:}T_{21} \to T_{22}$): If $T_1' = T_2'$, then $\langle T_1' \Rightarrow T_1' \rangle^l_\sigma\ v_2 \longrightarrow v_2$ by (E_REFL). If not, then

$$\begin{array}{l}\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22} \rangle^l_\sigma\ v_2 \longrightarrow \\ \lambda x{:}\sigma(T_{21}).\ \mathsf{let}\ y : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11} \rangle^l_{\sigma_1}\ x\ \mathsf{in}\ (\langle [y/x]\,T_{12} \Rightarrow T_{22} \rangle^l_{\sigma_2}\ (v_2\ y))\end{array}$$

for some fresh variable y, where $\sigma_i = \sigma|_{\mathrm{AFV}(T_{1\,i}) \cup \mathrm{AFV}(T_{2\,i})}$ ($i \in \{1, 2\}$), by (E_FUN).

Case $(T_2' = \{x{:}T_2'' \mid e\})$: If $T_1' = T_2''$, then $\langle T_1' \Rightarrow \{x{:}T_1' \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \langle \sigma(\{x{:}T_1' \mid e\}), \sigma([v_2/x]e), v_2\rangle^l$ by (E_CHECK). If not, then

$$\langle T_1' \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l_{\sigma_1}\, (\langle T_1' \Rightarrow T_2''\rangle^l_{\sigma_2}\, v_2)$$

, where $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2''|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_1')\cup\mathrm{AFV}(T_2'')}$, by (E_PRECHECK).

Case $(T_1' = \forall\alpha.\, T_{12})$:

Case $(T_2' = B$ or α or $x{:}T_{21} \to T_{22})$: Incompatible; contradictory.

Case $(T_2' = \forall\alpha.\, T_{22})$: If $T_1' = T_2'$, then $\langle T_1' \Rightarrow T_2'\rangle^l_\sigma\, v_2 \longrightarrow v_2$ by (E_REFL). If not, then $\langle \forall\alpha.\, T_{11} \Rightarrow \forall\alpha.\, T_{22}\rangle^l_\sigma\, v_2 \longrightarrow \Lambda\alpha.\, (\langle [\alpha/\alpha]T_{11} \Rightarrow T_{22}\rangle^l_\sigma\, (v_2\, \alpha))$ by (E_FORALL).

Case $(T_2' = \{x{:}T_2'' \mid e\})$: If $T_1' = T_2''$, then $\langle T_1' \Rightarrow \{x{:}T_1' \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \langle \sigma(\{x{:}T_1' \mid e\}), \sigma([v_2/x]e), v_2\rangle^l$ by (E_CHECK). If not, then $\langle T_1' \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \langle T_2'' \Rightarrow \{x{:}T_2'' \mid e\}\rangle^l_{\sigma_1}\, (\langle T_1' \Rightarrow T_2''\rangle^l_{\sigma_2}\, v_2)$ where $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2''|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_1')\cup\mathrm{AFV}(T_2'')}$, by (E_PRECHECK).

Case $(T_1' = \{x{:}T_1'' \mid e_1'\})$:

Case $(T_2' = B$ or α or $x{:}T_{21} \to T_{22}$ or $\forall\alpha.\, T_{22})$: We see

$$\langle \{x{:}T_1'' \mid e_1'\} \Rightarrow T_2'\rangle^l_\sigma\, v_2 \longrightarrow \langle T_1'' \Rightarrow T_2'\rangle^l_{\sigma'}\, v_2$$

where $\sigma' = \sigma|_{\mathrm{AFV}(T_1'')\cup\mathrm{AFV}(T_2')}$, by (E_FORGET).

Case $(T_2' = \{x{:}T_2'' \mid e_2'\})$: If $T_1' = T_2'$, then we immediately have $\langle T_1' \Rightarrow T_2'\rangle^l_\sigma\, v_2 \longrightarrow v_2$ by (E_REFL). If $T_1' = T_2''$, then

$$\langle T_1' \Rightarrow \{x{:}T_1' \mid e_2'\}\rangle^l_\sigma\, v_2 \longrightarrow \langle \sigma(\{x{:}T_1' \mid e_2'\}), \sigma([v_2/x]e_2'), v_2\rangle^l$$

by (E_CHECK). Otherwise,

$$\langle \{x{:}T_1'' \mid e_1'\} \Rightarrow \{x{:}T_2'' \mid e_2'\}\rangle^l_\sigma\, v_2 \longrightarrow \langle T_1'' \Rightarrow \{x{:}T_2'' \mid e_2'\}\rangle^l_{\sigma'}\, v_2$$

where $\sigma' = \sigma|_{\mathrm{AFV}(T_1'')\cup\mathrm{AFV}(\{x:T_2''|e_2'\})}$, by (E_FORGET).

Case (T_TABS): $\emptyset \vdash \Lambda\alpha.\, e' : \forall\alpha.\, T$. In this case, $\Lambda\alpha.\, e'$ is a result.

Case (T_TAPP): $\emptyset \vdash e_1\ T_2 : [T_2/\alpha]T_1$; by inversion, $\emptyset \vdash e_1 : \forall\alpha.\, T_1$ and $\emptyset \vdash T_2$. By the IH on the first derivation, e_1 steps or is a result. If $e_1 \longrightarrow e_1'$, then $e_1\ T_2 \longrightarrow e_1'\ T_2$ by (E_COMPAT). If $e_1 = \Uparrow l$, then $\Uparrow l\ T_2 \longrightarrow \Uparrow l$ by (E_BLAME).

If $e_1 = v_1$, then we know that $v_1 = \Lambda\alpha.\, e_1'$ by canonical forms (Lemma 21). We can see $(\Lambda\alpha.\, e_1')\ T_2 \longrightarrow [T_2/\alpha]e_1'$ by (E_TBETA).

Case (T_CAST): $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_\sigma : T_1 \to T_2$. In this case, $\langle T_1 \Rightarrow T_2\rangle^l_\sigma$ is a result.

Case (T_CHECK): $\emptyset \vdash \langle \{x{:}T \mid e_1\}, e_2, v\rangle^l : \{x{:}T \mid e_1\}$; by inversion, $\emptyset \vdash e_2 : \mathsf{bool}$. By the IH, either $e_2 \longrightarrow e_2'$ steps or $e_2 = r_2$. In the first case, $\langle \{x{:}T \mid e_1\}, e_2, v\rangle^l \longrightarrow \langle \{x{:}T \mid e_1\}, e_2', v\rangle^l$ by (E_COMPAT). In the second case, either $r_2 = \Uparrow l$ or $r_2 = v_2$. If we have blame, then the entire term steps by (E_BLAME). If we have a value, then we know that v_2 is either true or false, since it is typed at bool. If it is true, we step by (E_OK). Otherwise we step by (E_FAIL).

Case (T_BLAME): $\emptyset \vdash \Uparrow l : T$. In this case, $\Uparrow l$ is a result.

Case (T_CONV): $\emptyset \vdash e : T'$; by inversion, $\emptyset \vdash e : T$. By the IH, we see that $e \longrightarrow e'$ or $e = r$.

Case (T_EXACT): $\emptyset \vdash v : \{x{:}T \mid e\}$. Here, v is a result by assumption.

Case (T_FORGET): $\emptyset \vdash v : T$. Again, v is a result by assumption.

□

Lemma 22 (Context and type well formedness). *(1) If* $\Gamma \vdash e : T$, *then* $\vdash \Gamma$ *and* $\Gamma \vdash T$; *and (2) if* $\Gamma \vdash T$ *then* $\vdash \Gamma$.

Proof. By induction on the typing and well formedness derivations. □

Theorem 6 (Preservation). *If* $\emptyset \vdash e : T$ *and* $e \longrightarrow e'$, *then* $\emptyset \vdash e' : T$.

Proof. By induction on the typing derivation.

Case (T_VAR): Contradictory—we cannot have $\emptyset \vdash x : T$.

Case (T_CONST): $\emptyset \vdash k : \mathrm{ty}(k)$. Contradictory—values do not step.

Case (T_OP): $\emptyset \vdash op\,(e_1, \ldots, e_n) : \sigma(T)$. By cases on the step taken:

Case (E_REDUCE)/(E_OP): $op\,(v_1, \ldots, v_n) \longrightarrow [\![\mathrm{op}]\!]\,(v_1, \ldots, v_n)$. This case is by assumption.

Case (E_BLAME): $e_i = \Uparrow l$, and everything to its left is a value. By context and type well formedness (Lemma 22), $\emptyset \vdash \sigma(T)$. So by (T_BLAME), $\emptyset \vdash \Uparrow l : \sigma(T)$.

Case (E_COMPAT): Some $e_i \longrightarrow e_i'$. By the IH and (T_OP), using (T_CONV) to show that $\sigma(T) \equiv \sigma'(T)$ (Lemma 14).

Case (T_ABS): $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1 \to T_2)$. Contradictory—values do not step.

Case (T_APP): $\emptyset \vdash e_1\ e_2 : [e_2/x]T_2'$, with $\emptyset \vdash e_1 : (x{:}T_1' \to T_2')$ and $\emptyset \vdash e_2 : T_1'$, by inversion. By cases on the step taken.

Case (E_REDUCE)/(E_BETA): $(\lambda x{:}T_1.\ e_{12})\,v_2 \longrightarrow [v_2/x]e_{12}$. First, we have $\emptyset \vdash \lambda x{:}T_1.\ e_{12} : (x{:}T_1' \to T_2')$. By inversion for lambdas (Lemma 20), $x{:}T_1 \vdash e_{12} : T_2$. Moreover, $x{:}T_1 \to T_2 \equiv x{:}T_1' \to T_2'$, which means $T_2 \equiv T_2'$ (Lemma B.3.3).

By substitution, $\emptyset \vdash [v_2/x]e_{12} : [v_2/x]T_2$. We then see that $[v_2/x]T_2 \equiv [v_2/x]T_2'$ (Lemma 14), so (T_CONV) completes this case.

Case (E_REDUCE)/(E_REFL): $\langle T \Rightarrow T\rangle_\sigma^l\, v_2 \longrightarrow v_2$. By cast inversion (Lemma B.3.11), $\_ : \sigma(T) \to \sigma(T) \equiv x{:}T_1' \to T_2'$ and $\emptyset \vdash \sigma(T)$. In particular, we have $\sigma(T) \equiv T_2'$ and $\sigma(T) \equiv T_1'$ (Lemma B.3.3). By substitutivity of conversion (Lemma 14), $[v_2/x]\sigma(T) \equiv [v_2/x]T_2'$. Since $\sigma(T)$ is closed, we really know that $\sigma(T) \equiv [v_2/x]T_2'$.

By (C_SYM) and (C_TRANS), we have $T_1' \equiv \sigma(T) \equiv [v_2/x]T_2'$. By (T_CONV) on $\emptyset \vdash v_2 : T_1'$, we have $\emptyset \vdash v_2 : [v_2/x]T_2'$.

Case (E_REDUCE)/(E_FORGET): $\langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle_\sigma^l\, v_2 \longrightarrow \langle T_1 \Rightarrow T_2\rangle_{\sigma'}^l\, v_2$ where $\sigma' = \sigma|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)}$. We have $\sigma(T_1) = \sigma'(T_1)$ and $\sigma(T_2) = \sigma'(T_2)$. We restate the typing judgment and its inversion:

$$\begin{array}{l}\emptyset \vdash \langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle_\sigma^l\, v_2 : [v_2/y]T_2' \\ \emptyset \vdash \langle \{x{:}T_1 \mid e\} \Rightarrow T_2\rangle_\sigma^l : (y{:}T_1' \to T_2') \\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion (Lemma B.3.11), we know that $\emptyset \vdash \sigma(T_1)$ from $\emptyset \vdash \sigma(\{x{:}T_1 \mid e\})$ and $\emptyset \vdash \sigma(T_2)$—as well as $\_{:}\sigma(\{x{:}T_1 \mid e\}) \to \sigma(T_2) \equiv y{:}T_1' \to T_2'$ and $\{x{:}T_1 \mid e\} \parallel T_2$ and $\mathrm{AFV}(\sigma) \subseteq \emptyset$. Inverting this conversion (Lemma B.3.3), finding $\sigma(\{x{:}T_1 \mid e\}) \equiv T_1'$ and $\sigma(T_2) \equiv T_2'$. Then by (T_CONV) and (C_SYM), $\emptyset \vdash v_2 : \sigma(\{x{:}T_1 \mid e\})$; by (T_FORGET), $\emptyset \vdash v_2 : \sigma(T_1)$.

By (T_CAST), we have $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma'} : y{:}\sigma(T_1) \to \sigma(T_2)$, with $T_1 \parallel T_2$ iff $\{x{:}T_1 \mid e\} \parallel T_2$, and $\mathrm{AFV}(\sigma') \subseteq \mathrm{AFV}(\sigma) \subseteq \emptyset$. (Note, however, that y does not appear in $\sigma(T_2)$—we write it to clarify the substitutions below.)

By (T_APP), we find $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle^l_{\sigma'}\, v_2 : [v_2/y]\sigma(T_2)$. Since $\sigma(T_2) \equiv T_2'$, we have $[v_2/y]\sigma(T_2) \equiv [v_2/y]\,T_2'$ by Lemma 14. We are done by (T_CONV).

Case (E_REDUCE)/(E_PRECHECK):

$$\begin{array}{l}\langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \\ \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1}\, (\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2)\end{array}$$

where $\sigma_1 = \sigma|_{\mathrm{AFV}(\{x:T_2|e\})}$ and $\sigma_2 = \sigma|_{\mathrm{AFV}(T_1)\cup\mathrm{AFV}(T_2)}$. We have $\sigma(T_1) = \sigma_2(T_1)$ and $\sigma(T_2) = \sigma_1(T_2) = \sigma_2(T_2)$ and $\sigma(\{x{:}T_2 \mid e\}) = \sigma_1(\{x{:}T_2 \mid e\})$. We restate the typing judgment and its inversion:

$$\begin{array}{l}\emptyset \vdash \langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_\sigma\, v_2 : [v_2/y]\,T_2' \\ \emptyset \vdash \langle T_1 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_\sigma : y{:}T_1' \to T_2' \\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion (Lemma B.3.11), $\emptyset \vdash \sigma(T_1)$ and $\emptyset \vdash \sigma(\{x{:}T_2 \mid e\})$, and $y{:}\sigma(T_1) \to \sigma(\{x{:}T_2 \mid e\}) \equiv y{:}T_1' \to T_2'$ Also, $T_1 \parallel \{x{:}T_2 \mid e\}$ and $\mathrm{AFV}(\sigma) \subseteq \emptyset$.

By inversion on $\emptyset \vdash \sigma(\{x{:}T_2 \mid e\})$, we find $\emptyset \vdash \sigma(T_2)$. Next, $T_1 \parallel T_2$ iff $T_1 \parallel \{x{:}T_2 \mid e\}$, and $\mathrm{AFV}(\sigma_2) \subseteq \mathrm{AFV}(\sigma) \subseteq \emptyset$. Now by (T_CAST), we find $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2} : y{:}\sigma(T_1) \to \sigma(T_2)$. Note, however, that y does not occur in $\sigma(T_2)$.

We can take the convertible function types and see that their parts are convertible: $\sigma(T_1) \equiv T_1'$ and $\sigma(\{x{:}T_2 \mid e\}) \equiv T_2'$. Using the first conversion, we find $\emptyset \vdash v_2 : \sigma(T_1)$ by (T_CONV). By (T_APP), $\emptyset \vdash \langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2 : [v_2/y]\sigma(T_2)$, where $[v_2/y]\sigma(T_2) = \sigma(T_2)$.

By reflexivity of compatibility (easily proved) and (SIM_REFINER), $\sigma(T_2) \parallel \sigma(\{x{:}T_2 \mid e\})$. We have well formedness derivations for both types and $\mathrm{AFV}(\sigma_1) \subseteq \mathrm{AFV}(\sigma) \subseteq \emptyset$, as well, so $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1} : y{:}\sigma(T_2) \to \sigma(\{x{:}T_2 \mid e\})$ by (T_CAST). Again, y does not appear in $\sigma(e)$ or $\sigma(T_2)$. By (T_APP), we have $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1}\, (\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2) : [\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2/y]\sigma(\{x{:}T_2 \mid e\})$.

Since y is not in $\sigma(\{x{:}T_2 \mid e\})$, we can see:

$$[\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2/y]\sigma(\{x{:}T_2 \mid e\}) = \sigma(\{x{:}T_2 \mid e\}) = [v_2/y]\sigma(\{x{:}T_2 \mid e\})$$

By substitutivity of conversion (Lemma 14), we have $[v_2/y]\sigma(\{x{:}T_2 \mid e\}) \equiv [v_2/y]\,T_2'$. We can now apply (T_CONV) to find $\emptyset \vdash \langle T_2 \Rightarrow \{x{:}T_2 \mid e\}\rangle^l_{\sigma_1}\, (\langle T_1 \Rightarrow T_2\rangle^l_{\sigma_2}\, v_2) : [v_2/y]\,T_2'$.

Case (E_REDUCE)/(E_CHECK): $\langle T \Rightarrow \{x{:}T \mid e\}\rangle^l_\sigma\, v_2 \longrightarrow \langle \sigma(\{x{:}T \mid e\}), \sigma([v_2/x]e), v_2\rangle^{l'}$. Without loss of generality, we can suppose that x is fresh

for σ. We restate the typing judgment with its inversion:

$$\begin{array}{l}\emptyset \vdash \langle T \Rightarrow \{x{:}T \mid e\}\rangle^l_\sigma\ v_2 : [v_2/y]\,T_2' \\ \emptyset \vdash \langle T \Rightarrow \{x{:}T \mid e\}\rangle^l_\sigma : y{:}T_1' \to T_2' \\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion (Lemma B.3.11), $\emptyset \vdash \sigma(\{x{:}T \mid e\})$ and $\emptyset \vdash \sigma(T)$ and $\mathrm{AFV}(\sigma) \subseteq \emptyset$. Moreover, $y{:}\sigma(T) \to \sigma(\{x{:}T \mid e\}) \equiv y{:}T_1' \to T_2'$, where y does not occur in $\sigma(\{x{:}T \mid e\})$. This means that $\sigma(T) \equiv T_1'$ and $\sigma(\{x{:}T \mid e\}) \equiv T_2'$.

Using (T_CONV) and (C_SYM) with the first conversion shows $\emptyset \vdash v_2 : \sigma(T)$. By inversion on $\emptyset \vdash \sigma(\{x{:}T \mid e\})$, we see $x{:}\sigma(T) \vdash \sigma(e) : \mathsf{bool}$. By term substitution (Lemma 18), we find $\emptyset \vdash [v_2/x]\sigma(e) : \mathsf{bool}$. Since $[v_2/x]\sigma = \sigma$, by Lemma B.1.4, $[v_2/x]\sigma(e) = \sigma([v_2/x]e)$. Finally, $\sigma([v_2/x]e) \longrightarrow^* \sigma([v_2/x]e)$ by reflexivity (Lemma B.3.1).

(T_CHECK) (with (WF_EMPTY)) shows $\emptyset \vdash \langle \sigma(\{x{:}T \mid e\}), \sigma([v_2/x]e), v_2\rangle^l : \sigma(\{x{:}T \mid e\})$. By substitutivity of conversion (Lemma 14), $[v_2/y]\sigma(\{x{:}T \mid e\}) \equiv [v_2/y]\,T_2'$. Since y does not occur in $\sigma(\{x{:}T \mid e\})$, we know that $[v_2/y]\sigma(\{x{:}T \mid e\}) = \sigma(\{x{:}T \mid e\})$, so we can show that $\sigma(\{x{:}T \mid e\}) \equiv [v_2/y]\,T_2'$ by (C_SYM), and now $\emptyset \vdash \langle \sigma(\{x{:}T \mid e\}), \sigma([v_2/x]e), v_2\rangle^l : [v_2/y]\,T_2'$ by (T_CONV).

Case (E_REDUCE)/(E_FUN):

$$\begin{array}{l}\langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l_\sigma\ v_2 \longrightarrow \\ \lambda x{:}\sigma(T_{21}).\ \mathsf{let}\ z : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\ x\ \mathsf{in}\ (\langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2}\ (v_2\ z))\end{array}$$

for some fresh variable z, where $\sigma_i = \sigma|_{\mathrm{AFV}(T_{1\,i}) \cup \mathrm{AFV}(T_{2\,i})}$ $(i \in \{1,2\})$. Without loss of generality, we can suppose that x is fresh for σ. We have $\sigma(T_{j\,i}) = \sigma_i(T_{j\,i})$ $(j \in \{1,2\})$. We restate the typing judgment with its inversion:

$$\begin{array}{l}\emptyset \vdash \langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l_\sigma\ v_2 : [v_2/y]\,T_2' \\ \emptyset \vdash \langle x{:}T_{11} \to T_{12} \Rightarrow x{:}T_{21} \to T_{22}\rangle^l_\sigma : (y{:}T_1' \to T_2') \\ \emptyset \vdash v_2 : T_1'\end{array}$$

By cast inversion on the first derivation:

$$\begin{array}{cc}\emptyset \vdash \sigma(x{:}T_{11} \to T_{12}) & \emptyset \vdash \sigma(x{:}T_{21} \to T_{22}) \\ x{:}T_{11} \to T_{12} \parallel x{:}T_{21} \to T_{22} & \mathrm{AFV}(\sigma) \subseteq \emptyset \\ \_{:}\sigma(x{:}T_{11} \to T_{12}) \to \sigma(x{:}T_{21} \to T_{22}) \equiv y{:}T_1' \to T_2' & \end{array}$$

By inversion of this last (Lemma B.3.3):

$$\sigma(x{:}T_{11} \to T_{12}) \equiv T_1' \qquad \sigma(x{:}T_{21} \to T_{22}) \equiv T_2'$$

So by (T_CONV) and (C_SYM), we have $\emptyset \vdash v_2 : \sigma(x{:}T_{11} \to T_{12})$. By weakening (Lemma 16), $x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash v_2 : \sigma(x{:}T_{11} \to T_{12})$.

By inversion of the well formedness of the function types:

$$\emptyset \vdash \sigma(T_{11}) \qquad x{:}\sigma(T_{11}) \vdash \sigma(T_{12}) \qquad \emptyset \vdash \sigma(T_{21}) \qquad x{:}\sigma(T_{21}) \vdash \sigma(T_{22})$$

By weakening (Lemma 16), we find $x{:}\sigma(T_{21}) \vdash \sigma(T_{11})$ and $x{:}\sigma(T_{21}) \vdash \sigma(T_{21})$. By compatibility:

$$T_{11} \parallel T_{21} \qquad\qquad T_{12} \parallel T_{22}$$

Since $\mathrm{AFV}(\sigma_1) \subseteq \mathrm{AFV}(\sigma) \subseteq \emptyset$, we have $x{:}\sigma(T_{21}) \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1} : (\_{:}\sigma(T_{21}) \to \sigma(T_{11}))$ by (T_CAST) (compatibility is symmetric, per Lemma B.3.7). By (T_APP) and (T_VAR), we can see $x{:}\sigma(T_{21}) \vdash \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\, x : [x/\_]\sigma(T_{11}) = \sigma(T_{11})$. Again by (T_APP), we have $x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash v_2\, z : [z/x]\sigma(T_{12})$. By weakening (Lemma 16) and substitution (Lemma 18), we have the following two derivations:

$$\begin{array}{l} x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash [z/x]\sigma(T_{12}) = [z/x]\sigma_2(T_{12}) = \sigma_2([z/x]\,T_{12}) \\ x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash \sigma(T_{22}) \end{array}$$

By (T_CAST) and Lemma B.3.8:

$$x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash \langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2} : (y{:}[z/x]\sigma(T_{12}) \to \sigma(T_{22}))$$

Noting that y is free here. By (T_APP):

$$\begin{array}{rl} x{:}\sigma(T_{21}), z{:}\sigma(T_{11}) \vdash & \langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^l\,(v_2\, z) \\ : & [v_2\, z/y]\,T_{22} (= T_{22}) \end{array}$$

Finally, by (T_ABS) and (T_APP):

$$\emptyset \vdash \lambda x{:}\sigma(T_{21}).\ \begin{array}{l} \mathsf{let}\ z : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\, x\ \mathsf{in} \\ \langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2}\,(v_2\, z) \end{array} \ : x{:}\sigma(T_{21}) \to \sigma(T_{22})$$

since $[\langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\, x/z]\sigma(T_{22}) = \sigma(T_{22})$.

Since y is not in $x{:}\sigma(T_{21}) \to \sigma(T_{22})$, we can see that $x{:}\sigma(T_{21}) \to \sigma(T_{22}) = [v_2/y](x{:}\sigma(T_{21}) \to \sigma(T_{22}))$. Using this fact with substitutivity of conversion (Lemma 14), we find $x{:}\sigma(T_{21}) \to \sigma(T_{22}) \equiv [v_2/y]\,T_2'$. So—finally—by (T_CONV) we have:

$$\emptyset \vdash \lambda x{:}\sigma(T_{21}).\ \mathsf{let}\ z : \sigma(T_{11}) = \langle T_{21} \Rightarrow T_{11}\rangle^l_{\sigma_1}\, x\ \mathsf{in}\ \langle [z/x]\,T_{12} \Rightarrow T_{22}\rangle^l_{\sigma_2}\,(v_2\, z) : [v_2/y]\,T_2'$$

Case (E_REDUCE)/(E_FORALL): $\langle \forall\alpha.\, T_1 \Rightarrow \forall\alpha.\, T_2\rangle^l_\sigma\, v_2 \longrightarrow (\Lambda\alpha.\ \langle [\alpha/\alpha]\,T_1 \Rightarrow T_2\rangle^l_\sigma\,(v\,\alpha))$ Without loss of generality, we can suppose that α is fresh for σ. We restate the typing and its inversion:

$$\begin{array}{l} \emptyset \vdash \langle \forall\alpha.\, T_1 \Rightarrow \forall\alpha.\, T_2\rangle^l_\sigma\, v_2 : [v_2/x]\,T_2' \\ \emptyset \vdash \langle \forall\alpha.\, T_1 \Rightarrow \forall\alpha.\, T_2\rangle^l_\sigma : x{:}T_1' \to T_2' \\ \emptyset \vdash v_2 : T_1' \end{array}$$

By cast inversion (Lemma B.3.11):

$$\begin{array}{ll} \emptyset \vdash \sigma(\forall\alpha.\, T_1) & \emptyset \vdash \sigma(\forall\alpha.\, T_2) \\ \forall\alpha.\, T_1 \parallel \forall\alpha.\, T_2 & \mathrm{AFV}(\sigma) \subseteq \emptyset \\ \multicolumn{2}{l}{\_{:}\sigma(\forall\alpha.\, T_1) \to \sigma(\forall\alpha.\, T_2) \equiv x{:}T_1' \to T_2'} \end{array}$$

By inversion of this last $\sigma(\forall\alpha.\, T_1) \equiv T_1'$ and $\sigma(\forall\alpha.\, T_2) \equiv T_2'$ (Lemma B.3.3). By

(T_CONV) and (C_SYM), $\emptyset \vdash v_2 : \sigma(\forall\alpha.\, T_1) = \forall\alpha.\sigma(T_1)$. By type variable weakening (Lemma 17), (WF_TVAR), and (T_TAPP), we have:

$$\alpha \vdash v_2\, \alpha : [\alpha/\alpha]\sigma(T_1) = \sigma([\alpha/\alpha]\, T_1)$$

. Note that $\sigma([\alpha/\alpha]\, T_1)$ may be syntactically different from $\sigma(T_1)$. By inversion of the universal type's well formedness, compatibility, type weakening (Lemma 17), type substitution (Lemma 19) and Lemma B.3.10:

$$\alpha \vdash \sigma([\alpha/\alpha]\, T_1) \qquad \alpha \vdash \sigma(T_2) \qquad [\alpha/\alpha]\, T_1 \parallel T_2$$

So by (T_CAST), $\alpha \vdash \langle [\alpha/\alpha]\, T_1 \Rightarrow T_2 \rangle^l_\sigma : (x{:}\sigma([\alpha/\alpha]\, T_1) \to \sigma(T_2))$, noting that x does not occur in $\sigma(T_2)$. By (T_APP), $\alpha \vdash \langle [\alpha/\alpha]\, T_1 \Rightarrow T_2 \rangle^l_\sigma\, (v_2\, \alpha) : [v_2\, \alpha/x]\sigma(T_2) = \sigma(T_2)$. By (T_TABS), $\emptyset \vdash \Lambda\alpha.\, (\langle [\alpha/\alpha]\, T_1 \Rightarrow T_2 \rangle^l_\sigma\, (v\, \alpha)) : \forall\alpha.\sigma(T_2)$.

We know that $\forall\alpha.\sigma(T_2) \equiv T_2'$, so by term substitutivity of conversion (Lemma 14), $[v_2/x]\forall\alpha.\sigma(T_2) \equiv [v_2/x]\, T_2'$. Since x is not in $\forall\alpha.\sigma(T_2)$, we know that $\forall\alpha.\sigma(T_2) \equiv [v_2/x]\, T_2'$. Now we can see by (T_CONV) that $\emptyset \vdash \Lambda\alpha.\, (\langle [\alpha/\alpha]\, T_1 \Rightarrow T_2 \rangle^l_\sigma\, (v\, \alpha)) : [v_2/x]\, T_2'$.

Case (E_COMPAT): $E\,[e] \longrightarrow E\,[e']$ when $e \longrightarrow e'$ By cases on E:

Case ($E = [\,]\, e_2$, $e_1 \longrightarrow e_1'$): By the IH and (T_APP).

Case ($E = v_1\, [\,]$, $e_2 \longrightarrow e_2'$): By the IH, (T_APP), and (T_CONV), since $[e_2/x]\, T_2 \equiv [e_2'/x]\, T_2$ by reflexivity (Lemma B.3.1) and substitutivity (Lemma 14).

Case (E_BLAME): $E\,[\Uparrow l] \longrightarrow \Uparrow l$ $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma 22), we know that $\emptyset \vdash T$. We then have $\emptyset \vdash \Uparrow l : T$ by (T_BLAME).

Case (T_TABS): $\emptyset \vdash \Lambda\alpha.\, e : \forall\alpha.\, T$. This case is contradictory—values do not step.

Case (T_TAPP): $\emptyset \vdash e\ T : [T/\alpha]\, T'$. By cases on the step taken.

Case (E_REDUCE)/(E_TBETA): $(\Lambda\alpha.\, e')\ T \longrightarrow [T/\alpha]e'$ We restate the typing derivation and its inversion:

$$\emptyset \vdash (\Lambda\alpha.\, e')\ T : [T/\alpha]\, T' \qquad \emptyset \vdash \Lambda\alpha.\, e' : \forall\alpha.\, T' \qquad \emptyset \vdash T$$

By type abstraction inversion (Lemma B.3.12): $\alpha \vdash e' : T''$ and $\forall\alpha.\, T'' \equiv \forall\alpha.\, T'$; by inversion of this last (Lemma B.3.5), $T'' \equiv T'$.

By type variable substitution (Lemma 19), $\emptyset \vdash [T/\alpha]e' : [T/\alpha]\, T''$. By type substitutivity of conversion (Lemma 15), $[T/\alpha]\, T'' \equiv [T/\alpha]\, T'$. (T_CONV) gives us $\emptyset \vdash [T/\alpha]e' : [T/\alpha]\, T'$ as desired.

Case (E_COMPAT): $E\,[e] \longrightarrow E\,[e']$, where $E = [\,]\ T$. By the IH and (T_TAPP).

Case (E_BLAME): $E\,[\Uparrow l] \longrightarrow \Uparrow l$. $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma 22), we know that $\emptyset \vdash T$. So we see $\emptyset \vdash \Uparrow l : T$ by (T_BLAME).

Case (T_CAST): $\emptyset \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : \sigma(T_1) \to \sigma(T_2)$. This case is contradictory—values do not step.

Case (T_CHECK): $\emptyset \vdash \langle \{x{:}T \mid e_1\}, e_2, v \rangle^l : \{x{:}T \mid e_1\}$. By cases on the step taken.

Case (E_REDUCE)/(E_OK): $\langle\{x{:}T \mid e_1\}, \mathsf{true}, v\rangle^l \longrightarrow v$. By inversion, $\emptyset \vdash v : T$ and $\emptyset \vdash \{x{:}T \mid e\}$; we also have $[v/x]e_1 \longrightarrow^* \mathsf{true}$. By (WF_EMPTY) and the assumption that $[v/x]e \longrightarrow^* \mathsf{true}$, we can find $\emptyset \vdash v : \{x{:}T \mid e\}$ by (T_EXACT).

Case (E_REDUCE)/(E_FAIL): $\langle\{x{:}T \mid e_1\}, \mathsf{false}, v\rangle^l \longrightarrow \Uparrow l$ We have $\emptyset \vdash \{x{:}T \mid e\}$ by inversion. By (WF_EMPTY) and (T_BLAME), $\emptyset \vdash \Uparrow l : \{x{:}T \mid e\}$.

Case (E_COMPAT): $E\,[e] \longrightarrow E\,[e']$, where $E = \langle\{x{:}T \mid e_1\}, [\,], v\rangle^l$. By the IH on e, we know that $\emptyset \vdash e' : \mathsf{bool}$. We still have $\emptyset \vdash \{x{:}T \mid e_1\}$ and $\emptyset \vdash v : T$ from our original derivation. Since $[v/x]e_1 \longrightarrow^* e$ and $e \longrightarrow e'$, then $[v/x]e_1 \longrightarrow^* e'$. Therefore, $\emptyset \vdash \langle\{x{:}T \mid e_1\}, e', v\rangle^l : \{x{:}T \mid e_1\}$ by (T_CHECK).

Case (E_BLAME): $E\,[\Uparrow l] \longrightarrow \Uparrow l$. $\emptyset \vdash E\,[\Uparrow l] : T$ by assumption. By type well formedness (Lemma 22), we know that $\emptyset \vdash T$. So $\emptyset \vdash \Uparrow l : T$ by (T_BLAME).

Case (T_BLAME): $\emptyset \vdash \Uparrow l : T$. This case is contradictory—blame does not step.

Case (T_CONV): $\emptyset \vdash e : T'$; by inversion we have $\emptyset \vdash e : T$ and $T \equiv T'$ and $\emptyset \vdash T'$ (and, trivially, $\vdash \emptyset$). By the IH on the first derivation, we know that $\emptyset \vdash e' : T$. By (T_CONV), we can see that $\emptyset \vdash e' : T'$.

Case (T_EXACT): $\emptyset \vdash v : \{x{:}T \mid e\}$. This case is contradictory—values do not step.

Case (T_FORGET): $\emptyset \vdash v : T$. This case is contradictory—values do not step.

□

B.4 Parametricity

This section proves parametricity; an outline of the proof is described in Section 3.4.2. We write $R_{T,\theta,\delta}$ for $\{(r_1, r_2) \mid r_1 \sim r_2 : T; \theta; \delta\}$.

Lemma 23 (Term compositionality). *If $\theta_1(\delta_1(e)) \longrightarrow^* v_1$ and $\theta_2(\delta_2(e)) \longrightarrow^* v_2$ then $r_1 \sim r_2 : T; \theta; \delta[(v_1, v_2)/x]$ iff $r_1 \sim r_2 : [e/x]T; \theta; \delta$.*

Proof. By induction on the (simple) structure of T, proving both directions simultaneously. We treat the case where $r_1 = r_2 = \Uparrow l$ separately from the induction, since it is the same easy proof in all cases: $\Uparrow l \sim \Uparrow l : T; \theta; \delta$ irrespective of T and δ. So for the rest of proof, we know $r_1 = v_1$ and $r_2 = v_2$. Only the refinement case is interesting.

Case ($T = \{y{:}T' \mid e'\}$): We show both directions simultaneously, where $x \neq y$, i.e., y is fresh. By the IH for T', we know that

$$v_1 \sim v_2 : T'; \theta; \delta[(e_1, e_2)/x] \text{ iff } v_1 \sim v_2 : [e/x]T'; \theta; \delta.$$

It remains to show that the values satisfy their refinements.

That is, we must show:

$$\theta_1(\delta_1([v_1/y][e_1/x]e')) \longrightarrow^* \mathsf{true} \text{ iff } \theta_1(\delta_1([v_1/y][e/x]e')) \longrightarrow^* \mathsf{true}$$

$$\theta_2(\delta_2([v_2/y][e_2/x]e')) \longrightarrow^* \mathsf{true} \text{ iff } \theta_2(\delta_2([v_2/y][e/x]e')) \longrightarrow^* \mathsf{true}$$

So let:

$$\begin{array}{rcccccl} \sigma_1 & = & \theta_1\delta_1[\delta_1(e)/x, v_1/y] & \longrightarrow^* & \theta_1\delta_1[e_1/x, v_1/y] & = & \sigma_1' \\ \sigma_2 & = & \theta_2\delta_2[\delta_2(e)/x, v_2/y] & \longrightarrow^* & \theta_2\delta_2[e_2/x, v_2/y] & = & \sigma_2' \end{array}$$

We have $\sigma_1 \longrightarrow^* \sigma_1'$ by reflexivity except for $\delta_1(e) \longrightarrow^* e_1$, which we have by assumption; likewise, we have $\sigma_2 \longrightarrow^* \sigma_2'$. Then $\sigma_i(e')$ and $\sigma_i'(e')$ coterminate (Lemma 11), and we are done. □

Lemma B.4.1 (Term Weakening/Strengthening). *If* $x \notin T$, *then* $r_1 \sim r_2 : T; \theta; \delta[(e_1, e_2)/x]$ *iff* $r_1 \sim r_2 : T; \theta; \delta$.

Proof. Similar to Lemma 23. □

Lemma B.4.2 (Type Weakening/Strengthening). *If* $\alpha \notin T$, *then* $r_1 \sim r_2 : T; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$ *iff* $r_1 \sim r_2 : T; \theta; \delta$.

Proof. Similar to Lemma 23. □

Lemma 24 (Type compositionality).
$r_1 \sim r_2 : T; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$ *iff* $r_1 \sim r_2 : [T'/\alpha]T; \theta; \delta$.

Proof. By induction on the (simple) structure of T, proving both directions simultaneously. As for Lemma 23, we treat the case where $r_1 = r_2 = \Uparrow l$ separately from the induction, since it is the same easy proof in all cases: $\Uparrow l \sim \Uparrow l : T; \theta; \delta$ irrespective of T and δ. So for the rest of proof, we know $r_1 = v_1$ and $r_2 = v_2$. Here, the interesting case is for function types, where we must deal with some asymmetries in the definition of the logical relation. We also include the case for quantified types.

Case $(T = x{:}T_1 \to T_2)$: There are two cases:

Case (⇒): Given $v_1 \sim v_2 : (x{:}T_1 \to T_2); \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$, we wish to show that $v_1 \sim v_2 : [T'/\alpha](x{:}T_1 \to T_2); \theta; \delta$. Let $v_1' \sim v_2' : [T'/\alpha]T_1; \theta; \delta$. We must show that $v_1\, v_1' \simeq v_2\, v_2' : [T'/\alpha]T_2; \theta; \delta[(v_1', v_2')/x]$. By the IH on T_1, $v_1' \sim v_2' : T_1; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. By assumption,

$$v_1\, v_1' \simeq v_2\, v_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[(v_1', v_2')/x].$$

These normalize to

$$r_1' \sim r_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[(v_1', v_2')/x].$$

Since $x \notin T'$, Lemma B.4.1 gives $R_{T',\theta,\delta} = R_{T',\theta,\delta[(v_1',v_2')/x]}$ and so

$$r_1' \sim r_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta[(v_1',v_2')/x]}, \theta_1(\delta_1([v_1'/x]T')), \theta_2(\delta_2([v_2'/x]T'))]; \delta[(v_1', v_2')/x].$$

By the IH on T_2, $r_1' \sim r_2' : [T'/\alpha]T_2; \theta; \delta[(v_1', v_2')/x]$. By expansion, $v_1\, v_1'' \simeq v_2\, v_2'' : [T'/\alpha]T_2; \theta; \delta[(v_1', v_2')/x]$.

Case (⇐): This case is similar: Given $v_1 \sim v_2 : [T'/\alpha](x{:}T_1 \to T_2); \theta; \delta$, we wish to show that $v_1 \sim v_2 : (x{:}T_1 \to T_2); \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. Let $v_1' \sim v_2' : T_1; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. We must show that

$$v_1\, v_1' \simeq v_2\, v_2' : T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[(v_1', v_2')/x].$$

By the IH on T_1, $v_1' \sim v_2' : [T'/\alpha]T_1; \theta; \delta$. By assumption, $v_1\, v_1' \simeq v_2\, v_2' : [T'/\alpha]T_2; \theta; \delta[(v_1', v_2')/x]$. These normalize to $r_1' \simeq r_2' : [T'/\alpha]T_2; \theta; \delta[(v_1', v_2')/x]$.

By the IH on T_2,

$$r_1' \simeq r_2' : [T'/\alpha]\,T_2;\\ \theta[\alpha \mapsto R_{T',\theta,\delta[(v_1',v_2')/x]}, \theta_1(\delta_1([v_1'/x]\,T')), \theta_2(\delta_2([v_2'/x]\,T'))];\\ \delta[(v_1', v_2')/x].$$

Since $x \notin T'$, Lemma B.4.1 gives

$$r_1' \simeq r_2' : [T'/\alpha]\,T_2; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta[(v_1', v_2')/x].$$

Finally, by expansion

$$v_1\ v_1' \simeq v_2\ v_2' : [T'/\alpha]\,T_2;\\ \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))];\\ \delta[(v_1', v_2')/x].$$

Case $(T = \forall\alpha'.T_0)$: There are two cases:

Case $(\Rightarrow)$: Given $v_1 \sim v_2 : \forall\alpha'.T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$, we wish to show that $v_1 \sim v_2 : \forall\alpha'.([T'/\alpha]\,T_0); \theta; \delta$. Let a relation R and closed types T_1 and T_2 be given. By assumption, we know that $v_1\ T_1 \simeq v_2\ T_2 : T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))][\alpha' \mapsto R, T_1, T_2]; \delta$. They normalize to $r_1' \sim r_2' : T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))][\alpha' \mapsto R, T_1, T_2]; \delta$. By the IH, $r_1' \sim r_2' : [T'/\alpha]\,T_0; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. By expansion, $v_1\ T_1 \simeq v_2\ T_2 : [T'/\alpha]\,T_0; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. Then, $v_1 \sim v_2 : \forall\alpha'.([T'/\alpha]\,T_0); \theta; \delta$.

Case $(\Leftarrow)$: This case is similar: given $v_1 \sim v_2 : \forall\alpha'.([T'/\alpha]\,T_0); \theta; \delta$, we wish to show that $v_1 \sim v_2 : \forall\alpha'.T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. Let a relation R and closed types T_1 and T_2 be given. By assumption, we know that $v_1\ T_1 \simeq v_2\ T_2 : [T'/\alpha]\,T_0; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. They normalize to $r_1' \sim r_2' : [T'/\alpha]\,T_0; \theta[\alpha' \mapsto R, T_1, T_2]; \delta$. By the IH, $r_1' \sim r_2' : T_0; \theta[\alpha' \mapsto R, T_1, T_2][\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. By expansion, $v_1\ T_1 \simeq v_2\ T_2 : T_0; \theta[\alpha' \mapsto R, T_1, T_2][\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$. Then, $v_1 \sim v_2 : \forall\alpha'.T_0; \theta[\alpha \mapsto R_{T',\theta,\delta}, \theta_1(\delta_1(T')), \theta_2(\delta_2(T'))]; \delta$.

□

Lemma 25 (Convertibility). *If $T_1 \equiv T_2$ then $r_1 \sim r_2 : T_1; \theta; \delta$ iff $r_1 \sim r_2 : T_2; \theta; \delta$.*

Proof. By induction on the conversion relation, leaving θ and δ general. The case where $r_1 = r_2 = \Uparrow l$ is immediate, so we only need to consider the case where $r_1 = v_1$ and $r_2 = v_2$.

Case (C_VAR): It must be that $T_1 = T_2 = \alpha$, so we are done immediately.

Case (C_BASE): It must be that $T_1 = T_2 = B$, so we are done immediately.

Case (C_REFINE): We have that $T_1 = \{x{:}T_1' \mid \sigma_1(e)\}$ and $T_2 = \{x{:}T_2' \mid \sigma_2(e)\}$, where $T_1' \equiv T_2'$ and $\sigma_1 \longrightarrow^* \sigma_2$.

By cotermination (Lemma 11):

$$[v_1/x](\theta_1(\delta_1(\sigma_1(e)))) \longrightarrow^* \mathsf{true} \text{ iff } [v_1/x](\theta_1(\delta_1(\sigma_2(e)))) \longrightarrow^* \mathsf{true}\\ [v_2/x](\theta_2(\delta_2(\sigma_1(e)))) \longrightarrow^* \mathsf{true} \text{ iff } [v_2/x](\theta_2(\delta_2(\sigma_2(e)))) \longrightarrow^* \mathsf{true}.$$

We have $[v_i/x](\theta_i(\delta_i(\sigma_j(e)))) = \sigma_j([v_i/x](\theta_i(\delta_i(e))))$ for $i, j \in \{1, 2\}$ since all substitutions here are closing.

Case (C_FUN): We have that $T_1 = x{:}T_{11} \to T_{12} \equiv x{:}T_{21} \to T_{22} = T_2$.

Let $v_1' \sim v_2' : T_{21}; \theta; \delta$ be given; we must show that $v_1\ v_1' \simeq v_2\ v_2' : T_{22}; \theta; \delta[(v_1', v_2')/x]$.

By the IH, we know that $v_1' \sim v_2' : T_{11}; \theta; \delta$, so we know that $v_1\ v_1' \simeq v_2\ v_2' : T_{12}; \theta; \delta[(v_1', v_2')/x]$. We are done by another application of the IH.

The other direction is similar.

Case (C_FORALL): We have that $T_1 = \forall\alpha.T_1' \equiv \forall\alpha.T_2' = T_2$.

Let R, T, and T' be given. We must show that $v_1\ T \simeq v_2\ T' : T_2'; \theta[\alpha \mapsto R, T, T']; \delta$. We know that $v_1\ T \simeq v_2\ T' : T_1'; \theta[\alpha \mapsto R, T, T']; \delta$, so we are done by the IH.

The other direction is similar.

Case (C_SYM): By the IH.

Case (C_TRANS): By the IHs. □

Lemma 26 (Cast reflexivity). *If* $\vdash \Gamma$ *and* $T_1 \parallel T_2$ *and* $\Gamma \vdash \sigma(T_1) \simeq \sigma(T_1) : *$ *and* $\Gamma \vdash \sigma(T_2) \simeq \sigma(T_2) : *$ *and* $\mathrm{AFV}(\sigma) \subseteq \mathrm{dom}(\Gamma)$, *then* $\Gamma \vdash \langle T_1 \Rightarrow T_2\rangle^l_\sigma \simeq \langle T_1 \Rightarrow T_2\rangle^l_\sigma : \sigma(\_{:}T_1 \to T_2)$.

Proof. By induction on $cc(\langle T_1 \Rightarrow T_2\rangle^l)$. We omit the majority of this proof, but we leave in the case when $T_1 = T_2$ to highlight the need for the (E_REFL) reduction rule.

Case $T_1 = T_2$: Given $\Gamma \vdash \theta; \delta$, we wish to show that

$$\langle\theta_1(\delta_1(T_1)) \Rightarrow \theta_1(\delta_1(T_1))\rangle^l_\sigma \simeq \langle\theta_2(\delta_2(T_1)) \Rightarrow \theta_2(\delta_2(T_1))\rangle^l_\sigma : \sigma(T_1 \to T_1); \theta; \delta.$$

Let $v_1 \sim v_2 : \sigma(T_1); \theta; \delta$. We must show that

$$\begin{array}{l}\langle\theta_1(\delta_1(T_1)) \Rightarrow \theta_1(\delta_1(T_1))\rangle^l_\sigma\ v_1 \simeq \\ \langle\theta_2(\delta_2(T_1)) \Rightarrow \theta_2(\delta_2(T_1))\rangle^l_\sigma\ v_2 : \sigma(T_1); \theta; \delta[(v_1, v_2)/z]\end{array}$$

for fresh z. By (E_REFL), these normalize to $v_1 \sim v_2 : \sigma(T_1); \theta; \delta[(v_1, v_2)/z]$. Lemma B.4.1 finishes the case.

□

Theorem 8 (Parametricity). *(1) If* $\Gamma \vdash e : T$ *then* $\Gamma \vdash e \simeq e : T$; *and (2) if* $\Gamma \vdash T$ *then* $\Gamma \vdash T \simeq T : *$.

Proof. By simultaneous induction on the derivations with case analysis on the last rule used.

Case (T_VAR): Let $\Gamma \vdash \theta; \delta$. We wish to show that $\theta_1(\delta_1(x)) \simeq \theta_2(\delta_2(x)) : T; \theta; \delta$, which follows from the assumption.

Case (T_CONST): By the assumption that constants are assigned correct types.

Case (T_OP): By the assumption that operators are assigned correct types (and the IHs for the operator's arguments).

Case (T_ABS): We have $e = \lambda x{:}T_1.\ e_{12}$ and $T = x{:}T_1 \to T_2$ and $\Gamma, x{:}T_1 \vdash e_{12} : T_2$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(\lambda x{:}T_1.\ e_{12})) \sim \theta_2(\delta_2(\lambda x{:}T_1.\ e_{12})) : (x{:}T_1 \to T_2); \theta; \delta.$$

Let $v_1 \sim v_2 : T_1; \theta; \delta$. We must show that

$$(\lambda x{:}\theta_1(\delta_1(T_1)).\ \theta_1(\delta_1(e_{12})))\ v_1 \simeq (\lambda x{:}\theta_2(\delta_2(T_1)).\ \theta_2(\delta_2(e_{12})))\ v_2 : T_2; \theta; \delta[(v_1, v_2)/x].$$

Since

$$\begin{array}{rcl}(\lambda x{:}\theta_1(\delta_1(T_1)).\ \theta_1(\delta_1(e_{12})))\ v_1 & \longrightarrow & [v_1/x]\theta_1(\delta_1(e_{12})) \\ (\lambda x{:}\theta_2(\delta_2(T_1)).\ \theta_2(\delta_2(e_{12})))\ v_2 & \longrightarrow & [v_2/x]\theta_2(\delta_2(e_{12})),\end{array}$$

it suffices to show

$$[v_1/x]\theta_1(\delta_1(e_{12})) \simeq [v_2/x]\theta_2(\delta_2(e_{12})) : T_2; \theta; \delta[(v_1, v_2)/x].$$

By the IH, $\Gamma, x{:}T_1 \vdash e_{12} \simeq e_{12} : T_2$. The fact that $\Gamma, x{:}T_1 \vdash \theta; \delta[(v_1, v_2)/x]$ finishes the case.

Case (T_APP): We have $e = e_1\ e_2$ and $\Gamma \vdash e_1 : x{:}T_1 \to T_2$ and $\Gamma \vdash e_2 : T_1$ and $T = [e_2/x]\,T_2$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(e_1\ e_2)) \simeq \theta_2(\delta_2(e_1\ e_2)) : [e_2/x]\,T_2; \theta; \delta.$$

By the IH,

$$\begin{array}{l}\theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_2)) : x{:}T_1 \to T_2; \theta; \delta, \text{ and} \\ \theta_1(\delta_1(e_2)) \simeq \theta_2(\delta_2(e_2)) : T_1; \theta; \delta.\end{array}$$

These normalize to $r_{11} \sim r_{12} : x{:}T_1 \to T_2; \theta; \delta$ and $r_{21} \simeq r_{22} : T_1; \theta; \delta$, respectively. If $r_{11} = r_{12} = \Uparrow l$ or $r_{21} = r_{22} = \Uparrow l$ for some l, then we are done:

$$\begin{array}{rcl}\theta_1(\delta_1(e_1\ e_2)) & \longrightarrow^* & \Uparrow l \\ \theta_2(\delta_2(e_1\ e_2)) & \longrightarrow^* & \Uparrow l.\end{array}$$

So let $r_{i\,j} = v_{i\,j}$. By definition,

$$v_{11}\ v_{21} \simeq v_{12}\ v_{22} : T_2; \theta; \delta[(v_{21}, v_{22})/x].$$

These normalize to $r_1' \sim r_2' : T_2; \theta; \delta[(v_{21}, v_{22})/x]$. By Lemma 23,

$$r_1' \sim r_2' : [e_2/x]\,T_2; \theta; \delta.$$

By expansion, we can then see

$$\theta_1(\delta_1(e_1\ e_2)) \simeq \theta_2(\delta_2(e_1\ e_2)) : [e_2/x]\,T_2; \theta; \delta.$$

Case (T_TABS): We have $e = \Lambda\alpha.\ e_0$ and $T = \forall\alpha.\,T_0$ and $\Gamma, \alpha \vdash e_0 : T_0$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(\Lambda\alpha.\ e_0)) \sim \theta_2(\delta_2(\Lambda\alpha.\ e_0)) : \forall\alpha.\,T_0; \theta; \delta.$$

Let R, T_1, T_2 be given. We must show that

$$\theta_1(\delta_1(\Lambda\alpha.\ e_0))\ T_1 \simeq \theta_2(\delta_2(\Lambda\alpha.\ e_0))\ T_2 : T_0; \theta[\alpha \mapsto R, T_1, T_2]; \delta.$$

Since

$$\begin{array}{rcl}\theta_1(\delta_1(\Lambda\alpha.\ e_0))\ T_1 & \longrightarrow & [T_1/\alpha]\theta_1(\delta_1(e_0)) \\ \theta_2(\delta_2(\Lambda\alpha.\ e_0))\ T_2 & \longrightarrow & [T_2/\alpha]\theta_2(\delta_2(e_0))\end{array}$$

it suffices to show that

$$[T_1/\alpha]\theta_1(\delta_1(e_0)) \simeq [T_2/\alpha]\theta_2(\delta_2(e_0)) : T_0; \theta[\alpha \mapsto R, T_1, T_2]; \delta.$$

Since $\Gamma, \alpha \vdash \theta[\alpha \mapsto R, T_1, T_2]; \delta$, the IH finishes the case with $\Gamma, \alpha \vdash e_0 \simeq e_0 : T_0$.

Case (T_TAPP): We have $e = e_1\ T_2$ and $\Gamma \vdash e_1 : \forall\alpha.T_0$ and $\Gamma \vdash T_2$ and $T = [T_2/\alpha]T_0$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(e_1\ T_2)) \simeq \theta_2(\delta_2(e_1\ T_2)) : [T_2/\alpha]T_0; \theta; \delta.$$

By the IH,

$$\theta_1(\delta_1(e_1)) \simeq \theta_2(\delta_2(e_1)) : \forall\alpha.T_0; \theta; \delta.$$

These normalize to $r_1 \sim r_2 : \forall\alpha.T_0; \theta; \delta$. If both results are blame, $\theta_1(\delta_1(e_1\ T_2))$ and $\theta_2(\delta_2(e_1\ T_2))$ also normalize to blame, and we are done. So let $r_1 = v_1$ and $r_2 = v_2$. Then, by definition,

$$v_1\ T_1' \simeq v_2\ T_2' : T_0; \theta[\alpha \mapsto R, T_1', T_2']; \delta$$

for any R, T_1', T_2'. In particular,

$$v_1\,\theta_1(\delta_1(T_2)) \simeq v_2\,\theta_2(\delta_2(T_2)) : T_0; \theta[\alpha \mapsto R_{T_2,\theta,\delta}, \theta_1(\delta_1(T_2)), \theta_2(\delta_2(T_2))]; \delta.$$

These normalize to

$$r_1' \sim r_2' : T_0; \theta[\alpha \mapsto R_{T_2,\theta,\delta}, \theta_1(\delta_1(T_2)), \theta_2(\delta_2(T_2))]; \delta.$$

By Lemma 24, $r_1' \sim r_2' : [T_2/\alpha]T_0; \theta; \delta$. By expansion,

$$\theta_1(\delta_1(e_1\ T_2)) \simeq \theta_2(\delta_2(e_1\ T_2)) : [T_2/\alpha]T_0; \theta; \delta.$$

Case (T_CAST): We have $e = \langle T_1 \Rightarrow T_2 \rangle^l_\sigma$ and $\vdash \Gamma$ and $T_1 \parallel T_2$ and $\Gamma \vdash T_1$, $\Gamma \vdash T_2$ and $T = T_1 \to T_2$. By the IH, $\Gamma \vdash T_1 \simeq T_1 : *$ and $\Gamma \vdash T_2 \simeq T_2 : *$. By Lemma 26,

$$\Gamma \vdash \langle T_1 \Rightarrow T_2 \rangle^l_\sigma \simeq \langle T_1 \Rightarrow T_2 \rangle^l_\sigma : \sigma(T_1 \to T_2),$$

which is exactly what we were looking for.

Case (T_BLAME): Immediate.

Case (T_CHECK): We have $e = \langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l$ and $\emptyset \vdash v : T_1$ and $\emptyset \vdash e_2 : \mathsf{bool}$, $\vdash \Gamma$ and $\emptyset \vdash \{x{:}T_1 \mid e_1\}$ and $[v/x]e_1 \longrightarrow^* e_2$ and $T = \{x{:}T_1 \mid e_1\}$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$\theta_1(\delta_1(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) \simeq \theta_2(\delta_2(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) : \{x{:}T_1 \mid e_1\}; \theta; \delta.$$

By the IH,

$$\theta_1(\delta_1(e_2)) \simeq \theta_2(\delta_2(e_2)) : \mathsf{bool}; \theta; \delta$$

and these normalize to the same result. If the result is false or $\Uparrow l'$ for some l', then, for some l'',

$$\begin{aligned} \theta_1(\delta_1(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) &\longrightarrow^* \Uparrow l'' \\ \theta_2(\delta_2(\langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^l)) &\longrightarrow^* \Uparrow l''. \end{aligned}$$

Otherwise, the result is true. Then, by the IH, $v \sim v : T_1; \theta; \delta$ and $\emptyset \vdash \{x{:}T_1 \mid e_1\} \simeq \{x{:}T_1 \mid e_1\} : *$. By definition,

$$[v/x]\theta_1(\delta_1(e_1)) \simeq [v/x]\theta_2(\delta_2(e_1)) : \mathsf{bool}; \theta; \delta[(v, v)/x].$$

Then, we have

$$\begin{array}{rcl} [v/x]\theta_1(\delta_1(e_1)) = [v/x]e_1 & \longrightarrow^* & \mathsf{true} \\ [v/x]\theta_2(\delta_2(e_1)) = [v/x]e_1 & \longrightarrow^* & \mathsf{true}. \end{array}$$

By definition, $v \simeq v : \{x{:}T_1 \mid e_1\}; \theta; \delta$. By expansion,

$$\theta_1(\delta_1(\langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^l)) \simeq \theta_2(\delta_2(\langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^l)) : \{x{:}T_1 \mid e_1\}; \theta; \delta.$$

Case (T_CONV): By Lemma 25.

Case (T_EXACT): We have $e = v$ and $\emptyset \vdash v : T$ and $\emptyset \vdash \{x{:}T_0 \mid e_0\}$ and $[v/x]e_0 \longrightarrow^* \mathsf{true}$ and $T = \{x{:}T_0 \mid e_0\}$. Let $\Gamma \vdash \theta; \delta$. We wish to show that

$$v \sim v : \{x{:}T_0 \mid e_0\}; \theta; \delta.$$

By the IH, $v \sim v : T_0; \theta; \delta$. Since $\emptyset \vdash \{x{:}T_0 \mid e_0\}$, the only free variable in e_0 is x and

$$\begin{array}{rcl} [v/x]\theta_1(\delta_1(e_0)) = [v/x]e_0 & \longrightarrow^* & \mathsf{true} \\ [v/x]\theta_2(\delta_2(e_0)) = [v/x]e_0 & \longrightarrow^* & \mathsf{true}. \end{array}$$

By definition, $v \sim v : \{x{:}T_0 \mid e_0\}; \theta; \delta$.

Case (T_FORGET): By the IH, $\emptyset \vdash v \simeq v : \{x{:}T \mid e\}$, which implies $\Gamma \vdash v \simeq v : T$.

Case (WF_BASE): Trivial.

Case (WF_TVAR): Trivial.

Case (WF_FUN): By the IH.

Case (WF_FORALL): By the IH.

Case (WF_REFINE): By the IH.

□

Appendix C

Proofs of Manifest Contracts with Algebraic Datatypes

This chapter shows properties of $\lambda^{H}_{\mathbf{dt}}$. We start with showing a few properties of type and term equivalence (Section C.1). Similarly to Chapter B, we prove cotermination in Section C.2 and type soundness in Section C.3. Finally, Section C.4 shows correctness of our syntactic type translation given in Section 4.3. In what follows, we use notation R^i, which denotes i-times composition of relation R, and function *unref*, which returns the underlying type of an argument type. The formal definition of *unref* is the same as the one given in Section 3.2.3.

C.1 Term and Type Equivalence

Lemma C.1.1 (Type and Term Equivalences are Equivalences)**.**

(1) The relation $\equiv$ over types is a equivalence relation:

- $T \equiv T$ *for any* T.
- *If* $T_1 \equiv T_2$ *and* $T_2 \equiv T_3$, *then* $T_1 \equiv T_3$.
- *If* $T_1 \equiv T_2$, *then* $T_2 \equiv T_1$.

(2) The relation $\equiv$ over terms is a equivalence relation:

- $e \equiv e$ *for any* e.
- *If* $e_1 \equiv e_2$ *and* $e_2 \equiv e_3$, *then* $e_1 \equiv e_3$.
- *If* $e_1 \equiv e_2$, *then* $e_2 \equiv e_1$.

Proof. Since $\equiv$ is the transitive and symmetric closure of $\Rightarrow\!\!\!\!\!\Rightarrow$, transitivity and symmetry hold obviously.

We show reflexivity of $\equiv$ over types. Let T be a type, and x be a variable such that $x \notin \mathsf{FV}(T)$. Suppose that $e_1 \longrightarrow e_2$ for some e_1 and e_2 (e.g., $e_1 = \lambda\, x{:}\mathsf{bool}.x$ and $e_2 = \mathsf{true}$). Then, we have $T\{e_1/x\} \Rightarrow\!\!\!\!\!\Rightarrow T\{e_2/x\}$. Since $T\{e_1/x\} = T\{e_2/x\} = T$, we finish.

Reflexivity of $\equiv$ over terms can be shown similarly. Let e be a term, and x be a variable such that $x \notin \mathsf{FV}(e)$. Suppose that $e_1 \longrightarrow e_2$ for some e_1 and e_2 (e.g., $e_1 = \lambda\, x{:}\mathsf{bool}.x$ and $e_2 = \mathsf{true}$). Then, we have $e\{e_1/x\} \Rightarrow\!\!\!\!\!\Rightarrow e\{e_2/x\}$. Since $e\{e_1/x\} = e\{e_2/x\} = e$, we finish. □

Lemma C.1.2. *If* $e_1 \longrightarrow e_2$, *then* $e_1 \Rightarrow\!\!\!\!\!\Rightarrow e_2$.

Proof. Obvious because $x\{e_1/x\} \Rightarrow\!\!\!\!\!\Rightarrow x\{e_2/x\}$. □

Lemma C.1.3.

(1) If $e_1 \Rrightarrow e_2$, *then* $T\{e_1/x\} \Rrightarrow T\{e_2/x\}$.

(2) If $e_1 \Rrightarrow^* e_2$, *then* $T\{e_1/x\} \Rrightarrow^* T\{e_2/x\}$.

(3) If $e_1 \equiv e_2$, *then* $T\{e_1/x\} \equiv T\{e_2/x\}$.

Proof.

1. Since $e_1 \Rrightarrow e_2$, there exist e, y, e_1' and e_2' such that $e_1 = e\{e_1'/y\}$ and $e_2 = e\{e_2'/y\}$ and $e_1' \longrightarrow e_2'$. Suppose that z is a fresh variable. Here, we have
   - $T\{e_1/x\} = T\{e\{e_1'/y\}/x\} = T\{e\{z/y\}\{e_1'/z\}/x\} = T\{e\{z/y\}/x\}\{e_1'/z\}$,
   - $T\{e\{z/y\}/x\}\{e_1'/z\} \Rrightarrow T\{e\{z/y\}/x\}\{e_2'/z\}$, and
   - $T\{e\{z/y\}/x\}\{e_2'/z\} = T\{e\{z/y\}\{e_2'/z\}/x\} = T\{e\{e_2'/y\}/x\} = T\{e_2/x\}$.

   Thus, $T\{e_1/x\} \Rrightarrow T\{e_2/x\}$.

2. By mathematical induction on the number of steps of $e_1 \Rrightarrow^* e_2$.

   Case 0: Obvious because $e_1 = e_2$.

   Case $i+1$: We are given $e_1 \Rrightarrow e_3 \Rrightarrow^i e_2$ for some e_3. By the IH and the first case, we finish.

3. By induction on $e_1 \equiv e_2$.

   Case $e_1 \Rrightarrow e_2$: By the first case.

   Case transitivity and symmetry: By the IH(s). □

Lemma C.1.4.

(1) If $T_1 \Rrightarrow T_2$, *then* $T_1\{e/x\} \Rrightarrow T_2\{e/x\}$

(2) If $T_1 \Rrightarrow^* T_2$, *then* $T_1\{e/x\} \Rrightarrow^* T_2\{e/x\}$

(3) If $T_1 \equiv T_2$, *then* $T_1\{e/x\} \equiv T_2\{e/x\}$.

Proof.

1. By definition, there exist T, y, e_1 and e_2 such that $T_1 = T\{e_1/y\}$ and $T_2 = T\{e_2/y\}$ and $e_1 \longrightarrow e_2$. Suppose that z is a fresh variable. Since the evaluation relation is defined over closed terms, it is found that e_1 and e_2 are closed. Here, we have
   - $T_1\{e/x\} = T\{e_1/y\}\{e/x\} = T\{z/y\}\{e_1/z\}\{e/x\} = T\{z/y\}\{e/x\}\{e_1/z\}$,
   - $T\{z/y\}\{e/x\}\{e_1/z\} \Rrightarrow T\{z/y\}\{e/x\}\{e_2/z\}$, and
   - $T\{z/y\}\{e/x\}\{e_2/z\} = T\{z/y\}\{e_2/z\}\{e/x\} = T\{e_2/y\}\{e/x\} = T_2\{e/x\}$.

   Thus, $T_1\{e/x\} \Rrightarrow T_2\{e/x\}$.

2. By mathematical induction on the number of steps of $T_1 \Rrightarrow^* T_2$.

   Case 0: Obvious because $T_1 = T_2$.

   Case $i+1$: We are given $T_1 \Rrightarrow T_3 \Rrightarrow^i T_2$ for some T_3. By the IH and the first case, we finish.

3. By induction on $T_1 \equiv T_2$.

   Case $T_1 \Rrightarrow T_2$: By the first case.

   Case transitivity and symmetry: Obvious by the IH(s). □

Lemma C.1.5.

(1) If $e_1 \Rrightarrow e_2$, *then* $e\{e_1/x\} \Rrightarrow e\{e_2/x\}$.

(2) If $e_1 \Rrightarrow^* e_2$, *then* $e\{e_1/x\} \Rrightarrow^* e\{e_2/x\}$.

(3) If $e_1 \equiv e_2$, *then* $e\{e_1/x\} \equiv e\{e_2/x\}$

Proof.

1. Since $e_1 \Rrightarrow e_2$, there exists some e', y, e_1' and e_2' such that $e_1 = e'\{e_1'/y\}$ and $e_2 = e'\{e_2'/y\}$ and $e_1' \longrightarrow e_2'$. Suppose that z is a fresh variable. Here, we have

   - $e\{e_1/x\} = e\{e'\{e_1'/y\}/x\} = e\{e'\{z/y\}\{e_1'/z\}/x\} = e\{e'\{z/y\}/x\}\{e_1'/z\}$,
   - $e\{e'\{z/y\}/x\}\{e_1'/z\} \Rrightarrow e\{e'\{z/y\}/x\}\{e_2'/z\}$, and
   - $e\{e'\{z/y\}/x\}\{e_2'/z\} = e\{e'\{z/y\}\{e_2'/z\}/x\} = e\{e'\{e_2'/y\}/x\} = e\{e_2/x\}$.

   Thus, $e\{e_1/x\} \Rrightarrow e\{e_2/x\}$.

2. By mathematical induction on the number of steps of $e_1 \Rrightarrow^* e_2$.

   Case 0: Obvious because $e_1 = e_2$.

   Case $i+1$: We are given $e_1 \Rrightarrow e_3 \Rrightarrow^i e_2$ for some e_3. By the IH and the first case, we finish.

3. By induction on $e_1 \equiv e_2$.

   Case $e_1 \Rrightarrow e_2$: By the first case.

   Case transitivity and symmetry: By the IH(s). □

C.2 Cotermination

The proof of cotermination in $\lambda^H_{\mathbf{dt}}$ is similar to the one in $\mathrm{F}^\sigma_{\mathrm{H}}$: we show that the relation $\{(e\{e_1/x\}, e\{e_2/x\}) \mid e_1 \longrightarrow e_2\}$ is weak bisimulation (Lemmas C.2.18 and C.2.21), using auxiliary lemmas, and then cotermination (Lemma 29).

Lemma C.2.1 (Determinism). *If* $e \longrightarrow e_1$ *and* $e \longrightarrow e_2$, *then* $e_1 = e_2$.

Proof. Straightforward. □

Lemma C.2.2 (Value Construction Closed Substitution). *For any* v, x, *and* e, $v\{e/x\}$ *is a value.*

Proof. By structural induction on v.

Case $v = c$, $\texttt{fix}\, f(x{:}T) = e$ or $\langle T_1 \Leftarrow T_2 \rangle^\ell$: Obvious.

Case $v = (v_1, v_2)$ or $C\langle e' \rangle v'$: By the IHs. □

Lemma C.2.3. *If e_1 is not a value and $e_2\{e_1/x\}$ is, then e_2 is a value.*

Proof. By structural induction on e_2.

Case $e_2 = y$: If $x = y$, then $e_2\{e_1/x\} = e_1$, which leads to a contradiction from the assumptions that e_1 is not a value and $e_2\{e_1/x\}$ is. Otherwise, if $x \neq y$, then there is a contradiction because $e_2\{e_1/x\}$ is a value but $e_2\{e_1/x\} = y$ is not.

Case $e_2 = v$: By Lemma C.2.2.

Case $e_2 = e_2'\ e_2'$, $e.\,i$, match e_0' with $\overline{C_i\ y_i \to e_i'}^{\,i}$, if e_1' then e_2' else e_3', $\Uparrow \ell$, $\langle \{y{:}T \mid e_1'\}, e_2', v' \rangle^\ell$, or $\langle\!\langle \{y{:}T \mid e_1'\}, e_2' \rangle\!\rangle^\ell$: Contradictory.

Case $e = (e_1, e_2)$ or $C\langle e_1 \rangle v_2$: By the IH(s).

□

Lemma C.2.4. *Let e_1 and e_2 are closed terms such that $e_1 \equiv e_2$. If $(v_1\ v_2)\{e_1/x\} \longrightarrow e$, then $(v_1\ v_2)\{e_2/x\} \longrightarrow e'\{e_2/x\}$ for some e' such that $e = e'\{e_1/x\}$.*

Proof. By Lemma C.2.2, $v_1\{e_1/x\}$, $v_1\{e_2/x\}$, $v_2\{e_1/x\}$ and $v_2\{e_2/x\}$ are values. We proceed by case analysis on v_1. Note that v_1 takes the form of either lambda abstraction or cast since $(v_1\ v_2)\{e_1/x\}$ takes a step and that if $(v_1\ v_2)\{e_1/x\}$ is closed, then so is $(v_1\ v_2)\{e_2/x\}$. In the following, let $i \in \{1, 2\}$.

Case $v_1 = \texttt{fix}\, f(y{:}T) = e'$: Without loss of generality, we can suppose that y and f are fresh. By (E_RED)/(R_BETA),

$$((\texttt{fix}\, f(y{:}T) = e')\ v_2)\{e_i/x\} \longrightarrow e'\{e_i/x\}\{v_2\{e_i/x\}/y, v_1\{e_i/x\}/f\}.$$

Because $e'\{e_i/x\}\{v_2\{e_i/x\}/y, v_1\{e_i/x\}/f\} = e'\{v_2/y, v_1/f\}\{e_i/x\}$, we finish.

Case $v_1 = \langle \mathsf{bool} \Leftarrow \mathsf{bool} \rangle^\ell$: Obvious because $(\langle \mathsf{bool} \Leftarrow \mathsf{bool} \rangle^\ell\ v_2)\{e_i/x\} \longrightarrow v_2\{e_i/x\}$ by (E_RED)/(R_BASE).

Case $v_1 = \langle y{:}T_{11} \to T_{12} \Leftarrow y{:}T_{21} \to T_{22} \rangle^\ell$: Without loss of generality, we can suppose that y is fresh. By (E_RED)/(R_FUN),

$$\begin{array}{rl} & (\langle y{:}T_{11} \to T_{12} \Leftarrow y{:}T_{21} \to T_{22} \rangle^\ell\ v_2)\{e_i/x\} \\ \longrightarrow & \lambda\, y{:}T_{11}\{e_i/x\}. \\ & \quad (\lambda\, z{:}T_{21}\{e_i/x\}.\langle T_{12}\{e_i/x\} \Leftarrow T_{22}\{e_i/x\}\{z/y\} \rangle^\ell\ (v_2\{e_i/x\}\ z)) \\ & \quad (\langle T_{21}\{e_i/x\} \Leftarrow T_{11}\{e_i/x\} \rangle^\ell\ y) \\ = & (\lambda\, y{:}T_{11}.(\lambda\, z{:}T_{21}.\langle T_{12} \Leftarrow T_{22}\{z/y\} \rangle^\ell\ (v_2\ z))\ (\langle T_{21} \Leftarrow T_{11} \rangle^\ell\ y))\{e_i/x\} \end{array}$$

for some fresh variable z. Thus, we finish.

Case $v_1 = \langle y{:}T_{11} \times T_{12} \Leftarrow y{:}T_{21} \times T_{22} \rangle^\ell$: Without loss of generality, we can suppose that y is fresh. It is found that $v_2 = (v_1', v_2')$ for some v_1' and v_2' because (1)

$(\langle y{:}T_{11} \times T_{12} \Leftarrow y{:}T_{21} \times T_{22}\rangle^{\ell}\, v_2)\,\{e_1/x\}$ takes a step, (2) the only rule applicable to the application term is (E_RED)/(R_PROD), and (3) v_2 is a value (thus not a variable). By (E_RED)/(R_PROD),

$$\begin{array}{rl} & (\langle y{:}T_{11} \times T_{12} \Leftarrow y{:}T_{21} \times T_{22}\rangle^{\ell}\,(v_1', v_2'))\,\{e_i/x\} \\ \longrightarrow & (\lambda\, y{:}T_{11}\,\{e_i/x\}.(y, \langle T_{12}\,\{e_i/x\} \Leftarrow T_{22}\,\{e_i/x\}\,\{v_1'\,\{e_i/x\}/y\}\rangle^{\ell}\, v_2'\,\{e_i/x\})) \\ & (\langle T_{11}\,\{e_i/x\} \Leftarrow T_{21}\,\{e_i/x\}\rangle^{\ell}\, v_1'\,\{e_i/x\}) \\ = & ((\lambda\, y{:}T_{11}.(y, \langle T_{12} \Leftarrow T_{22}\,\{v_1'/y\}\rangle^{\ell}\, v_2'))\,(\langle T_{11} \Leftarrow T_{21}\rangle^{\ell}\, v_1'))\,\{e_i/x\}. \end{array}$$

Case $v_1 = \langle T_1 \Leftarrow \{y{:}T_2 \mid e\}\rangle^{\ell}$: By (E_RED)/(R_FORGET),

$$\begin{array}{rcl} (\langle T_1 \Leftarrow \{y{:}T_2 \mid e\}\rangle^{\ell}\, v_2)\,\{e_i/x\} & \longrightarrow & \langle T_1\,\{e_i/x\} \Leftarrow T_2\,\{e_i/x\}\rangle^{\ell}\, v_2\,\{e_i/x\} \\ & = & (\langle T_1 \Leftarrow T_2\rangle^{\ell}\, v_2)\,\{e_i/x\}. \end{array}$$

Case $v_1 = \langle \{y{:}T_1 \mid e\} \Leftarrow T_2\rangle^{\ell}$ where T_2 is not a refinement type: By (E_RED)/(R_PRECHECK),

$$\begin{array}{rl} & (\langle \{y{:}T_1 \mid e\} \Leftarrow T_2\rangle^{\ell}\, v_2)\,\{e_i/x\} \\ \longrightarrow & \langle\!\langle \{y{:}T_1 \mid e\}\,\{e_i/x\}, \langle T_1\,\{e_i/x\} \Leftarrow T_2\,\{e_i/x\}\rangle^{\ell}\, v_2\,\{e_i/x\}\rangle\!\rangle^{\ell} \\ = & \langle\!\langle \{y{:}T_1 \mid e\}, \langle T_1 \Leftarrow T_2\rangle^{\ell}\, v_2\rangle\!\rangle^{\ell}\,\{e_i/x\}. \end{array}$$

Case $v_1 = \langle \tau_1\langle e_1''\rangle \Leftarrow \tau_2\langle e_2''\rangle\rangle^{\ell}$: There are three reduction rules by which $(v_1\, v_2)\,\{e_1/x\}$ takes a step.

Case (E_RED)/(R_DATATYPE): We find that $v_2 = C_2\langle e''\rangle v''$ for some C_2, e'' and v'' since v_2 is a value (thus not a variable). We are given

$$\begin{array}{rl} & (\langle \tau_1\langle e_1''\rangle \Leftarrow \tau_2\langle e_2''\rangle\rangle^{\ell}\, C_2\langle e''\rangle v'')\,\{e_1/x\} \\ \longrightarrow & C_1\langle e_1''\,\{e_1/x\}\rangle(\langle T_1'\,\{e_1''\,\{e_1/x\}/y_1\} \Leftarrow T_2'\,\{e_2''\,\{e_1/x\}/y_2\}\rangle^{\ell}\, v''\,\{e_1/x\}) \\ = & (C_1\langle e_1''\rangle(\langle T_1'\,\{e_1''/y_1\} \Leftarrow T_2'\,\{e_2''/y_2\}\rangle^{\ell}\, v''))\,\{e_1/x\} \end{array}$$

where $\delta((\langle \tau_1\langle e_1''\rangle \Leftarrow \tau_2\langle e_2''\rangle\rangle^{\ell}\, C_2\langle e''\rangle v'')\,\{e_1/x\}) = C_1$ and, for $j \in \{1,2\}$, $\mathit{ArgTypeOf}(\tau_j) = y_j{:}T_j$ and $\mathit{CtrArgOf}(C_j) = T_j'$. Note that only y_1 and y_2 can occur free in T_1' and T_2', respectively, because of well-formedness of the type definition environment. Since $e_1 \equiv e_2$, we have $(v_1\, v_2)\,\{e_1/x\} \equiv (v_1\, v_2)\,\{e_2/x\}$ by Lemma C.1.5 (3). From well-formedness of the constructor choice function, we have $\delta((v_1\, v_2)\,\{e_2/x\}) = \delta((v_1\, v_2)\,\{e_1/x\}) = C_1$. Thus, by (E_RED)/(R_DATATYPE),

$$\begin{array}{rl} & (\langle \tau_1\langle e_1''\rangle \Leftarrow \tau_2\langle e_2''\rangle\rangle^{\ell}\, C_2\langle e''\rangle v'')\,\{e_2/x\} \\ \longrightarrow & C_1\langle e_1''\,\{e_2/x\}\rangle(\langle T_1'\,\{e_1''\,\{e_2/x\}/y_1\} \Leftarrow T_2'\,\{e_2''\,\{e_2/x\}/y_2\}\rangle^{\ell}\, v''\,\{e_2/x\}) \\ = & (C_1\langle e_1''\rangle(\langle T_1'\,\{e_1''/y_1\} \Leftarrow T_2'\,\{e_2''/y_2\}\rangle^{\ell}\, v''))\,\{e_2/x\}. \end{array}$$

Case (E_RED)/(R_DATATYPEMONO): By (E_RED)/(R_DATATYPEMONO), $(\langle \tau_1 \Leftarrow \tau_2\rangle^{\ell}\, v_2)\,\{e_i/x\} \longrightarrow v_2\,\{e_i/x\}$.

Case (E_RED)/(R_DATATYPEFAIL): We are given $(\langle \tau_1\langle e_1''\rangle \Leftarrow \tau_2\langle e_2''\rangle\rangle^{\ell}\, v_2)\,\{e_1/x\} \longrightarrow \Uparrow\ell$ and $\delta((\langle \tau_1\langle e_1''\rangle \Leftarrow \tau_2\langle e_2''\rangle\rangle^{\ell}\, v_2)\,\{e_1/x\})$ is undefined. Since $e_1 \equiv e_2$, we have $(v_1\, v_2)\,\{e_1/x\} \equiv (v_1\, v_2)\,\{e_2/x\}$ by Lemma C.1.5 (3). If $\delta((v_1\, v_2)\,\{e_2/x\})$ is defined, then so is

$\delta((v_1\ v_2)\,\{e_1/x\})$ from well-formedness of the constructor choice function but it contradicts. Thus, $\delta((v_1\ v_2)\,\{e_2/x\})$ is also undefined and so, by (E_RED)/(R_DATATYPEFAIL), $(\langle\tau_1\langle e_1''\rangle \Leftarrow \tau_2\langle e_2''\rangle\rangle^{\ell}\ v_2)\,\{e_2/x\} \longrightarrow \Uparrow\ell$. □

Lemma C.2.5. *Let e_1 and e_2 be terms such that $e_1 \longrightarrow e_2$.*

(1) If $(v_1\ v_2)\,\{e_1/x\} \longrightarrow e$, then $(v_1\ v_2)\,\{e_2/x\} \longrightarrow e'\,\{e_2/x\}$ for some e' such that $e = e'\,\{e_1/x\}$.

(2) If $(v_1\ v_2)\,\{e_2/x\} \longrightarrow e$, then $(v_1\ v_2)\,\{e_1/x\} \longrightarrow e'\,\{e_1/x\}$ for some e' such that $e = e'\,\{e_2/x\}$.

Proof. Since the evaluation relation is defined over closed terms, e_1 and e_2 are closed. Thus, we finish by Lemma C.2.4. □

Lemma C.2.6. *Let e_1 and e_2 are closed terms, and $i \in \{1, 2\}$. If $(v.\,i)\,\{e_1/x\} \longrightarrow e$, then $(v.\,i)\,\{e_2/x\} \longrightarrow e'\,\{e_2/x\}$ for some e' such that $e = e'\,\{e_1/x\}$.*

Proof. By Lemma C.2.2, $v\,\{e_1/x\}$ and $v\,\{e_2/x\}$ are values. We find that v takes the form of pair since $(v.\,i)\,\{e_1/x\}$ takes a step. Note that if $(v.\,i)\,\{e_1/x\}$ is closed, then so is $(v.\,i)\,\{e_2/x\}$.

We are given $v = (v_1, v_2)$ for some v_1 and v_2. By (E_RED)/(R_PROJi), for $j \in \{1, 2\}$,

$$((v_1, v_2).\,i)\,\{e_j/x\} \longrightarrow v_i\,\{e_j/x\}.$$

Thus, we finish. □

Lemma C.2.7. *Let e_1 and e_2 be terms such that $e_1 \longrightarrow e_2$, and $i \in \{1, 2\}$.*

(1) If $(v.\,i)\,\{e_1/x\} \longrightarrow e$, then $(v.\,i)\,\{e_2/x\} \longrightarrow e'\,\{e_2/x\}$ for some e' such that $e = e'\,\{e_1/x\}$.

(2) If $(v.\,i)\,\{e_2/x\} \longrightarrow e$, then $(v.\,i)\,\{e_1/x\} \longrightarrow e'\,\{e_1/x\}$ for some e' such that $e = e'\,\{e_2/x\}$.

Proof. Since the evaluation relation is defined over closed terms, e_1 and e_2 are closed. Thus, we finish by Lemma C.2.6. □

Lemma C.2.8. *Let e_1 and e_2 are closed terms. If $(\mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_1/x\} \longrightarrow e$, then $(\mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_2/x\} \longrightarrow e'\,\{e_2/x\}$ for some e' such that $e = e'\,\{e_1/x\}$.*

Proof. By Lemma C.2.2, $v\,\{e_1/x\}$ and $v\,\{e_2/x\}$ are values. Note that v takes the form of Boolean value since $(\mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_1/x\}$ takes a step and that if $(\mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_1/x\}$ is closed, then so is $(\mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_2/x\}$. By case analysis on v. In the following, let $i \in \{1, 2\}$.

Case $v = \mathsf{true}$: By (E_RED)/(R_IFTRUE),

$$(\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_i/x\} \longrightarrow e_1'\,\{e_i/x\}.$$

Case $v = \mathsf{false}$: By (E_RED)/(R_IFFALSE),

$$(\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_i/x\} \longrightarrow e_2'\,\{e_i/x\}.$$

□

Lemma C.2.9. *Let e_1 and e_2 be terms such that $e_1 \longrightarrow e_2$.*

(1) If $(\mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_1/x\} \longrightarrow e$, *then* $(\mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_2/x\} \longrightarrow e'\,\{e_2/x\}$ *for some e' such that $e = e'\,\{e_1/x\}$.*

(2) If $(\mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_2/x\} \longrightarrow e$, *then* $(\mathsf{if}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_2')\,\{e_1/x\} \longrightarrow e'\,\{e_1/x\}$ *for some e' such that $e = e'\,\{e_2/x\}$.*

Proof. Since the evaluation relation is defined over closed terms, e_1 and e_2 are closed. Thus, we finish by Lemma C.2.8. □

Lemma C.2.10. *Let e_1 and e_2 are closed terms. If* $(\mathsf{match}\ v\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_1/x\} \longrightarrow e$, *then* $(\mathsf{match}\ v\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_2/x\} \longrightarrow e'\,\{e_2/x\}$ *for some e' such that $e = e'\,\{e_1/x\}$.*

Proof. Without loss of generality, we can suppose that each y_i is fresh. By Lemma C.2.2, $v\,\{e_1/x\}$ and $v\,\{e_2/x\}$ are values. We find that v takes the form of constructor application since $(\mathsf{match}\ v\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_1/x\}$ takes a step. Note that if $(\mathsf{match}\ v\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_1/x\}$ is closed, then so is $(\mathsf{match}\ v\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_2/x\}$.

We are given $v = C_j\langle e'\rangle v'$ for some $C_j \in \overline{C_i}^{\,i}$, e' and v'. By (E_RED)/(R_MATCH), for $k \in \{1, 2\}$,

$$\begin{array}{rcl}(\mathsf{match}\ C_j\langle e'\rangle v'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_k/x\} & \longrightarrow & e_j'\,\{e_k/x\}\,\{v'\,\{e_k/x\}/y_j\} \\ & = & e_j'\,\{v'/y_j\}\,\{e_k/x\}.\end{array}$$

Thus, we finish. □

Lemma C.2.11. *Let e_1 and e_2 be terms such that $e_1 \longrightarrow e_2$.*

(1) If $(\mathsf{match}\ v\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_1/x\} \longrightarrow e$, *then* $(\mathsf{match}\ v\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_2/x\} \longrightarrow e'\,\{e_2/x\}$ *for some e' such that $e = e'\,\{e_1/x\}$.*

(2) If $(\mathsf{match}\ v\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_2/x\} \longrightarrow e$, *then* $(\mathsf{match}\ v\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\,\{e_1/x\} \longrightarrow e'\,\{e_1/x\}$ *for some e' such that $e = e'\,\{e_2/x\}$.*

Proof. Since the evaluation relation is defined over closed terms, e_1 and e_2 are closed. Thus, we finish by Lemma C.2.10. □

Lemma C.2.12. *Let e_1 and e_2 are closed terms. If $\langle\!\langle \{y{:}T \mid e_1'\}, v\rangle\!\rangle^{\ell}\,\{e_1/x\} \longrightarrow e$, then $\langle\!\langle \{y{:}T \mid e_1'\}, v\rangle\!\rangle^{\ell}\,\{e_2/x\} \longrightarrow e'\,\{e_2/x\}$ for some e' such that $e = e'\,\{e_1/x\}$.*

Proof. Without loss of generality, we can suppose that y is fresh. By Lemma C.2.2, $v\,\{e_1/x\}$ and $v\,\{e_2/x\}$ are values. Note that if $\langle\!\langle \{y{:}T \mid e_1'\}, v\rangle\!\rangle^{\ell}\,\{e_1/x\}$ is closed, then so is $\langle\!\langle \{y{:}T \mid e_1'\}, v\rangle\!\rangle^{\ell}\,\{e_2/x\}$. Letting $i \in \{1, 2\}$, by (E_RED)/(R_CHECK),

$$\begin{array}{rcl}\langle\!\langle \{y{:}T \mid e_1'\}, v\rangle\!\rangle^{\ell}\,\{e_i/x\} & \longrightarrow & \langle \{y{:}T \mid e_1'\}\,\{e_i/x\}, e_1'\,\{e_i/x\}\,\{v\,\{e_i/x\}/y\}, v\,\{e_i/x\}\rangle^{\ell} \\ & = & \langle \{y{:}T \mid e_1'\}, e_1'\,\{v/y\}, v\rangle^{\ell}\,\{e_i/x\}.\end{array}$$

Thus, we finish. □

Lemma C.2.13. *Let e_1 and e_2 be terms such that $e_1 \longrightarrow e_2$.*

(1) If $\langle\!\langle \{y{:}T \mid e_1'\}, v\rangle\!\rangle^{\ell}\,\{e_1/x\} \longrightarrow e$, then $\langle\!\langle \{y{:}T \mid e_1'\}, v\rangle\!\rangle^{\ell}\,\{e_2/x\} \longrightarrow e'\,\{e_2/x\}$ for some e' such that $e = e'\,\{e_1/x\}$.

(2) If $\langle\!\langle \{y{:}T \mid e_1'\}, v \rangle\!\rangle^{\ell} \{e_2/x\} \longrightarrow e$, *then* $\langle\!\langle \{y{:}T \mid e_1'\}, v \rangle\!\rangle^{\ell} \{e_1/x\} \longrightarrow e' \{e_1/x\}$ *for some* e' *such that* $e = e' \{e_2/x\}$.

Proof. Since the evaluation relation is defined over closed terms, e_1 and e_2 are closed. Thus, we finish by Lemma C.2.12. □

Lemma C.2.14. *Let* e_1 *and* e_2 *are closed terms. If* $\langle \{y{:}T \mid e_1'\}, v_1, v_2 \rangle^{\ell} \{e_1/x\} \longrightarrow e$, *then* $\langle \{y{:}T \mid e_1'\}, v_1, v_2 \rangle^{\ell} \{e_2/x\} \longrightarrow e' \{e_2/x\}$ *for some* e' *such that* $e = e' \{e_1/x\}$.

Proof. By Lemma C.2.2, $v_1 \{e_1/x\}$ and $v_1 \{e_2/x\}$ are values. Note that v_1 takes the form of Boolean value since $\langle \{y{:}T \mid e_1'\}, v_1, v_2 \rangle^{\ell} \{e_1/x\}$ takes a step and that if $\langle \{y{:}T \mid e_1'\}, v_1, v_2 \rangle^{\ell} \{e_1/x\}$ is closed, then so is $\langle \{y{:}T \mid e_1'\}, v_1, v_2 \rangle^{\ell} \{e_2/x\}$. By case analysis on v_1. In the following, let $i \in \{1, 2\}$.

Case $v_1 = \mathsf{true}$: By (E_RED)/(R_OK), $\langle \{y{:}T \mid e_1'\}, \mathsf{true}, v_2 \rangle^{\ell} \{e_i/x\} \longrightarrow v_2 \{e_i/x\}$.

Case $v_2 = \mathsf{false}$: By (E_RED)/(R_FAIL), $\langle \{y{:}T \mid e_1'\}, \mathsf{false}, v_2 \rangle^{\ell} \{e_i/x\} \longrightarrow \Uparrow \ell$. □

Lemma C.2.15. *Let* e_1 *and* e_2 *be terms such that* $e_1 \longrightarrow e_2$.

(1) If $\langle \{y{:}T \mid e_1'\}, v_1, v_2 \rangle^{\ell} \{e_1/x\} \longrightarrow e$, *then* $\langle \{y{:}T \mid e_1'\}, v_1, v_2 \rangle^{\ell} \{e_2/x\} \longrightarrow e' \{e_2/x\}$ *for some* e' *such that* $e = e' \{e_1/x\}$.

(2) If $\langle \{y{:}T \mid e_1'\}, v_1, v_2 \rangle^{\ell} \{e_2/x\} \longrightarrow e$, *then* $\langle \{y{:}T \mid e_1'\}, v_1, v_2 \rangle^{\ell} \{e_1/x\} \longrightarrow e' \{e_1/x\}$ *for some* e' *such that* $e = e' \{e_2/x\}$.

Proof. Since the evaluation relation is defined over closed terms, e_1 and e_2 are closed. Thus, we finish by Lemma C.2.14. □

Lemma C.2.16.

(1) If $e_1 \longrightarrow^n e_2$ *is derived by* (E_RED), *then* $E[e_1] \longrightarrow^n E[e_2]$ *is derived by applying only* (E_RED).

(2) If $e \longrightarrow^* \Uparrow \ell$, *then* $E[e] \longrightarrow^* \Uparrow \ell$.

Proof.

1. By induction on the number of evaluation steps of $e_1 \longrightarrow^n e_2$.

   Case 0: Obvious.

   Case $i + 1$: We are given $e_1 \longrightarrow e_3 \longrightarrow^i e_2$ for some e_3. Since $e_1 \longrightarrow e_3$ is derived by (E_RED), there exist some E', e_1' and e_3' such that $e_1' \rightsquigarrow e_3'$. Since $E[E'[e_1']] \longrightarrow E[E'[e_3']]$ by (E_RED), we finish by the IH.

2. By induction on the number of evaluation steps of $e_1 \longrightarrow^* \Uparrow \ell$.

   Case 0: Since $e = \Uparrow \ell$, we finish by (E_BLAME) if $E \neq [\,]$.

   Case $n + 1$: We are given $e \longrightarrow e' \longrightarrow^n \Uparrow \ell$ for some e'. If the evaluation rule applied to e is (E_RED), then $e = E'[e_1]$ and $e' = E'[e_2]$ for some E', e_1 and e_2 such that $e_1 \rightsquigarrow e_2$. Since $E[E'[e_1]] \longrightarrow E[E'[e_2]]$ by (E_RED), we finish by the IH. Otherwise, if the evaluation rule applied to e is (E_BLAME), then $e = E'[\Uparrow \ell]$ for some E', and $e' = \Uparrow \ell$. By (E_BLAME), $E[E'[\Uparrow \ell]] \longrightarrow \Uparrow \ell$.

□

Lemma C.2.17. *Suppose that* $e_1 \longrightarrow e_2$. *If* $e\,\{e_1/x\} = E_1[\Uparrow\ell]$, *then there exists some* E_2 *such that* $e\,\{e_2/x\} = E_2[\Uparrow\ell]$.

Proof. By structural induction on e

Case $e = x$: It is found that $e_1 = e\,\{e_1/x\} = E_1[\Uparrow\ell]$. Since $E_1[\Uparrow\ell] \longrightarrow \Uparrow\ell$ by (E_BLAME), $e_2 = \Uparrow\ell$.

Case $e = v$: Contradictory.

Case $e = \Uparrow\ell'$: If $\ell' = \ell$, then obvious. Otherwise, if $\ell' \neq \ell$, then contradictory since $e\,\{e_1/x\} = E_1[\Uparrow\ell]$.

Case $e = e_1'\ e_2'$: Since $e\,\{e_1/x\} = E_1[\Uparrow\ell]$, there are two cases we have to consider.

- Case $E_1 = E_1'\ e_2'\,\{e_1/x\}$: Since $e_1'\,\{e_1/x\} = E_1'[\Uparrow\ell]$, there exists some E_2' such that $e_1'\,\{e_2/x\} = E_2'[\Uparrow\ell]$, by the IH. Since $E_2'\ e_2'\,\{e_2/x\}$ is an evaluation context and $e\,\{e_2/x\} = E_2'[\Uparrow\ell]\ e_2'\,\{e_2/x\}$, we finish.
- Case $E_1 = e_1'\,\{e_1/x\}\ E_1'$ where $e_1'\,\{e_1/x\}$ is a value: Since $e_2'\,\{e_1/x\} = E_1'[\Uparrow\ell]$, there exists some E_2' such that $e_2'\,\{e_2/x\} = E_2'[\Uparrow\ell]$, by the IH. Since $e_1'\,\{e_1/x\}$ is a value and e_1 is not a value from $e_1 \longrightarrow e_2$, it is found by Lemmas C.2.3 and C.2.2 that $e_1'\,\{e_2/x\}$ is a value. Thus, since $e_1'\,\{e_2/x\}\ E_2'$ is an evaluation context and $e\,\{e_2/x\} = e_1'\,\{e_2/x\}\ E_2'[\Uparrow\ell]$, we finish.

Case $e = (e_1', e_2')$ which is a not value: Since $e\,\{e_1/x\} = E_1[\Uparrow\ell]$, there are two cases we have to consider.

- Case $E_1 = (E_1', e_2'\,\{e_1/x\})$: Since $e_1'\,\{e_1/x\} = E_1'[\Uparrow\ell]$, there exists some E_2' such that $e_1'\,\{e_2/x\} = E_2'[\Uparrow\ell]$, by the IH. Since $(E_2', e_2'\,\{e_2/x\})$ is an evaluation context and $e\,\{e_2/x\} = (E_2'[\Uparrow\ell], e_2'\,\{e_2/x\})$, we finish.
- Case $E_1 = (e_1'\,\{e_1/x\}, E_1')$ where $e_1'\,\{e_1/x\}$ is a value: Since $e_2'\,\{e_1/x\} = E_1'[\Uparrow\ell]$, there exists some E_2' such that $e_2'\,\{e_2/x\} = E_2'[\Uparrow\ell]$, by the IH. Since $e_1'\,\{e_1/x\}$ is a value, it is found by Lemmas C.2.3 and C.2.2 that $e_1'\,\{e_2/x\}$ is a value. Thus, since $(e_1'\,\{e_2/x\}, E_2')$ is an evaluation context and $e\,\{e_2/x\} = e_1'\,\{e_2/x\}\ E_2'[\Uparrow\ell]$, we finish.

Case $e = e'.\,i$ $(i \in \{1, 2\})$: Since $e\,\{e_1/x\} = E_1[\Uparrow\ell]$, there exists some E_1' such that $e'\,\{e_1/x\} = E_1'[\Uparrow\ell]$. By the IH, there exists some E_2' such that $e'\,\{e_2/x\} = E_2'[\Uparrow\ell]$. Since $e\,\{e_2/x\} = E_2'[\Uparrow\ell].\,i$, we finish.

Case $e = C\langle e_1'\rangle e_2'$ which is a not value: Since $e\,\{e_1/x\} = E_1[\Uparrow\ell]$, there exists some E_1' such that $e_2'\,\{e_1/x\} = E_1'[\Uparrow\ell]$. By the IH, there exists some E_2' such that $e_2'\,\{e_2/x\} = E_2'[\Uparrow\ell]$. Since $C\langle e_1'\,\{e_2/x\}\rangle E_2'$ is an evaluation context and $e\,\{e_2/x\} = C\langle e_1'\,\{e_2/x\}\rangle E_2'[\Uparrow\ell]$, we finish.

Case $e = \mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}$: Since $e\,\{e_1/x\} = E_1[\Uparrow\ell]$, there exists some E_1' such that $e_0'\,\{e_1/x\} = E_1'[\Uparrow\ell]$. By the IH, there exists some E_2' such that $e_0'\,\{e_2/x\} = E_2'[\Uparrow\ell]$. Since $\mathsf{match}\ E_2'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}$ is an evaluation context and $e\,\{e_2/x\} = \mathsf{match}\ E_2'[\Uparrow\ell]\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}$, we finish.

Case $e = \mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'$: Since $e\,\{e_1/x\} = E_1[\Uparrow\ell]$, there exists some E_1' such that $e_1'\,\{e_1/x\} = E_1'[\Uparrow\ell]$. By the IH, there exists some E_2' such that $e_1'\,\{e_2/x\} = E_2'[\Uparrow\ell]$. Since $\mathsf{if}\ E_2'\ \mathsf{then}\ e_2'\,\{e_2/x\}\ \mathsf{else}\ e_3'\,\{e_2/x\}$ is an evaluation context and $e\,\{e_2/x\} = \mathsf{if}\ E_2'[\Uparrow\ell]\ \mathsf{then}\ e_2'\,\{e_2/x\}\ \mathsf{else}\ e_3'\,\{e_2/x\}$, we finish.

Case $e = \langle\!\langle \{y{:}T \mid e_1'\}, e_2' \rangle\!\rangle^\ell$: Since $e\,\{e_1/x\} = E_1[\Uparrow\ell]$, there exists some E_1' such that $e_2'\,\{e_1/x\} = E_1'[\Uparrow\ell]$. By the IH, there exists some E_2' such that $e_2'\,\{e_2/x\} = E_2'[\Uparrow\ell]$. Since $\langle\!\langle \{y{:}T \mid e_1'\}\,\{e_2/x\}, E_2' \rangle\!\rangle^\ell$ is an evaluation context and $e\,\{e_2/x\} = \langle\!\langle \{y{:}T \mid e_1'\}\,\{e_2/x\}, E_2'[\Uparrow\ell] \rangle\!\rangle^\ell$, we finish.

Case $e = \langle \{y{:}T \mid e_1'\}, e_2', v' \rangle^\ell$: Since $e\,\{e_1/x\} = E_1[\Uparrow\ell]$, there exists some E_1' such that $e_2'\,\{e_1/x\} = E_1'[\Uparrow\ell]$. By the IH, there exists some E_2' such that $e_2'\,\{e_2/x\} = E_2'[\Uparrow\ell]$. Since $\langle \{y{:}T \mid e_1'\}\,\{e_2/x\}, E_2', v'\,\{e_2/x\} \rangle^\ell$ is an evaluation context by Lemma C.2.2 and $e\,\{e_2/x\} = \langle \{y{:}T \mid e_1'\}\,\{e_2/x\}, E_2'[\Uparrow\ell], v'\,\{e_2/x\} \rangle^\ell$, we finish. □

Lemma C.2.18 (Weak bisimulation, left side)**.** *Let e_1 and e_2 be terms such that $e_1 \longrightarrow e_2$. If $e\,\{e_1/x\} \longrightarrow e'$, then $e\,\{e_2/x\} \longrightarrow^* e''\,\{e_2/x\}$ for some e'' such that $e' = e''\,\{e_1/x\}$. Moreover, if $e\,\{e_1/x\} \longrightarrow e'$ is derived by (*E_RED*), then the evaluation $e\,\{e_2/x\} \longrightarrow^* e''\,\{e_2/x\}$ is derived by applying only (*E_RED*).*

Proof. By structural induction on e. If $e\,\{e_1/x\} \longrightarrow e'$ is derived by (E_BLAME), then there exist some E_1 and ℓ such that $e\,\{e_1/x\} = E_1[\Uparrow\ell]$ and $e' = \Uparrow\ell$. By Lemma C.2.17, there exists some E_2 such that $e\,\{e_2/x\} = E_2[\Uparrow\ell]$. Thus, by (E_BLAME), $e\,\{e_2/x\} \longrightarrow \Uparrow\ell$.

In what follows, we suppose that $e\,\{e_1/x\} \longrightarrow e'$ is derived by (E_RED). We proceed by case analysis on e. Note that e_1 is not a value from $e_1 \longrightarrow e_2$.

Case $e = y$: If $x = y$, then we have $e\,\{e_1/x\} = e_1$ and $e\,\{e_2/x\} = e_2$. We finish by letting $e'' = e_2$ because $e\,\{e_1/x\} = e_1 \longrightarrow e_2$ and $e_2\,\{e_1/x\} = e_2\,\{e_2/x\} = e_2$. Note that e_2 is closed since the evaluation relation is defined over closed terms.

Otherwise, if $x \neq y$, then there is a contradiction because the assumption says that $e\,\{e_1/x\} = y$ takes a step.

Case $e = \Uparrow\ell$: Contradictory.

Case $e = v$: Contradictory by Lemma C.2.2 since $e\,\{e_1/x\}$ takes a step.

Case $e = e_1'\ e_2'$: Since $e\,\{e_1/x\}$ takes a step, there are three cases we have to consider.

- Case $e_1'\,\{e_1/x\} \longrightarrow e''$ by (E_RED): By the IH, there exists some e_1'' such that $e_1'\,\{e_2/x\} \longrightarrow^* e_1''\,\{e_2/x\}$ and $e'' = e_1''\,\{e_1/x\}$. Moreover, the evaluation $e_1'\,\{e_2/x\} \longrightarrow^* e_1''\,\{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e_1'\ e_2')\,\{e_1/x\} \longrightarrow (e_1''\ e_2')\,\{e_1/x\}$ and $(e_1'\ e_2')\,\{e_2/x\} \longrightarrow^* (e_1''\ e_2')\,\{e_2/x\}$.
- Case $e_1'\,\{e_1/x\}$ is a value and $e_2'\,\{e_1/x\} \longrightarrow e''$ by (E_RED): By Lemmas C.2.3 and C.2.2, $e_1'\,\{e_2/x\}$ is a value. By the IH, there exists some e_2'' such that $e_2'\,\{e_2/x\} \longrightarrow^* e_2''\,\{e_2/x\}$ and $e'' = e_2''\,\{e_1/x\}$. Moreover, the evaluation $e_2'\,\{e_2/x\} \longrightarrow^* e_2''\,\{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e_1'\ e_2')\,\{e_1/x\} \longrightarrow (e_1'\ e_2'')\,\{e_1/x\}$ and $(e_1'\ e_2')\,\{e_2/x\} \longrightarrow^* (e_1'\ e_2'')\,\{e_2/x\}$.
- Case $e_1'\,\{e_1/x\}$ and $e_2'\,\{e_1/x\}$ are values: Since e_1' and e_2' are values by Lemma C.2.3, we finish by Lemma C.2.5 (1).

Case $e = (e_1', e_2')$: Similarly to the case for application term. Since $e\,\{e_1/x\}$ takes a step, there are two cases we have to consider.

Case $e_1' \{e_1/x\} \longrightarrow e''$ by (E_RED): By the IH, there exists some e_1'' such that $e_1' \{e_2/x\} \longrightarrow^* e_1'' \{e_2/x\}$ and $e'' = e_1'' \{e_1/x\}$. Moreover, the evaluation $e_1' \{e_2/x\} \longrightarrow^* e_1'' \{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e_1', e_2') \{e_1/x\} \longrightarrow (e_1'', e_2') \{e_1/x\}$ and $(e_1', e_2') \{e_2/x\} \longrightarrow^* (e_1'', e_2') \{e_2/x\}$.

Case $e_1' \{e_1/x\}$ is a value and $e_2' \{e_1/x\} \longrightarrow e''$ by (E_RED): By Lemmas C.2.3 and C.2.2, $e_1' \{e_2/x\}$ is a value. By the IH, there exists some e_2'' such that $e_2' \{e_2/x\} \longrightarrow^* e_2'' \{e_2/x\}$ and $e'' = e_2'' \{e_1/x\}$. Moreover, the evaluation $e_2' \{e_2/x\} \longrightarrow^* e_2'' \{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e_1', e_2') \{e_1/x\} \longrightarrow (e_1', e_2'') \{e_1/x\}$ and $(e_1', e_2') \{e_2/x\} \longrightarrow^* (e_1', e_2'') \{e_2/x\}$.

Case $e = e'.\, i$ for $i \in \{1, 2\}$: Similarly to the case for application term except for use of Lemma C.2.7 (1). Since $e \{e_1/x\}$ takes a step, there are two cases we have to consider.

Case $e' \{e_1/x\} \longrightarrow e''$ by (E_RED): By the IH, there exists some e''' such that $e' \{e_2/x\} \longrightarrow^* e''' \{e_2/x\}$ and $e'' = e''' \{e_1/x\}$. Moreover, the evaluation $e' \{e_2/x\} \longrightarrow^* e''' \{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e'.\, i) \{e_1/x\} \longrightarrow (e'''.\, i) \{e_1/x\}$ and $(e'.\, i) \{e_2/x\} \longrightarrow^* (e'''.\, i) \{e_2/x\}$.

Case $e' \{e_1/x\}$ is a value: Since e' is a value by Lemma C.2.3, we finish by Lemma C.2.7 (1).

Case $e = C\langle e_1' \rangle e_2'$: Similarly to the case for application term. Since $e \{e_1/x\}$ takes a step, it is found that $e_2' \{e_1/x\} \longrightarrow e''$ by (E_RED) for some e''. By the IH, there exists some e_2'' such that $e_2' \{e_2/x\} \longrightarrow^* e_2'' \{e_2/x\}$ and $e'' = e_2'' \{e_1/x\}$. Moreover, the evaluation $e_2' \{e_2/x\} \longrightarrow^* e_2'' \{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(C\langle e_1' \rangle e_2') \{e_1/x\} \longrightarrow (C\langle e_1' \rangle e_2'') \{e_1/x\}$ and $(C\langle e_1' \rangle e_2') \{e_2/x\} \longrightarrow^* (C\langle e_1' \rangle e_2'') \{e_2/x\}$.

Case $e = \mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}$: Similarly to the case for application term except for use of Lemma C.2.11 (1). Since $e \{e_1/x\}$ takes a step, there are two cases we have to consider.

Case $e_0' \{e_1/x\} \longrightarrow e''$ by (E_RED): By the IH, there exists some e_0'' such that $e_0' \{e_2/x\} \longrightarrow^* e_0'' \{e_2/x\}$ and $e'' = e_0'' \{e_1/x\}$. Moreover, the evaluation $e_0' \{e_2/x\} \longrightarrow^* e_0'' \{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1),

$$(\mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}) \{e_1/x\} \longrightarrow (\mathsf{match}\ e_0''\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}) \{e_1/x\}$$
$$(\mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}) \{e_2/x\} \longrightarrow^* (\mathsf{match}\ e_0''\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}) \{e_2/x\}.$$

Case $e_0' \{e_1/x\}$ is a value: Since e_0' is a value by Lemma C.2.3, we finish by Lemma C.2.11 (1).

Case $e = \mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'$: Similarly to the case for application term except for use of Lemma C.2.9 (1). Since $e \{e_1/x\}$ takes a step, there are two cases we have to consider.

Case $e_1' \{e_1/x\} \longrightarrow e''$ by (E_RED): By the IH, there exists some e_1'' such that $e_1' \{e_2/x\} \longrightarrow^* e_1'' \{e_2/x\}$ and $e'' = e_1'' \{e_1/x\}$. Moreover, the evaluation

$e_1'\{e_2/x\} \longrightarrow^* e_1''\{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1),

$$\begin{array}{rcl}(\mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3')\{e_1/x\} & \longrightarrow & (\mathsf{if}\ e_1''\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3')\{e_1/x\} \\ (\mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3')\{e_2/x\} & \longrightarrow^* & (\mathsf{if}\ e_1''\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3')\{e_2/x\}.\end{array}$$

Case $e_1'\{e_1/x\}$ is a value: Since e_1' is a value by Lemma C.2.3, we finish by Lemma C.2.9 (1).

Case $e = \langle\{y{:}T \mid e_1'\}, e_2', v\rangle^\ell$: Similarly to the case for application term except for use of Lemma C.2.15 (1). Since $e\{e_1/x\}$ takes a step, there are two cases we have to consider.

Case $e_2'\{e_1/x\} \longrightarrow e''$ by (E_RED): By the IH, there exists some e_2'' such that $e_2'\{e_2/x\} \longrightarrow^* e_2''\{e_2/x\}$ and $e'' = e_2''\{e_1/x\}$. Moreover, the evaluation $e_2'\{e_2/x\} \longrightarrow^* e_2''\{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1),

$$\begin{array}{rcl}(\langle\{y{:}T \mid e_1'\}, e_2', v\rangle^\ell)\{e_1/x\} & \longrightarrow & (\langle\{y{:}T \mid e_1'\}, e_2'', v\rangle^\ell)\{e_1/x\} \\ (\langle\{y{:}T \mid e_1'\}, e_2', v\rangle^\ell)\{e_2/x\} & \longrightarrow^* & (\langle\{y{:}T \mid e_1'\}, e_2'', v\rangle^\ell)\{e_2/x\}.\end{array}$$

Case $e_2'\{e_1/x\}$ is a value: Since e_2' is a value by Lemma C.2.3, we finish by Lemma C.2.15 (1).

Case $e = \langle\!\langle\{y{:}T \mid e_1'\}, e_2'\rangle\!\rangle^\ell$: Similarly to the case for application term except for use of Lemma C.2.13 (1). Since $e\{e_1/x\}$ takes a step, there are two cases we have to consider.

Case $e_2'\{e_1/x\} \longrightarrow e''$ by (E_RED): By the IH, there exists some e_2'' such that $e_2'\{e_2/x\} \longrightarrow^* e_2''\{e_2/x\}$ and $e'' = e_2''\{e_1/x\}$. Moreover, the evaluation $e_2'\{e_2/x\} \longrightarrow^* e_2''\{e_2/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1),

$$\begin{array}{rcl}(\langle\!\langle\{y{:}T \mid e_1'\}, e_2'\rangle\!\rangle^\ell)\{e_1/x\} & \longrightarrow & (\langle\!\langle\{y{:}T \mid e_1'\}, e_2''\rangle\!\rangle^\ell)\{e_1/x\} \\ (\langle\!\langle\{y{:}T \mid e_1'\}, e_2'\rangle\!\rangle^\ell)\{e_2/x\} & \longrightarrow^* & (\langle\!\langle\{y{:}T \mid e_1'\}, e_2''\rangle\!\rangle^\ell)\{e_2/x\}.\end{array}$$

Case $e_2'\{e_1/x\}$ is a value: Since e_2' is a value by Lemma C.2.3, we finish by Lemma C.2.13 (1). □

Lemma C.2.19. *If $e_1 \longrightarrow e_2$, and $e\{e_2/x\}$ is a value, then there exists some e' such that*

- $e\{e_1/x\} \longrightarrow^* e'\{e_1/x\}$,
- *$e'\{e_1/x\}$ is a value, and*
- $e\{e_2/x\} = e'\{e_2/x\}$.

Proof. By structural induction on e.

Case $e = y$: If $x = y$, then $e\{e_2/x\} = e_2$ is a value. Thus, we finish by letting $e' = e_2$ because $e_2\{e_1/x\} = e_2\{e_2/x\} = e_2$. Note that e_2 is closed since the evaluation relation is defined over closed terms. Otherwise, if $x \neq y$, then contradiction because $e\{e_2/x\}$ is a value but $e\{e_2/x\} = y$ is not.

Case $e = v$: Obvious by letting $e' = v$ because $v\{e_1/x\}$ is a value by Lemma C.2.2.

Case $e = \Uparrow\ell$, $e_1'\ e_2'$, $e'.i$ for $i \in \{1,2\}$, match e_0' with $\overline{C_i\ y_i \to e_i'}^{\,i}$, if e_1' then e_2' else e_3', $\langle\{y{:}T \mid e_1'\}, e_2', v\rangle^\ell$ or $\langle\!\langle\{y{:}T \mid e_1'\}, e_2'\rangle\!\rangle^\ell$: Contradictory: $e\{e_2/x\}$ is a value.

Case $e = (e_1', e_2')$: Let $i \in \{1,2\}$. By the assumption, $e_i'\{e_2/x\}$ is a value. By the IH, there exists some e_i'' such that $e_i'\{e_1/x\} \longrightarrow^* e_i''\{e_1/x\}$ and $e_i''\{e_1/x\}$ is a value and $e_i'\{e_2/x\} = e_i''\{e_2/x\}$. Thus, $(e_1', e_2')\{e_1/x\} \longrightarrow^* (e_1'', e_2'')\{e_1/x\}$ and $(e_1'', e_2'')\{e_1/x\}$ is a value and $e\{e_2/x\} = (e_1'', e_2'')\{e_2/x\}$.

Case $e = C\langle e_1'\rangle e_2'$: By the assumption, $e_2'\{e_2/x\}$ is a value. By the IH, there exists some e_2'' such that $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ and $e_2''\{e_1/x\}$ is a value and $e_2'\{e_2/x\} = e_2''\{e_2/x\}$. Thus, $(C\langle e_1'\rangle e_2')\{e_1/x\} \longrightarrow^* (C\langle e_1'\rangle e_2'')\{e_1/x\}$ and $(C\langle e_1'\rangle e_2'')\{e_1/x\}$ is a value and $(C\langle e_1'\rangle e_2')\{e_2/x\} = (C\langle e_1'\rangle e_2'')\{e_2/x\}$. □

Lemma C.2.20. *If $e_1 \longrightarrow e_2$ and $e\{e_2/x\} = E_2[\Uparrow\ell]$, then $e\{e_1/x\} \longrightarrow^* \Uparrow\ell$.*

Proof. By structural induction on e.

Case $e = x$: Obvious since $e_1 \longrightarrow e_2 = e\{e_2/x\} = E_2[\Uparrow\ell] \longrightarrow \Uparrow\ell$.

Case $e = \Uparrow\ell$: Obvious.

Case $e = y$ where $y \neq x$, $\Uparrow\ell'$ where $\ell \neq \ell'$, and v: Contradictory (by Lemma C.2.3 in the case that $e = v$) since $e\{e_2/x\} = E_2[\Uparrow\ell]$.

Case $e = e_1'\ e_2'$: Since $e\{e_2/x\} = E_2[\Uparrow\ell]$, there are two cases we have to consider.

- Case $E_2 = E_2'\ e_2'\{e_2/x\}$: Since $e_1'\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e_1'\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. Thus, we finish by Lemma C.2.16 (2).
- Case $E_2 = e_1'\{e_2/x\}\ E_2'$ where $e_1'\{e_2/x\}$ is a value: By Lemma C.2.19, there exists some e_1'' such that $e_1'\{e_1/x\} \longrightarrow^* e_1''\{e_1/x\}$ and $e_1''\{e_1/x\}$ is a value and $e_1'\{e_2/x\} = e_1''\{e_2/x\}$. Since $e_2'\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e_2'\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. Thus, $(e_1'\ e_2')\{e_1/x\} \longrightarrow^* (e_1''\ e_2')\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by Lemmas C.2.16 (1) and (2).

Case $e = (e_1', e_2')$: Since $e\{e_2/x\} = E_2[\Uparrow\ell]$, there are two cases we have to consider.

- Case $E_2 = (E_2', e_2'\{e_2/x\})$: Since $e_1'\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e_1'\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. Thus, we finish by Lemma C.2.16 (2).
- Case $E_2 = (e_1'\{e_2/x\}, E_2')$ where $e_1'\{e_2/x\}$ is a value: By Lemma C.2.19, there exists some e_1'' such that $e_1'\{e_1/x\} \longrightarrow^* e_1''\{e_1/x\}$ and $e_1''\{e_1/x\}$ is a value and $e_1'\{e_2/x\} = e_1''\{e_2/x\}$. Since $e_2'\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e_2'\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. Thus, $(e_1', e_2')\{e_1/x\} \longrightarrow^* (e_1'', e_2')\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by Lemmas C.2.16 (1) and (2).

Case $e = e'.i$ for $i \in \{1,2\}$: Since $e\{e_2/x\} = E_2[\Uparrow\ell]$, there exists some E_2' such that $E_2 = E_2'.i$. Since $e'\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e'\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. By Lemma C.2.16 (2), we finish.

Case $e = C\langle e_1'\rangle e_2'$: Since $e\{e_2/x\} = E_2[\Uparrow\ell]$, there exists some E_2' such that $E_2 = C\langle e_1'\{e_2/x\}\rangle E_2'$. Since $e_2'\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e_2'\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. By Lemma C.2.16 (2), we finish.

Case $e = \mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}$: Since $e\,\{e_2/x\} = E_2[\Uparrow\ell]$, there exists some E_2' such that $E_2 = \mathsf{match}\ E_2'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}$. Since $e_0'\,\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e_0'\,\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. By Lemma C.2.16 (2), we finish.

Case $e = \mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'$: Since $e\,\{e_2/x\} = E_2[\Uparrow\ell]$, there exists some E_2' such that $E_2 = \mathsf{if}\ E_2'\ \mathsf{then}\ e_2'\,\{e_2/x\}\ \mathsf{else}\ e_3'\,\{e_2/x\}$. Since $e_1'\,\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e_1'\,\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. By Lemma C.2.16 (2), we finish.

Case $e = \langle\{y{:}T \mid e_1'\}, e_2', v\rangle^{\ell'}$: Since $e\,\{e_2/x\} = E_2[\Uparrow\ell]$, there exists some E_2' such that $E_2 = \langle\{y{:}T \mid e_1'\}\,\{e_2/x\}, E_2', v\,\{e_2/x\}\rangle^{\ell'}$. Since $e_2'\,\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e_2'\,\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. By Lemma C.2.16 (2), we finish.

Case $e = \langle\!\langle\{y{:}T \mid e_1'\}, e_2'\rangle\!\rangle^{\ell'}$: Since $e\,\{e_2/x\} = E_2[\Uparrow\ell]$, there exists some E_2' such that $E_2 = \langle\!\langle\{y{:}T \mid e_1'\}\,\{e_2/x\}, E_2'\rangle\!\rangle^{\ell'}$. Since $e_2'\,\{e_2/x\} = E_2'[\Uparrow\ell]$, we have $e_2'\,\{e_1/x\} \longrightarrow^* \Uparrow\ell$ by the IH. By Lemma C.2.16 (2), we finish. □

Lemma C.2.21 (Weak bisimulation, right side). *Suppose that $e_1 \longrightarrow e_2$. If $e\,\{e_2/x\} \longrightarrow e'$, then $e\,\{e_1/x\} \longrightarrow^* e''\,\{e_1/x\}$ for some e'' such that $e' = e''\,\{e_2/x\}$. Moreover, if $e\,\{e_2/x\} \longrightarrow e'$ is derived by (E_RED), then the evaluation $e\,\{e_1/x\} \longrightarrow^* e''\,\{e_1/x\}$ is derived by applying only (E_RED).*

Proof. By structural induction on e. If $e\,\{e_2/x\} \longrightarrow e'$ is derived by (E_BLAME), then there exist some E_2 and ℓ such that $e\,\{e_2/x\} = E_2[\Uparrow\ell]$ and $e' = \Uparrow\ell$. By Lemma C.2.20, $e\,\{e_1/x\} \longrightarrow^* \Uparrow\ell$. We finish by letting $e'' = \Uparrow\ell$.

In what follows, we suppose that $e\,\{e_2/x\}$ is derived by (E_RED). We proceed by case analysis on e.

Case $e = y$: If $x = y$, then we have $e\,\{e_1/x\} = e_1$ and $e\,\{e_2/x\} = e_2$. Thus, we finish by letting $e_1' = e_2'$ because $e_2'\,\{e_1/x\} = e_2'\,\{e_2/x\} = e_2'$. Note that the evaluation relation is defined over closed terms. Otherwise, if $x \neq y$, then contradiction because $e\,\{e_2/x\} = y$ takes a step.

Case $e = \Uparrow\ell$: Contradictory.

Case $e = v$: Contradictory by Lemma C.2.2 since $e\,\{e_2/x\} \longrightarrow e_2'$.

Case $e = e_1'\ e_2'$: Since $e\,\{e_2/x\}$ takes a step, there are three cases we have to consider.

- Case $e_1'\,\{e_2/x\} \longrightarrow e''$ by (E_RED): By the IH, there exists some e_1'' such that $e_1'\,\{e_1/x\} \longrightarrow^* e_1''\,\{e_1/x\}$ and $e'' = e_1''\,\{e_2/x\}$. Moreover, the evaluation $e_1'\,\{e_1/x\} \longrightarrow^* e_1''\,\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e_1'\ e_2')\,\{e_2/x\} \longrightarrow (e_1''\ e_2')\,\{e_2/x\}$ and $(e_1'\ e_2')\,\{e_1/x\} \longrightarrow^* (e_1''\ e_2')\,\{e_1/x\}$.
- Case $e_1'\,\{e_2/x\}$ is a value and $e_2'\,\{e_2/x\} \longrightarrow e''$ by (E_RED): By Lemma C.2.19, there exists some e_1'' such that $e_1'\,\{e_1/x\} \longrightarrow^* e_1''\,\{e_1/x\}$ and $e_1''\,\{e_1/x\}$ is a value and $e_1'\,\{e_2/x\} = e_1''\,\{e_2/x\}$. By the IH, there exists some e_2'' such that $e_2'\,\{e_1/x\} \longrightarrow^* e_2''\,\{e_1/x\}$ and $e'' = e_2''\,\{e_2/x\}$. Moreover, the evaluation $e_2'\,\{e_1/x\} \longrightarrow^* e_2''\,\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e_1'\ e_2')\,\{e_2/x\} \longrightarrow (e_1''\ e_2'')\,\{e_2/x\}$ and $(e_1'\ e_2')\,\{e_1/x\} \longrightarrow^* (e_1''\ e_2'')\,\{e_1/x\}$.
- Case $e_1'\,\{e_2/x\}$ and $e_2'\,\{e_2/x\}$ are values: Let $i \in \{1, 2\}$. By Lemma C.2.19, there exist some e_i'' such that $e_i'\,\{e_1/x\} \longrightarrow^* e_i''\,\{e_1/x\}$ and $e_i''\,\{e_1/x\}$ is a value and $e_i'\,\{e_2/x\} = e_i''\,\{e_2/x\}$. Since e_1'' and e_2'' are values by Lemma C.2.3, we finish by Lemmas C.2.5 (2) and C.2.16 (1).

Case $e = (e_1', e_2')$: Similarly to the case for application term. Since $e\{e_2/x\}$ takes a step, there are two cases we have to consider.

- Case $e_1'\{e_2/x\} \longrightarrow e''$ by (E_RED): By the IH, there exists some e_1'' such that $e_1'\{e_1/x\} \longrightarrow^* e_1''\{e_1/x\}$ and $e'' = e_1''\{e_2/x\}$. Moreover, the evaluation $e_1'\{e_1/x\} \longrightarrow^* e_1''\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e_1', e_2')\{e_2/x\} \longrightarrow (e_1'', e_2')\{e_2/x\}$ and $(e_1', e_2')\{e_1/x\} \longrightarrow^* (e_1'', e_2')\{e_1/x\}$.
- Case $e_1'\{e_2/x\}$ is a value and $e_2'\{e_2/x\} \longrightarrow e''$ by (E_RED): By Lemma C.2.19, there exists some e_1'' such that $e_1'\{e_1/x\} \longrightarrow^* e_1''\{e_1/x\}$ and $e_1''\{e_1/x\}$ is a value and $e_1'\{e_2/x\} = e_1''\{e_2/x\}$. By the IH, there exists some e_2'' such that $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ and $e'' = e_2''\{e_2/x\}$. Moreover, the evaluation $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e_1', e_2')\{e_2/x\} \longrightarrow (e_1'', e_2'')\{e_2/x\}$ and $(e_1', e_2')\{e_1/x\} \longrightarrow^* (e_1'', e_2'')\{e_1/x\}$.

Case $e = e'.\,i$ for $i \in \{1, 2\}$: Similarly to the case for application term except for use of Lemma C.2.7 (2). If there exists some e'' such that $e'\{e_2/x\} \longrightarrow e''$ by (E_RED), then, by the IH, there exists some e''' such that $e'\{e_1/x\} \longrightarrow^* e'''\{e_1/x\}$ and $e'' = e'''\{e_2/x\}$. Moreover, the evaluation $e'\{e_1/x\} \longrightarrow^* e'''\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(e'.\,i)\{e_2/x\} \longrightarrow (e'''.\,i)\{e_2/x\}$ and $(e'.\,i)\{e_1/x\} \longrightarrow^* (e'''.\,i)\{e_1/x\}$. Otherwise, if $e'\{e_2/x\}$ is a value, then there exists some e'' such that $e'\{e_1/x\} \longrightarrow^* e''\{e_1/x\}$ and $e''\{e_1/x\}$ is a value and $e'\{e_2/x\} = e''\{e_2/x\}$. Since e'' is a value by Lemma C.2.3, we finish by Lemmas C.2.7 (2) and C.2.16 (1).

Case $e = C\langle e_1'\rangle e_2'$: Similarly to the case for application term. Since $e\{e_2/x\}$ takes a step, there exists some e'' such that $e_2'\{e_2/x\} \longrightarrow e''$ by (E_RED). By the IH, there exists some e_2'' such that $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ and $e'' = e_2''\{e_2/x\}$. Moreover, the evaluation $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1), $(C\langle e_1'\rangle e_2')\{e_2/x\} \longrightarrow (C\langle e_1'\rangle e_2'')\{e_2/x\}$ and $(C\langle e_1'\rangle e_2')\{e_1/x\} \longrightarrow^* (C\langle e_1'\rangle e_2'')\{e_1/x\}$.

Case $e = \mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i}$: Similarly to the case for application term except for use of Lemma C.2.11 (2). If there exists some e'' such that $e_0'\{e_2/x\} \longrightarrow e''$ by (E_RED), then, by the IH, there exists some e_0'' such that $e_0'\{e_1/x\} \longrightarrow^* e_0''\{e_1/x\}$ and $e'' = e_0''\{e_2/x\}$. Moreover, the evaluation $e_0'\{e_1/x\} \longrightarrow^* e_0''\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1),

$$(\mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\{e_2/x\} \longrightarrow (\mathsf{match}\ e_0''\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\{e_2/x\}$$
$$(\mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\{e_1/x\} \longrightarrow^* (\mathsf{match}\ e_0''\ \mathsf{with}\ \overline{C_i\ y_i \to e_i'}^{\,i})\{e_1/x\}.$$

Otherwise, if $e_0'\{e_2/x\}$ is a value, then there exists some e_0'' such that $e_0'\{e_1/x\} \longrightarrow^* e_0''\{e_1/x\}$ and $e_0''\{e_1/x\}$ is a value and $e_0'\{e_2/x\} = e_0''\{e_2/x\}$. Since e_0'' is a value by Lemma C.2.3, we finish by Lemmas C.2.11 (2) and C.2.16 (1).

Case $e = \mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'$: Similarly to the case for application term except for use of Lemma C.2.9 (2). If there exists some e'' such that $e_1'\{e_2/x\} \longrightarrow e''$ by (E_RED), then, by the IH, there exists some e_1'' such that $e_1'\{e_1/x\} \longrightarrow^* e_1''\{e_1/x\}$ and

$e'' = e_1''\{e_2/x\}$. Moreover, the evaluation $e_1'\{e_1/x\} \longrightarrow^* e_1''\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1),

$$\begin{array}{rcl}(\mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3')\{e_2/x\} & \longrightarrow & (\mathsf{if}\ e_1''\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3')\{e_2/x\} \\ (\mathsf{if}\ e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3')\{e_1/x\} & \longrightarrow^* & (\mathsf{if}\ e_1''\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3')\{e_1/x\}.\end{array}$$

Otherwise, if $e_1'\{e_2/x\}$ is a value, then there exists some e_1'' such that $e_1'\{e_1/x\} \longrightarrow^* e_1''\{e_1/x\}$ and $e_1''\{e_1/x\}$ is a value and $e_1'\{e_2/x\} = e_1''\{e_2/x\}$. Since e_1'' is a value by Lemma C.2.3, we finish by Lemmas C.2.9 (2) and C.2.16 (1).

Case $e = \langle\{y{:}T \mid e_1'\}, e_2', v\rangle^\ell$: Similarly to the case for application term except for use of Lemma C.2.15 (2). If there exists some e'' such that $e_2'\{e_2/x\} \longrightarrow e''$ by (E_RED), then, by the IH, there exists some e_2'' such that $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ and $e'' = e_2''\{e_2/x\}$. Moreover, the evaluation $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1),

$$\begin{array}{rcl}(\langle\{y{:}T \mid e_1'\}, e_2', v\rangle^\ell)\{e_2/x\} & \longrightarrow & (\langle\{y{:}T \mid e_1'\}, e_2'', v\rangle^\ell)\{e_2/x\} \\ (\langle\{y{:}T \mid e_1'\}, e_2', v\rangle^\ell)\{e_1/x\} & \longrightarrow^* & (\langle\{y{:}T \mid e_1'\}, e_2'', v\rangle^\ell)\{e_1/x\}.\end{array}$$

Otherwise, if $e_2'\{e_2/x\}$ is a value, then there exists some e_2'' such that $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ and $e_2''\{e_1/x\}$ is a value and $e_2'\{e_2/x\} = e_2''\{e_2/x\}$. Since e_2'' is a value by Lemma C.2.3, we finish by Lemmas C.2.15 (2) and C.2.16 (1).

Case $e = \langle\!\langle\{y{:}T \mid e_1'\}, e_2'\rangle\!\rangle^\ell$: Similarly to the case for application term except for use of Lemma C.2.13 (2). If there exists some e'' such that $e_2'\{e_2/x\} \longrightarrow e''$ by (E_RED), then, by the IH, there exists some e_2'' such that $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ and $e'' = e_2''\{e_2/x\}$. Moreover, the evaluation $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ is derived by applying only (E_RED). Thus, by Lemma C.2.16 (1),

$$\begin{array}{rcl}(\langle\!\langle\{y{:}T \mid e_1'\}, e_2'\rangle\!\rangle^\ell)\{e_2/x\} & \longrightarrow & (\langle\!\langle\{y{:}T \mid e_1'\}, e_2''\rangle\!\rangle^\ell)\{e_2/x\} \\ (\langle\!\langle\{y{:}T \mid e_1'\}, e_2'\rangle\!\rangle^\ell)\{e_1/x\} & \longrightarrow^* & (\langle\!\langle\{y{:}T \mid e_1'\}, e_2''\rangle\!\rangle^\ell)\{e_1/x\}.\end{array}$$

Otherwise, if $e_2'\{e_2/x\}$ is a value, then there exists some e_2'' such that $e_2'\{e_1/x\} \longrightarrow^* e_2''\{e_1/x\}$ and $e_2''\{e_1/x\}$ is a value and $e_2'\{e_2/x\} = e_2''\{e_2/x\}$. Since e_2'' is a value by Lemma C.2.3, we finish by Lemmas C.2.13 (2) and C.2.16 (1). □

Lemma C.2.22. *Suppose that* $e_1 \longrightarrow e_2$.

(1) If $e\{e_1/x\} \longrightarrow^* v_1$, *then* $e\{e_2/x\} \longrightarrow^* e'\{e_2/x\}$ *for some* e' *such that* $v_1 = e'\{e_1/x\}$, *and* $e'\{e_2/x\}$ *is a value.*

(2) If $e\{e_2/x\} \longrightarrow^* v_2$, *then* $e\{e_1/x\} \longrightarrow^* e'\{e_1/x\}$ *for some* e' *such that* $v_2 = e'\{e_2/x\}$, *and* $e'\{e_1/x\}$ *is a value.*

Proof.

1. By mathematical induction on the number of evaluation steps of $e\{e_1/x\}$.

   Case 0: We are given $e\{e_1/x\}$ is a value. Since e_1 is not a value from $e_1 \longrightarrow e_2$, we find that e is a value by Lemma C.2.3. By Lemma C.2.2, so is $e\{e_2/x\}$. Thus, we finish when letting $e' = e$.

Case $i+1$: We are given $e\{e_1/x\} \longrightarrow e_1' \longrightarrow^i v_1$. By Lemma C.2.18, there exists some e'' such that $e\{e_2/x\} \longrightarrow^* e''\{e_2/x\}$ and $e_1' = e''\{e_1/x\}$. By the IH, there exists some e' such that $e''\{e_2/x\} \longrightarrow^* e'\{e_2/x\}$ and $v_1 = e'\{e_1/x\}$, and $e'\{e_2/x\}$ is a value. Thus, we finish.

2. By mathematical induction on the number of evaluation steps of $e\{e_2/x\}$.

Case 0: We are given $e\{e_2/x\}$ is a value. By Lemma C.2.19, there exists some e' such that $e\{e_1/x\} \longrightarrow^* e'\{e_1/x\}$ and $e\{e_2/x\} = e'\{e_2/x\}$ and $e'\{e_1/x\}$ is a value.

Case $i+1$: We are given $e\{e_2/x\} \longrightarrow e_2' \longrightarrow^i v_2$. By Lemma C.2.21, there exists some e'' such that $e\{e_1/x\} \longrightarrow^* e''\{e_1/x\}$ and $e_2' = e''\{e_2/x\}$. By the IH, there exists some e' such that $e''\{e_1/x\} \longrightarrow^* e'\{e_1/x\}$ and $v_2 = e'\{e_2/x\}$, and $e'\{e_1/x\}$ is a value. Thus, we finish. □

Lemma C.2.23. *Suppose that* $e_1 \Rrightarrow^* e_2$.

(1) If $e_1 \longrightarrow^* v_1$, *then* $e_2 \longrightarrow^* v_2$ *for some* v_2 *such that* $v_1 \Rrightarrow^* v_2$.

(2) If $e_2 \longrightarrow^* v_2$, *then* $e_1 \longrightarrow^* v_1$ *for some* v_1 *such that* $v_1 \Rrightarrow^* v_2$.

Proof. By mathematical induction on the number of steps of $e_1 \Rrightarrow^* e_2$.

Case 0: Obvious because $e_1 = e_2$.

Case $i+1$: We are given $e_1 \Rrightarrow e_3 \Rrightarrow^i e_2$. We are given some e, e_1', e_3' and x such that $e_1 = e\{e_1'/x\}$ and $e_3 = e\{e_3'/x\}$ and $e_1' \longrightarrow e_3'$. Thus, we finish by Lemma C.2.22 and the IHs and transitivity of $\Rrightarrow^*$. □

Lemma C.2.24.

(1) If $c \Rrightarrow^* v$, *then* $v = c$.

(2) If $v \Rrightarrow^* c$, *then* $v = c$.

Proof.

1. By mathematical induction on the number of steps of $c \Rrightarrow^* v$.

Case 0: Obvious.

Case $i+1$: We are given $c \Rrightarrow e \Rrightarrow^* v$. We are given e', e_1, e_2 and x such that $c = e'\{e_1/x\}$ and $e = e'\{e_2/x\}$ and $e_1 \longrightarrow e_2$. Since e_1 is not a value from $e_1 \longrightarrow e_2$, we find that e' is a value by Lemma C.2.3. Thus, $e' = c$ and so $e = c$. By the IH, we finish.

2. By mathematical induction on the number of steps of $v \Rrightarrow^* c$.

Case 0: Obvious.

Case $i+1$: We are given $v \Rrightarrow e \Rrightarrow^* c$. We are given e', e_1, e_2 and x such that $v = e'\{e_1/x\}$ and $e = e'\{e_2/x\}$ and $e_1 \longrightarrow e_2$. Since e_1 is not a value from $e_1 \longrightarrow e_2$, we find that e' is a value by Lemma C.2.3. Thus, so is $e'\{e_2/x\}$ by Lemma C.2.2. By the IH, $e'\{e_2/x\} = c$. Since e' is a value, $e' = c$ and so $v = c$. □

Lemma 29 (Cotermination). *Suppose that* $e_1 \Rrightarrow^* e_2$.

(1) *If* $e_1 \longrightarrow^* \mathsf{true}$, *then* $e_2 \longrightarrow^* \mathsf{true}$.

(2) *If* $e_2 \longrightarrow^* \mathsf{true}$, *then* $e_1 \longrightarrow^* \mathsf{true}$.

Proof. By Lemmas C.2.23 and C.2.24.

Lemma C.2.25. *Suppose that* $e_1 \equiv e_2$.

(1) *If* $e_1 \longrightarrow^* \mathsf{true}$, *then* $e_2 \longrightarrow^* \mathsf{true}$.

(2) *If* $e_2 \longrightarrow^* \mathsf{true}$, *then* $e_1 \longrightarrow^* \mathsf{true}$.

Proof. Straightforward by induction on $e_1 \equiv e_2$. In particular, if $e_1 \Rrightarrow e_2$, then we finish by Lemma 29. □

C.3 Type Soundness

The proof of type soundness in $\lambda^H_{\mathbf{dt}}$ is also similar to the one in $\mathrm{F}^\sigma_{\mathrm{H}}$: we show Type Soundness (Theorem 9) via Progress (Lemma 27) and Preservation (Lemma 28), using standard and auxiliary lemmas.

Lemma C.3.1 (Weakening)**.** *Suppose that* x *is a fresh variable and* $\Gamma_1 \vdash T_1$.

(1) *If* $\Gamma_1, \Gamma_2 \vdash e : T$, *then* $\Gamma_1, x{:}T_1, \Gamma_2 \vdash e : T$.

(2) *If* $\Gamma_1, \Gamma_2 \vdash T$, *then* $\Gamma_1, x{:}T_1, \Gamma_2 \vdash T$.

(3) *If* $\vdash \Gamma_1, \Gamma_2$, *then* $\vdash \Gamma_1, x{:}T_1, \Gamma_2$.

Proof. Straightforward by induction on each derivation. □

Lemma C.3.2 (Substitution)**.** *Suppose that* $\Gamma_1 \vdash e' : T'$.

(1) *If* $\Gamma_1, x{:}T', \Gamma_2 \vdash e : T$, *then* $\Gamma_1, \Gamma_2\{e'/x\} \vdash e\{e'/x\} : T\{e'/x\}$.

(2) *If* $\Gamma_1, x{:}T', \Gamma_2 \vdash T$, *then* $\Gamma_1, \Gamma_2\{e'/x\} \vdash T\{e'/x\}$.

(3) *If* $\vdash \Gamma_1, x{:}T', \Gamma_2$, *then* $\vdash \Gamma_1, \Gamma_2\{e'/x\}$.

Proof. Straightforward by induction on each derivation. The only interesting cases are for (T_Ctr) and (T_Match).

Case (T_Ctr): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash C\langle e_1\rangle e_2 : \tau\langle e_1\rangle$ for some C, e_1, e_2 and τ. By inversion, we have $\mathit{TypSpecOf}(C) = y{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle y\rangle$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash e_1 : T_1$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash e_2 : T_2\{e_1/y\}$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash \tau\langle e_1\rangle$. Without loss of generality, we can suppose that y is fresh.

By the IHs, $\Gamma_1, \Gamma_2\{e'/x\} \vdash e_1\{e'/x\} : T_1\{e'/x\}$ and $\Gamma_1, \Gamma_2\{e'/x\} \vdash e_2\{e'/x\} : T_2\{e_1/y\}\{e'/x\}$ and $\Gamma_1, \Gamma_2\{e'/x\} \vdash \tau\langle e_1\{e'/x\}\rangle$. From well-formedness of the type definition environment, it is found that $T_1\{e'/x\} = T_1$ and $T_2\{e_1/y\}\{e'/x\} = T_2\{e_1\{e'/x\}/y\}$. Thus, we finish by (T_Ctr).

Case (T_MATCH): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ y_i \to e_i}^{\,i} : T$. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash e_0 : \tau\langle e''\rangle$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash T$ and $\mathit{CtrsOf}(\tau) = \overline{C_i}^{\,i}$ and $\mathit{ArgTypeOf}(\tau) = z{:}T''$ and, for all i, $\mathit{CtrArgOf}(C_i) = T_i$ and $\Gamma_1, x{:}T', \Gamma_2, y_i{:}T_i\,\{e''/z\} \vdash e_i : T$. Without loss of generality, we can suppose that $\overline{y_i}^{\,i}$ and z are fresh.

By the IHs, $\Gamma_1, \Gamma_2\,\{e'/x\} \vdash e_0\,\{e'/x\} : \tau\langle e''\,\{e'/x\}\rangle$ and $\Gamma_1, \Gamma_2\,\{e'/x\} \vdash T\,\{e'/x\}$ and $\Gamma_1, \Gamma_2\,\{e'/x\}, y_i{:}T_i\,\{e''/z\}\,\{e'/x\} \vdash e_i\,\{e'/x\} : T\,\{e'/x\}$. From well-formedness of the type definition environment, it is found that $T_i\,\{e''/z\}\,\{e'/x\} = T_i\,\{e''\,\{e'/x\}/z\}$. Thus, we finish by (T_MATCH). □

Lemma C.3.3 (Base Types Equivalence Inversion). *If* $T_1 \equiv T_2$, *then*

(1) $T_1 = \mathsf{bool}$ *implies* $T_2 = \mathsf{bool}$, *and*

(2) $T_2 = \mathsf{bool}$ *implies* $T_1 = \mathsf{bool}$.

Proof. Straightforward by induction on $T_1 \equiv T_2$. In particular, if $T_1 \Rrightarrow T_2$, then there exist some T, x, e_1 and e_2 such that $T_1 = T\,\{e_1/x\}$ and $T_2 = T\,\{e_2/x\}$. Since $T_1 = \mathsf{bool}$ or $T_2 = \mathsf{bool}$, we have $T = \mathsf{bool}$. Thus $T_1 = T_2 = \mathsf{bool}$. □

Lemma C.3.4 (Dependent Function Types Equivalence Inversion). *If* $T_1 \equiv T_2$, *then*

(1) $T_1 = x{:}T_{11} \to T_{12}$ *implies*

- $T = x{:}T_{21} \to T_{22}$,
- $T_{11} \equiv T_{21}$, *and*
- $T_{12} \equiv T_{22}$

for some T_{21} *and* T_{22}, *and*

(2) $T_2 = x{:}T_{21} \to T_{22}$ *implies*

- $T_1 = x{:}T_{11} \to T_{12}$,
- $T_{11} \equiv T_{21}$, *and*
- $T_{12} \equiv T_{22}$

for some T_{11} *and* T_{12}.

Proof. Straightforward by induction on $T_1 \equiv T_2$. In particular, if $T_1 \Rrightarrow T_2$, then there exist some T, y, e_1 and e_2 such that $T_1 = T\,\{e_1/y\}$ and $T_2 = T\,\{e_2/y\}$ and $e_1 \longrightarrow e_2$. Without loss of generality, we can suppose that x is fresh for e_1, e_2 and y. Since $T_1 = x{:}T_{11} \to T_{12}$ or $T_2 = x{:}T_{21} \to T_{22}$, we have $T = x{:}T_1 \to T_2$ for some T_1 and T_2. Thus, $T_1 = x{:}T_1\,\{e_1/y\} \to T_2\,\{e_1/y\}$ and $T_2 = x{:}T_1\,\{e_2/y\} \to T_2\,\{e_2/y\}$. We have $T_1\,\{e_1/y\} \Rrightarrow T_1\,\{e_2/y\}$ and $T_2\,\{e_1/y\} \Rrightarrow T_2\,\{e_2/y\}$ by definition. □

Lemma C.3.5 (Dependent Product Types Equivalence Inversion). *If* $T_1 \equiv T_2$, *then*

(1) $T_1 = x{:}T_{11} \times T_{12}$ *implies*

- $T_2 = x{:}T_{21} \times T_{22}$,
- $T_{11} \equiv T_{21}$, *and*
- $T_{12} \equiv T_{22}$

for some T_{21} *and* T_{22}*, and*

(2) $T_2 \equiv x{:}T_{21} \times T_{22}$ *implies*

- $T_1 = x{:}T_{11} \times T_{12}$,
- $T_{11} \equiv T_{21}$, *and*
- $T_{12} \equiv T_{22}$.

for some T_{11} *and* T_{12}.

Proof. Similarly to Lemma C.3.4, straightforward by induction on $T_1 \equiv T_2$. In particular, if $T_1 \Rrightarrow T_2$, then there exist some T, y, e_1 and e_2 such that $T_1 = T\{e_1/y\}$ and $T_2 = T\{e_2/y\}$ and $e_1 \longrightarrow e_2$. Without loss of generality, we can suppose that x is fresh for e_1, e_2 and y. Since $T_1 = x{:}T_{11} \times T_{12}$ or $T_2 = x{:}T_{21} \times T_{22}$, we have $T = x{:}T_1 \times T_2$ for some T_1 and T_2. Thus, $T_1 = x{:}T_1\{e_1/y\} \times T_2\{e_1/y\}$ and $T_2 = x{:}T_1\{e_2/y\} \times T_2\{e_2/y\}$. We have $T_1\{e_1/y\} \Rrightarrow T_1\{e_2/y\}$ and $T_2\{e_1/y\} \Rrightarrow T_2\{e_2/y\}$ by definition. □

Lemma C.3.6 (Datatypes Equivalence Inversion). *If* $T_1 \equiv T_2$*, then*

(1) $T_1 = \tau\langle e_1\rangle$ *implies* $T_2 = \tau\langle e_2\rangle$ *and* $e_1 \equiv e_2$ *for some* e_2*, and*

(2) $T_2 = \tau\langle e_2\rangle$ *implies* $T_1 = \tau\langle e_1\rangle$ *and* $e_1 \equiv e_2$ *for some* e_1.

Proof. Similarly to Lemma C.3.4, straightforward by induction on $T_1 \equiv T_2$. In particular, if $T_1 \Rrightarrow T_2$, then there exist some T, x, e_1' and e_2' such that $T_1 = T\{e_1'/x\}$ and $T_2 = T\{e_2'/x\}$ and $e_1' \longrightarrow e_2'$. Since $T_1 = \tau\langle e_1\rangle$ or $T_2 = \tau\langle e_2\rangle$, we have $T = \tau\langle e\rangle$ for some e. Thus, $T_1 = \tau\langle e\{e_1'/x\}\rangle$ and $T_2 = \tau\langle e\{e_2'/x\}\rangle$. We have $e\{e_1'/x\} \Rrightarrow e\{e_2'/x\}$ by definition. □

Lemma C.3.7 (Refinement Types Equivalence Inversion). *If* $T_1 \equiv T_2$*, then*

(1) $T_1 = \{x{:}T_1' \mid e_1'\}$ *implies*

- $T_2 = \{x{:}T_2' \mid e_2'\}$,
- $T_1' \equiv T_2'$, *and*
- $e_1' \equiv e_2'$

for some T_2' *and* e_2'*, and*

(2) $T_2 = \{x{:}T_2' \mid e_2'\}$ *implies*

- $T_1 = \{x{:}T_1' \mid e_1'\}$,
- $T_1' \equiv T_2'$, *and*
- $e_1' \equiv e_2'$

for some T_1' *and* e_1'.

Proof. Similarly to Lemma C.3.4, straightforward by induction on $T_1 \equiv T_2$. In particular, if $T_1 \Rrightarrow T_2$, then there exist some T, y, e_1'' and e_2'' such that $T_1 = T\{e_1''/y\}$ and $T_2 = T\{e_2''/y\}$ and $e_1'' \longrightarrow e_2''$. Without loss of generality, we can suppose that x is fresh for e_1'', e_2'' and y. Since $T_1 = \{x{:}T_1' \mid e_1'\}$ or $T_2 = \{x{:}T_2' \mid e_2'\}$, we have $T = \{x{:}T' \mid e'\}$ for some T' and e'. Thus, $T_1 = \{x{:}T'\{e_1''/y\} \mid e'\{e_1''/y\}\}$ and $T_2 = \{x{:}T'\{e_2''/y\} \mid e'\{e_2''/y\}\}$. We have $T'\{e_1''/y\} \Rrightarrow T'\{e_2''/y\}$ and $e'\{e_1''/y\} \Rrightarrow e'\{e_2''/y\}$ by definition. □

Lemma C.3.8 (Type Equivalence Closed Under Unrefine). *If* $T_1 \equiv T_2$, *then* $\mathit{unref}(T_1) \equiv \mathit{unref}(T_2)$.

Proof. By induction on T_1.

Case $T_1 = \mathsf{bool}$, $x{:}T_1' \to T_2'$, $x{:}T_1' \times T_2'$, or $\tau\langle e\rangle$: We have $\mathit{unref}(T_1) = T_1$. Since $T_1 \equiv T_2$, we find that $\mathit{unref}(T_2) = T_2$ by Lemmas C.3.3 (1), C.3.4 (1), C.3.5 (1) and C.3.6 (1). Thus, we finish.

Case $T_1 = \{x{:}T_1' \mid e_1'\}$: By Lemma C.3.7 (1), there exist some T_2' and e_2' such that $T_2 = \{x{:}T_2' \mid e_2'\}$ and $T_1' \equiv T_2'$. By the IH, $\mathit{unref}(T_1') \equiv \mathit{unref}(T_2')$. Because $\mathit{unref}(T_1) = \mathit{unref}(T_1')$ and $\mathit{unref}(T_2) = \mathit{unref}(T_2')$, we finish.

□

Lemma C.3.9 (Lambda Inversion). *If* $\Gamma \vdash \mathtt{fix}\, f(x{:}T_1){:}T_2 = e : T$, *then*

- $\Gamma, f{:}(x{:}T_1 \to T_2), x{:}T_1 \vdash e : T_2$,
- $f \notin \mathsf{FV}(T_2)$, *and*
- $x{:}T_1 \to T_2 \equiv \mathit{unref}(T)$.

Proof. By induction on the typing derivation. Only four rules can be applied to the lambda abstraction.

Case (T_ABS): Since $T = x{:}T_1 \to T_2$, we have $x{:}T_1 \to T_2 \equiv \mathit{unref}(T)$ by Lemma C.1.1 (reflexivity). By inversion, we finish.

Case (T_CONV): By inversion, we have $\emptyset \vdash \mathtt{fix}\, f(x{:}T_1){:}T_2 = e : T'$ and $T' \equiv T$ for some T'. By the IH, we have $f{:}(x{:}T_1 \to T_2), x{:}T_1 \vdash e : T_2$ and $f \notin \mathsf{FV}(T_2)$ and $x{:}T_1 \to T_2 \equiv \mathit{unref}(T')$. Because $\mathit{unref}(T') \equiv \mathit{unref}(T)$ by Lemma C.3.8, we have $x{:}T_1 \to T_2 \equiv \mathit{unref}(T)$ by Lemma C.1.1 (transitivity). By Lemma C.3.1, we finish.

Case (T_FORGET): By inversion, we have $\emptyset \vdash \mathtt{fix}\, f(x{:}T_1){:}T_2 = e : \{y{:}T \mid e'\}$ for some y and e'. By the IH, $f{:}(x{:}T_1 \to T_2), x{:}T_1 \vdash e : T_2$ and $f \notin \mathsf{FV}(T_2)$ and $x{:}T_1 \to T_2 \equiv \mathit{unref}(\{y{:}T \mid e'\})$. Since $\mathit{unref}(T) = \mathit{unref}(\{y{:}T \mid e'\})$, we have $x{:}T_1 \to T_2 \equiv \mathit{unref}(T)$. By Lemma C.3.1, we finish.

Case (T_EXACT): We are given $\Gamma \vdash \mathtt{fix}\, f(x{:}T_1){:}T_2 = e : \{y{:}T' \mid e'\}$ for some y, T' and e'. By inversion, we have $\emptyset \vdash \mathtt{fix}\, f(x{:}T_1){:}T_2 = e : T'$. By the IH, we have $f{:}(x{:}T_1 \to T_2), x{:}T_1 \vdash e : T_2$ and $f \notin \mathsf{FV}(T_2)$ and $x{:}T_1 \to T_2 \equiv \mathit{unref}(T')$. Since $\mathit{unref}(T') = \mathit{unref}(\{y{:}T' \mid e'\})$, we have $x{:}T_1 \to T_2 \equiv \mathit{unref}(\{y{:}T' \mid e'\})$. By Lemma C.3.1, we finish. □

Lemma C.3.10 (Cast Inversion). *If* $\Gamma \vdash \langle T_1 \Leftarrow T_2\rangle^{\ell} : T$, *then*

- $\Gamma \vdash T_1$,
- $\Gamma \vdash T_2$,
- $T_1 \parallel T_2$, *and*
- $T_2 \to T_1 \equiv \mathit{unref}(T)$.

Proof. Similarly to Lemma C.3.9, by induction on the typing derivation. Only four rules can be applied to the cast.

Case (T_CAST): Since $T = T_2 \to T_1$, we have $T_2 \to T_1 \equiv \mathit{unref}(T)$ by Lemma C.1.1 (reflexivity). By inversion, we finish.

Case (T_CONV): By inversion, we have $\emptyset \vdash \langle T_1 \Leftarrow T_2 \rangle^\ell : T'$ and $T' \equiv T$ for some T'. By the IH, we have $\emptyset \vdash T_1$ and $\emptyset \vdash T_2$ and $T_1 \parallel T_2$ and $T_2 \to T_1 \equiv \mathit{unref}(T')$. Because $\mathit{unref}(T') \equiv \mathit{unref}(T)$ by Lemma C.3.8, we have $T_2 \to T_1 \equiv \mathit{unref}(T)$ by Lemma C.1.1 (transitivity). By Lemma C.3.1, we finish.

Case (T_FORGET): By inversion, we have $\emptyset \vdash \langle T_1 \Leftarrow T_2 \rangle^\ell : \{y{:}T \mid e\}$ for some y and e. By the IH, $\emptyset \vdash T_1$ and $\emptyset \vdash T_2$ and $T_1 \parallel T_2$ and $T_2 \to T_1 \equiv \mathit{unref}(\{y{:}T \mid e\})$. Since $\mathit{unref}(\{y{:}T \mid e\}) = \mathit{unref}(T)$, we have $T_2 \to T_1 \equiv \mathit{unref}(T)$. By Lemma C.3.1, we finish.

Case (T_EXACT): We are given $\Gamma \vdash \langle T_1 \Leftarrow T_2 \rangle^\ell : \{x{:}T' \mid e'\}$ for some x, T' and e'. By inversion, we have $\emptyset \vdash \langle T_1 \Leftarrow T_2 \rangle^\ell : T'$. By the IH, we have $\emptyset \vdash T_1$ and $\emptyset \vdash T_2$ and $T_1 \parallel T_2$ and $T_2 \to T_1 \equiv \mathit{unref}(T')$. Since $\mathit{unref}(T') = \mathit{unref}(\{x{:}T' \mid e'\})$, we have $T_2 \to T_1 \equiv \mathit{unref}(\{x{:}T' \mid e'\})$. By Lemma C.3.1, we finish. □

Lemma C.3.11 (Pair Inversion). *If* $\Gamma \vdash (v_1, v_2) : T$, *then*

- $\Gamma \vdash v_1 : T_1$,
- $\Gamma \vdash v_2 : T_2\{v_1/x\}$,
- $\Gamma, x{:}T_1 \vdash T_2$, *and*
- $x{:}T_1 \times T_2 \equiv \mathit{unref}(T)$

for some T_1, T_2 *and* x.

Proof. Similarly to Lemma C.3.9, by induction on the typing derivation. Only four rules can be applied to the pair.

Case (T_PAIR): Since $T = x{:}T_1 \times T_2$, we have $x{:}T_1 \times T_2 \equiv \mathit{unref}(T)$ by Lemma C.1.1 (reflexivity). By inversion, we finish.

Case (T_CONV): By inversion, we have $\emptyset \vdash (v_1, v_2) : T'$ and $T' \equiv T$ for some T'. By the IH, we have $\emptyset \vdash v_1 : T_1$ and $\emptyset \vdash v_2 : T_2\{v_1/x\}$ and $x{:}T_1 \vdash T_2$ and $x{:}T_1 \times T_2 \equiv \mathit{unref}(T')$. Because $\mathit{unref}(T') \equiv \mathit{unref}(T)$ by Lemma C.3.8, we have $x{:}T_1 \times T_2 \equiv \mathit{unref}(T)$ by Lemma C.1.1 (transitivity). By Lemma C.3.1, we finish.

Case (T_FORGET): By inversion, we have $\emptyset \vdash (v_1, v_2) : \{y{:}T \mid e'\}$ for some y and e'. By the IH, we have $\emptyset \vdash v_1 : T_1$ and $\emptyset \vdash v_2 : T_2\{v_1/x\}$ and $x{:}T_1 \vdash T_2$ and $x{:}T_1 \times T_2 \equiv \mathit{unref}(\{y{:}T \mid e'\})$. Since $\mathit{unref}(\{y{:}T \mid e'\}) = \mathit{unref}(T)$, we have $x{:}T_1 \times T_2 \equiv \mathit{unref}(T)$. By Lemma C.3.1, we finish.

Case (T_EXACT): We are given $\Gamma \vdash (v_1, v_2) : \{y{:}T' \mid e'\}$ for some y, T' and e'. By inversion, we have $\emptyset \vdash (v_1, v_2) : T'$. By the IH, we have $\emptyset \vdash v_1 : T_1$ and $\emptyset \vdash v_2 : T_2\{v_1/x\}$ and $x{:}T_1 \vdash T_2$ and $x{:}T_1 \times T_2 \equiv \mathit{unref}(T')$. Since $\mathit{unref}(T') = \mathit{unref}(\{y{:}T' \mid e'\})$, we have $x{:}T_1 \times T_2 \equiv \mathit{unref}(\{y{:}T' \mid e'\})$. By Lemma C.3.1, we finish. □

Lemma C.3.12 (Constructor Inversion). *If* $\Gamma \vdash C\langle e \rangle v : T$, *then*

- $\mathit{TypSpecOf}(C) = x{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle x\rangle$,
- $\Gamma \vdash v : T_2\{e/x\}$,
- $\Gamma \vdash \tau\langle e\rangle$, *and*
- $\tau\langle e\rangle \equiv \mathit{unref}(T)$.

Proof. Similarly to Lemma C.3.9, by induction on the typing derivation. Only four rules can be applied to the constructor application.

Case (T_CTR): Since $T = \tau\langle e\rangle$, we have $\tau\langle e\rangle \equiv \mathit{unref}(T)$ by Lemma C.1.1 (reflexivity). By inversion, we finish.

Case (T_CONV): By inversion, we have $\emptyset \vdash C\langle e\rangle v : T'$ and $T' \equiv T$ for some T'. By the IH, we have $\mathit{TypSpecOf}(C) = x{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle x\rangle$ and $\emptyset \vdash v : T_2\{e/x\}$ and $\emptyset \vdash \tau\langle e\rangle$ and $\tau\langle e\rangle \equiv \mathit{unref}(T')$. Because $\mathit{unref}(T') \equiv \mathit{unref}(T)$ by Lemma C.3.8, we have $\tau\langle e\rangle \equiv \mathit{unref}(T)$ by Lemma C.1.1 (transitivity). By Lemma C.3.1, we finish.

Case (T_FORGET): By inversion, we have $\emptyset \vdash C\langle e\rangle v : \{y{:}T \mid e'\}$ for some y and e'. By the IH, we have $\mathit{TypSpecOf}(C) = x{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle x\rangle$ and $\emptyset \vdash v : T_2\{e/x\}$ and $\emptyset \vdash \tau\langle e\rangle$ and $\tau\langle e\rangle \equiv \mathit{unref}(\{y{:}T \mid e'\})$. Since $\mathit{unref}(\{y{:}T \mid e'\}) = \mathit{unref}(T)$, we have $\tau\langle e\rangle \equiv \mathit{unref}(T)$. By Lemma C.3.1, we finish.

Case (T_EXACT): We are given $\Gamma \vdash C\langle e\rangle v : \{y{:}T' \mid e'\}$ for some y, T' and e'. By inversion, we have $\emptyset \vdash C\langle e\rangle v : T'$. By the IH, we have $\mathit{TypSpecOf}(C) = x{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle x\rangle$ and $\emptyset \vdash v : T_2\{e/x\}$ and $\emptyset \vdash \tau\langle e\rangle$ and $\tau\langle e\rangle \equiv \mathit{unref}(T')$. Since $\mathit{unref}(T') = \mathit{unref}(\{y{:}T' \mid e'\})$, we have $\tau\langle e\rangle \equiv \mathit{unref}(\{y{:}T' \mid e'\})$. By Lemma C.3.1, we finish. □

Lemma C.3.13 (Canonical Forms). *Suppose that* $\emptyset \vdash v : T$.

(1) If $\mathit{unref}(T) = \mathsf{bool}$, *then* $v = \mathsf{true}$ *or* false.

(2) If $\mathit{unref}(T) = x{:}T_1 \to T_2$, *then*

  (a) $v = \mathtt{fix}\, f(x{:}T_1'){:}T_2' = e$ *for some* f, T_1', T_2' *and* e, *or*
  (b) $v = \langle T_2' \Leftarrow T_1'\rangle^{\ell}$ *for some* T_2', T_1' *and* ℓ.

(3) If $\mathit{unref}(T) = x{:}T_1 \times T_2$, *then* $v = (v_1, v_2)$ *for some* v_1 *and* v_2.

(4) If $\mathit{unref}(T) = \tau\langle e\rangle$, *then* $v = C\langle e'\rangle v'$ *for some* C, e' *and* v'.

Proof. By induction on the typing derivation.

Case (T_CONST): We are given $\emptyset \vdash c : \mathsf{bool}$. By inversion, $c \in \{\mathsf{true}, \mathsf{false}\}$. Since $\mathit{unref}(\mathsf{bool}) = \mathsf{bool}$, we are in the case (1).

Case (T_VAR), (T_BLAME), (T_APP), (T_PROJi) for $i \in \{1, 2\}$, (T_MATCH), (T_IF), (T_ACHECK), (T_WCHECK): Contradictory: v is a value.

Case (T_ABS): We are given $\emptyset \vdash \mathtt{fix}\, f(x{:}T_1){:}T_2 = e : x{:}T_1 \to T_2$. Since $\mathit{unref}(x{:}T_1 \to T_2) = x{:}T_1 \to T_2$, we are in the case (2).

Case (T_CAST): We are given $\emptyset \vdash \langle T_2 \Leftarrow T_1\rangle^{\ell} : T_1 \to T_2$. Since $\mathit{unref}(T_1 \to T_2) = T_1 \to T_2$, we are in the case (2).

Case (T_PAIR): We are given $\emptyset \vdash (v_1, v_2) : x{:}T_1 \times T_2$. Since $\mathit{unref}(x{:}T_1 \times T_2) = x{:}T_1 \times T_2$, we are in the case (3).

Case (T_CTR): We are given $\emptyset \vdash C\langle e' \rangle v' : \tau\langle e' \rangle$. Since $\mathit{unref}(\tau\langle e' \rangle) = \tau\langle e' \rangle$, we are in the case (4).

Case (T_CONV): By inversion, we have $\emptyset \vdash v : T'$ for some T' such that $T' \equiv T$. By Lemma C.3.8, $\mathit{unref}(T') \equiv \mathit{unref}(T)$. By case analysis on $\mathit{unref}(T')$:

- Case $\mathit{unref}(T') = \mathsf{bool}$: By the IH, $v \in \{\mathsf{true}, \mathsf{false}\}$. By Lemma C.3.3 (1), $\mathit{unref}(T) = \mathsf{bool}$ and so we are in the case (1).
- Case $\mathit{unref}(T') = x{:}T_1 \to T_2$: By the IH, v is a lambda abstraction or a cast. By Lemma C.3.4 (1), $\mathit{unref}(T) = x{:}T_1' \to T_2'$ for some T_1' and T_2' and so we are in the case (2).
- Case $\mathit{unref}(T') = x{:}T_1 \times T_2$: By the IH, $v = (v_1, v_2)$ for some v_1 and v_2. By Lemma C.3.5 (1), $\mathit{unref}(T) = x{:}T_1' \times T_2'$ for some T_1' and T_2' and so we are in the case (3).
- Case $\mathit{unref}(T') = \tau\langle e' \rangle$: By the IH, $v = C\langle e'' \rangle v''$ for some e'' and v''. By Lemma C.3.6 (1), $\mathit{unref}(T) = \tau\langle e''' \rangle$ for some e''' and so we are in the case (4).

Case (T_FORGET): By inversion, we have $\emptyset \vdash v : \{x{:}T \mid e\}$ for some x and e. Since $\mathit{unref}(T) = \mathit{unref}(\{x{:}T \mid e\})$, we finish by the IH.

Case (T_EXACT): We are given $\emptyset \vdash v : \{x{:}T' \mid e\}$ for some x, T' and e. By inversion, we have $\emptyset \vdash v : T'$. Since $\mathit{unref}(\{x{:}T' \mid e\}) = \mathit{unref}(T')$, we finish by the IH. □

Lemma 27 (Progress). *If $\emptyset \vdash e : T$, then*

1. *$e \longrightarrow e'$ for some e',*
2. *e is a value, or*
3. *$e = \Uparrow\ell$ for some ℓ.*

Proof. By induction on the typing derivation.

Case (T_CONST), (T_BLAME), (T_ABS), (T_CAST), (T_FORGET), (T_EXACT): The term e is a blaming or a value.

Case (T_VAR): Contradictory: $\emptyset \vdash x : T$ cannot be derived for any x.

Case (T_APP): We are given $\emptyset \vdash e_1\, e_2 : T_2\{e_2/x\}$ for some e_1, e_2, T_2 and x. By inversion, we have $\emptyset \vdash e_1 : x{:}T_1 \to T_2$ and $\emptyset \vdash e_2 : T_1$ for some T_1.

By the IH, e_1 and e_2 are reducible, values, or blaming. If e_1 is reducible or a blaming, then $e_1\, e_2$ steps by one of evaluation rules. If e_1 is a value and e_2 is reducible or a blaming, then $e_1\, e_2$ steps by one of evaluation rules. Otherwise, if e_1 and e_2 are values, then there are two cases which we consider on e_1 by Lemma C.3.13.

- Case $e_1 = \mathtt{fix}\, f(x{:}T_1') = e_{12}$: The term $e_1\, e_2$ steps by (E_RED)/(R_BETA).
- Case $e_1 = \langle T_1' \Leftarrow T_2' \rangle^{\ell}$: If T_2' is a refinement type, then we finish by (E_RED)/(R_FORGET). In the following, we suppose that T_2' is not a refinement type. By Lemma C.3.10, we have $T_1' \parallel T_2'$ and $T_2' \to T_1' \equiv x{:}T_1 \to T_2$ We perform case analysis on T_1'.

Case $T_1' = \mathsf{bool}$: It is found from $\mathsf{bool} \parallel T_2'$ that $T_2' = \mathsf{bool}$ since T_2' is not a refinement type. We then finish by (E_RED)/(R_BASE).

Case $T_1' = y{:}T_{11} \to T_{12}$: It is found that from $y{:}T_{11} \to T_{12} \parallel T_2'$ that $T_2' = y{:}T_{21} \to T_{22}$ for some T_{21} and T_{22} since T_2' is not a refinement type. We then finish by (E_RED)/(R_FUN).

Case $T_1' = y{:}T_{11} \times T_{12}$: It is found that from $y{:}T_{11} \times T_{12} \parallel T_2'$ that $T_2' = y{:}T_{21} \times T_{22}$ for some T_{21} and T_{22} since T_2' is not a refinement type. By Lemmas C.3.4 and C.3.5 (1), $T_1 = y{:}T_{11}' \times T_{12}'$ for some T_{11}' and T_{12}'. Since $\emptyset \vdash e_2 : T_1 = y{:}T_{11}' \times T_{12}'$ and e_2 is a value, we have $e_2 = (v_1, v_2)$ for some v_1 and v_2 by Lemma C.3.13 (3). We then finish by (E_RED)/(R_PROD).

Case $T_1' = \tau_1\langle e_1' \rangle$: It is found that from $\Sigma \vdash \tau_1\langle e_1' \rangle \parallel T_2'$ that $T_2' = \tau_2\langle e_2' \rangle$ for some τ_2 and e_2' since T_2' is not a refinement type. If $\tau_1 = \tau_2$ and τ_1 is monomorphic, then we apply (E_RED)/(R_DATATYPEMONO); if $\tau_1 \neq \tau_2$ or τ_1 is not monomorphic, and $\delta(\langle \tau_1\langle e_1' \rangle \Leftarrow \tau_2\langle e_2' \rangle\rangle^{\ell}\ e_2)$ is defined, then (E_RED)/(R_DATATYPE); otherwise, (E_RED)/(R_DATATYPEFAIL).

Case $T_1' = \{y{:}T_1'' \mid e_1''\}$: Since T_2' is not a refinement type, we finish by (E_RED)/(R_PRECHECK).

Case (T_PAIR): We are given $\emptyset \vdash (e_1, e_2) : x{:}T_1 \times T_2$ for some e_1, e_2, x, T_1 and T_2. By inversion, we have $\emptyset \vdash e_1 : T_1$ and $\emptyset \vdash e_2 : T_2\{e_1/x\}$. By the IH, e_1 and e_2 are reducible, values, or blaming. If e_1 is reducible or a blaming, then we finish by one of evaluation rules. If e_1 is a value and e_2 is reducible or a blaming, then we finish by one of evaluation rules. Otherwise, if e_1 and e_2 are values, then so is (e_1, e_2) is.

Case (T_PROJ1): We are given $\emptyset \vdash e_1.1 : T_1$ for some e_1 and T_1. By inversion, we have $\emptyset \vdash e_1 : x{:}T_1 \times T_2$ for some x and T_2. By the IH, e_1 is reducible, a value, or a blaming. If e_1 is reducible or a blaming, then we finish by one of evaluation rules. Otherwise, if e_1 is a value, then $e_1 = (v_1, v_2)$ for some v_1 and v_2 by Lemma C.3.13 (3), and so we finish by (E_RED)/(R_PROJ1).

Case (T_PROJ2): Similarly to the case for (T_PROJ1). We are given $\emptyset \vdash e_2.2 : T_2\{e_2.1/x\}$ for some e_2, T_2, and x. By inversion, we have $\emptyset \vdash e_2 : x{:}T_1 \times T_2$ for some T_1. By the IH, e_2 is reducible, a value, or a blaming. If e_2 is reducible or a blaming, then we finish by one of evaluation rules. Otherwise, if e_2 is a value, then $e_2 = (v_1, v_2)$ for some v_1 and v_2 by Lemma C.3.13 (3), and so we finish by (E_RED)/(R_PROJ2).

Case (T_CTR): We are given $\emptyset \vdash C\langle e_1 \rangle e_2 : \tau\langle e_1 \rangle$. By inversion, we have $\emptyset \vdash e_2 : T'\{e_1/x\}$ for some T' and x such that $\mathit{TypSpecOf}(C) = x{:}T'' \rightarrowtail T' \rightarrowtail \tau\langle x \rangle$. By the IH, e_2 is reducible, a value, or a blaming. If e_2 is reducible or a blaming, then we finish by one of evaluation rules. Otherwise, if e_2 is a value, then so is $C\langle e_1 \rangle e_2$.

Case (T_MATCH): We are given $\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i \in \{1,\ldots,n\}} : T$ for some e_0 and $\overline{C_i\ x_i \to e_i}^{\,i \in \{1,\ldots,n\}}$. By inversion, we have $\Gamma \vdash e_0 : \tau\langle e' \rangle$ for some τ and e'. By the IH, e_0 is reducible, a value, or a blaming. If e_0 is reducible or a blaming, then we finish by one of evaluation rules. Otherwise, if e_0 is a value, then, by Lemma C.3.13 (4), we have $e_0 = C\langle e_1' \rangle v_2$ for some C, e_1' and v_2. By Lemmas C.3.12 and C.3.6, C is a constructor of τ. There therefore exists $j \in$

$\{1, ..., n\}$ such that $C = C_j$ since patterns are exhaustive. By (R_MATCH), we finish.

Case (T_IF): We are given $\emptyset \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ for some e_1, e_2 and e_3. By inversion, we have $\emptyset \vdash e_1 : \mathsf{bool}$. By the IH, e_1 is reducible, a value, or a blaming. If e_1 is reducible or a blaming, then we finish by one of evaluation rules. Otherwise, if e_1 is a value, then e_1 is true or false by Lemma C.3.13 (1). If e_1 is true (resp. false), then we finish by (R_IFTRUE) (resp. (R_IFFALSE)).

Case (T_WCHECK): We are given $\emptyset \vdash \langle\!\langle \{x{:}T' \mid e_1\}, e_2 \rangle\!\rangle^{\ell} : \{x{:}T' \mid e_1\}$ for some x, T', e_1, e_2 and ℓ. By inversion, we have $\emptyset \vdash e_2 : T'$. By the IH, e_2 is reducible, a value, or a blaming. If e_2 is reducible or a blaming, then we finish by one of evaluation rules. Otherwise, if e_2 is a value, we finish by (R_CHECK).

Case (T_ACHECK): We are given $\emptyset \vdash \langle \{x{:}T' \mid e_1\}, e_2, v \rangle^{\ell} : \{x{:}T' \mid e_1\}$ for some x, T', e_1, e_2, v and ℓ. By inversion, we have $\emptyset \vdash e_2 : \mathsf{bool}$. By the IH, e_2 is reducible, a value, or a blaming. If e_2 is reducible or a blaming, then we finish by one of evaluation rules. Otherwise, if e_2 is a value, then e_2 is true or false by Lemma C.3.13 (1). If e_2 is true (resp. false), then we finish by (R_OK) (resp. (R_FAIL)).

Case (T_CONV): By inversion, we have $\emptyset \vdash e : T'$. By the IH, we finish.

Lemma C.3.14 (Context and Type Well-Formedness)**.**

1. *If* $\Gamma \vdash e : T$, *then* $\vdash \Gamma$ *and* $\Gamma \vdash T$.
2. *If* $\Gamma \vdash T$, *then* $\vdash \Gamma$.

Proof. By induction on the derivation of each judgment.

1. By case analysis on the typing derivation.

   Case (T_CONST): We are given $\Gamma \vdash c : T$ for some c. By inversion, we have $\vdash \Gamma$ and $T = \mathsf{bool}$. By (WT_BASE), $\Gamma \vdash \mathsf{bool}$.

   Case (T_VAR): We are given $\Gamma \vdash x : T$ for some x. By inversion, we have $\vdash \Gamma$ and $x{:}T \in \Gamma$. Let Γ_1 and Γ_2 be typing contexts such that $\Gamma_1, x{:}T, \Gamma_2 = \Gamma$. By inversion of $\vdash \Gamma$, we have $\Gamma_1 \vdash T$. Since for any $y{:}T' \in \Gamma_2$, $\Gamma_1, x{:}T, \Gamma_2' \vdash T'$ where $\Gamma_2 = \Gamma_2', y{:}T', \Gamma_2''$ for some Γ_2'', we have $\Gamma_1, x{:}T, \Gamma_2 \vdash T$ by Lemma C.3.1.

   Case (T_BLAME): We are given $\Gamma \vdash \Uparrow \ell : T$ for some ℓ. By inversion, we have $\vdash \Gamma$ and $\emptyset \vdash T$. By Lemma C.3.1, $\Gamma \vdash T$.

   Case (T_ABS): We are given $\Gamma \vdash \mathtt{fix}\, f(x{:}T_1){:}T_2 = e_2 : x{:}T_1 \to T_2$ for some f, x, T_1, T_2 and e_2. By inversion, we have $\Gamma, f{:}(x{:}T_1 \to T_2), x{:}T_1 \vdash e_2 : T_2$. By the IH, we have $\vdash \Gamma, f{:}(x{:}T_1 \to T_2), x{:}T_1$. By inversion of it, $\vdash \Gamma$ and $\Gamma \vdash x : T_1 \to T_2$.

   Case (T_CAST): We are given $\Gamma \vdash \langle T_1 \Leftarrow T_2 \rangle^{\ell} : x{:}T_2 \to T_1$ for some T_1, T_2, ℓ and x. Without loss of generality, we can suppose that x is fresh. By inversion, we have $\Gamma \vdash T_1$ and $\Gamma \vdash T_2$. By the IH, we have $\vdash \Gamma$. By Lemma C.3.1, $\Gamma, x{:}T_2 \vdash T_1$. By (WT_FUN), we have $\Gamma \vdash x : T_2 \to T_1$.

   Case (T_APP): We are given $\Gamma \vdash e_1\, e_2 : T_2\{e_2/x\}$ for some T_2, e_2 and x. By inversion, we have $\Gamma \vdash e_1 : x{:}T_1 \to T_2$ and $\Gamma \vdash e_2 : T_1$. By the IH, we have $\vdash \Gamma$ and $\Gamma \vdash x : T_1 \to T_2$. By inversion of the latter, we have $\Gamma, x{:}T_1 \vdash T_2$. By Lemma C.3.2, we have $\Gamma \vdash T_2\{e_2/x\}$.

Case (T_PAIR): We are given $\Gamma \vdash (e_1, e_2) : x{:}T_1 \times T_2$ for some e_1, e_2, x, T_1 and T_2. By inversion, we have $\Gamma, x{:}T_1 \vdash T_2$. By the IH, $\vdash \Gamma, x{:}T_1$. By inversion of it, we have $\vdash \Gamma$. Since $\Gamma, x{:}T_1 \vdash T_2$, we finish by (WT_PROD).

Case (T_PROJ1): We are given $\Gamma \vdash e'.1 : T$ for some e'. By inversion, we have $\Gamma \vdash e' : x{:}T \times T'$ for some x and T'. By the IH, we have $\vdash \Gamma$ and $\Gamma \vdash x{:}T \times T'$. By inversion of the latter, we have $\Gamma \vdash T$.

Case (T_PROJ2): we are given $\Gamma \vdash e'.2 : T_2\{e'.1/x\}$ for some e', T_2 and x. By inversion, we have $\Gamma \vdash e' : x{:}T_1 \times T_2$ for some T_1. By the IH, $\vdash \Gamma$ and $\Gamma \vdash x{:}T_1 \times T_2$. By inversion of the latter, we have $\Gamma, x{:}T_1 \vdash T_2$. Since $\Gamma \vdash e' : x{:}T_1 \times T_2$, we have $\Gamma \vdash e'.1 : T_1$ by (T_PROJ1). By Lemma C.3.2, we have $\Gamma \vdash T_2\{e'.1/x\}$.

Case (T_CTR): We are given $\Gamma \vdash C\langle e_1\rangle e_2 : \tau\langle e_1\rangle$ for some C, e_1, e_2 and τ. By inversion, we have $\Gamma \vdash \tau\langle e_1\rangle$. By the IH, we have $\vdash \Gamma$.

Case (T_MATCH): We are given $\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i} : T$ for some e_0 and $\overline{C_i\ x_i \to e_i}^{\,i}$. By inversion, we have $\Gamma \vdash T$. By the IH, we have $\vdash \Gamma$.

Case (T_IF): We are given $\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ for some e_1, e_2 and e_3. By inversion, we have $\Gamma \vdash e_2 : T$. By the IH, we have $\vdash \Gamma$ and $\Gamma \vdash T$.

Case (T_WCHECK): We are given $\Gamma \vdash \langle\!\langle \{x{:}T_1 \mid e_1\}, e_2 \rangle\!\rangle^{\ell} : \{x{:}T_1 \mid e_1\}$ for some x, T_1, e_1, e_2 and ℓ. By inversion, we have $\vdash \Gamma$ and $\emptyset \vdash \{x{:}T_1 \mid e_1\}$. By Lemma C.3.1, we have $\Gamma \vdash \{x{:}T_1 \mid e_1\}$.

Case (T_ACHECK): We are given $\Gamma \vdash \langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^{\ell} : \{x{:}T_1 \mid e_1\}$ for some x, T_1, e_1, e_2, v and ℓ. By inversion, we have $\vdash \Gamma$ and $\emptyset \vdash \{x{:}T_1 \mid e_1\}$. By Lemma C.3.1, we have $\Gamma \vdash \{x{:}T_1 \mid e_1\}$.

Case (T_CONV): By inversion, we have $\vdash \Gamma$ and $\emptyset \vdash T$. By Lemma C.3.1, we have $\Gamma \vdash T$.

Case (T_FORGET): We are given $\Gamma \vdash v : T$ for some v. By inversion, we have $\vdash \Gamma$ and $\emptyset \vdash v : \{x{:}T \mid e'\}$ for some x and e'. By the IH, $\emptyset \vdash \{x{:}T \mid e'\}$. By inversion of it, we have $\emptyset \vdash T$. By Lemma C.3.1, $\Gamma \vdash T$.

Case (T_EXACT): We are given $\Gamma \vdash v : \{x{:}T' \mid e'\}$ for some v, x, T' and e'. By inversion, we have $\vdash \Gamma$ and $\emptyset \vdash \{x{:}T' \mid e'\}$. By Lemma C.3.1, we finish.

2. By case analysis on the well-formedness derivation.

Case (WT_BASE): We are given $\Gamma \vdash \mathsf{bool}$ for some bool. By inversion, we have $\vdash \Gamma$.

Case (WT_FUN): We are given $\Gamma \vdash x : T_1 \to T_2$ for some x, T_1 and T_2. By inversion, we have $\Gamma \vdash T_1$. By the IH, $\vdash \Gamma$.

Case (WT_REFINE): We are given $\Gamma \vdash \{x{:}T' \mid e'\}$ for some x, T' and e'. By inversion, we have $\Gamma \vdash T'$. By the IH, $\vdash \Gamma$.

Case (WT_PROD): We are given $\Gamma \vdash x{:}T_1 \times T_2$ for some x, T_1 and T_2. By inversion, we have $\Gamma \vdash T_1$. By the IH, $\vdash \Gamma$.

Case (WT_DATATYPE): We are given $\Gamma \vdash \tau\langle e\rangle$ for some τ and e. By inversion and the IH, we finish. □

Lemma C.3.15. *If* $T_1 \parallel \{x{:}T_2 \mid e_2\}$, *then* $T_1 \parallel T_2$.

Proof. By induction on $T_1 \parallel \{x{:}T_2 \mid e_2\}$. There are only two cases where $T_1 \parallel \{x{:}T_2 \mid e_2\}$ can be derived.

Case $\{x{:}T_1' \mid e_1'\} \parallel \{x{:}T_2 \mid e_2\}$: By inversion, we have $T_1' \parallel T_2$. By (C_REFINEL), $\{x{:}T_1' \mid e_1'\} \parallel T_2$.

Case (C_REFINEL): We are given $\{y{:}T_1' \mid e_1'\} \parallel \{x{:}T_2 \mid e_2\}$. By inversion, we have $T_1' \parallel \{x{:}T_2 \mid e_2\}$. By the IH, we have $T_1' \parallel T_2$. By (C_REFINEL), we finish. □

Lemma C.3.16. *If $T_1 \parallel T_2$, then $T_1 \parallel T_2\{e/x\}$ for any e and x.*

Proof. Straightforward by induction on $T_1 \parallel T_2$. □

Lemma 28 (Preservation). *Suppose that $\emptyset \vdash e : T$.*

(1) If $e \rightsquigarrow e'$, then $\emptyset \vdash e' : T$.

(2) If $e \longrightarrow e'$, then $\emptyset \vdash e' : T$.

Proof.

1. By induction on the typing derivation.

   Case (T_CONST), (T_VAR), (T_BLAME), (T_ABS), (T_CAST), (T_PAIR), (T_CTR), (T_FORGET) or (T_EXACT): Trivial because e does not step in the reduction relation.

   Case (T_APP): We are given $\emptyset \vdash e_1\, e_2 : T_2\{e_2/x\}$ for some e_1, e_2, T_2 and x. Without loss of generality, we can suppose that x is fresh. By inversion, we have $\emptyset \vdash e_1 : x{:}T_1 \to T_2$ and $\emptyset \vdash e_2 : T_1$ for some T_1. By case analysis on the reduction rule applied.

   Case (R_BETA): We are given $(\texttt{fix}\ f(x{:}T_1'){:}T_2' = e_{12})\, v_2 \rightsquigarrow e_{12}\{v_2/x, \texttt{fix}\, f(x{:}T_1') = e_{12}/f\}$ for some f, T_1', T_2', e_{12} and v_2. Without loss of generality, we can suppose that f is fresh. By Lemma C.3.9, we have $f{:}(x{:}T_1' \to T_2'), x{:}T_1' \vdash e_{12} : T_2'$ and $f \notin \mathsf{FV}(T_2')$ and $x{:}T_1' \to T_2' \equiv x{:}T_1 \to T_2$ for some T_2'. Note that x (resp. f) does not occur in T_1' (resp. T_1' and T_2'). By Lemma C.3.14 and inversion, we have $\emptyset \vdash x{:}T_1' \to T_2'$, and thus $\emptyset \vdash T_1'$. Because $\emptyset \vdash e_1 : x{:}T_1' \to T_2'$ by Lemma C.1.1 (symmetry) and (T_CONV), we have $x{:}T_1' \vdash e_{12}\{e_1/f\} : T_2'$ by Lemma C.3.2. Since $T_1 \equiv T_1'$ by Lemma C.3.4, we have $\emptyset \vdash v_2 : T_1'$ by (T_CONV). By Lemma C.3.2, $\emptyset \vdash e_{12}\{e_1/f, v_2/x\} : T_2'\{v_2/x\}$ (note that e_1 is closed). Since $T_2 \equiv T_2'$ by Lemma C.3.4, we have $T_2\{v_2/x\} \equiv T_2'\{v_2/x\}$ by Lemma C.1.4 (3). Because $\emptyset \vdash T_2\{v_2/x\}$ by Lemma C.3.14, we have $\emptyset \vdash e_{12}\{e_1/f, v_2/x\} : T_2\{v_2/x\}$ by Lemma C.1.1 (symmetry) and (T_CONV).

   Case (R_BASE): We are given $\langle \mathsf{bool} \Leftarrow \mathsf{bool}\rangle^\ell\, v_2 \rightsquigarrow v_2$ for ℓ and v_2. By Lemmas C.3.10, C.3.4 and C.3.3, we have $T_1 = T_2 = \mathsf{bool}$. Since $T_2\{e_2/x\} = \mathsf{bool}$ and so $\emptyset \vdash v_2 : \mathsf{bool}$, we finish.

   Case (R_FUN): We are given

$$\langle y{:}T_{11} \to T_{12} \Leftarrow y{:}T_{21} \to T_{22}\rangle^\ell\, v_2 \rightsquigarrow \\ \lambda\, y{:}T_{11}.(\lambda\, z{:}T_{21}.\langle T_{12} \Leftarrow T_{22}\{z/y\}\rangle^\ell\, (v_2\, z))\, (\langle T_{21} \Leftarrow T_{11}\rangle^\ell\, y)$$

   for some y, T_{11}, T_{12}, T_{21}, T_{22}, ℓ, v_2 and z such that z is fresh. By Lemma C.3.10, we have $\emptyset \vdash y{:}T_{11} \to T_{12}$, $\emptyset \vdash y{:}T_{21} \to T_{22}$, $y{:}T_{11} \to T_{12} \parallel y{:}T_{21} \to T_{22}$ and $x{:}(y{:}T_{21} \to T_{22}) \to (y{:}T_{11} \to T_{12}) \equiv x{:}T_1 \to$

T_2. Note that x does not occur in $y{:}T_{11} \to T_{12}$. By inversion of derivations, $\emptyset \vdash T_{11}$, $\emptyset \vdash T_{21}$, $y{:}T_{11} \vdash T_{12}$, $y{:}T_{21} \vdash T_{22}$, $T_{11} \parallel T_{21}$, and $T_{12} \parallel T_{22}$.
Since $T_{21} \parallel T_{11}$ by symmetry of the compatibility relation, we have $\emptyset \vdash \langle T_{21} \Leftarrow T_{11} \rangle^{\ell} : T_{11} \to T_{21}$ by (T_CAST). Since $\emptyset \vdash T_{11}$, we have $y{:}T_{11} \vdash \langle T_{21} \Leftarrow T_{11} \rangle^{\ell} : T_{11} \to T_{21}$ by Lemma C.3.1. Since $y{:}T_{11} \vdash y : T_{11}$ by (T_VAR), we have $y{:}T_{11} \vdash \langle T_{21} \Leftarrow T_{11} \rangle^{\ell}\, y : T_{21}$ by (T_APP).
By Lemma C.3.4, $y{:}T_{21} \to T_{22} \equiv T_1$ and $y{:}T_{11} \to T_{12} \equiv T_2$, and thus, by Lemma C.3.4 (1), $T_1 = y{:}T'_{21} \to T'_{22}$ and $T_2 = y{:}T'_{11} \to T'_{12}$ for some T'_{21}, T'_{22}, T'_{11} and T'_{12}. Since $\emptyset \vdash v_2 : y{:}T'_{21} \to T'_{22}$ and $\emptyset \vdash y{:}T_{21} \to T_{22}$, we have $\emptyset \vdash v_2 : y{:}T_{21} \to T_{22}$ by Lemma C.1.1 (symmetry) and (T_CONV). We have $z{:}T_{21} \vdash v_2 : y{:}T_{21} \to T_{22}$ by Lemma C.3.1, and thus $z{:}T_{21} \vdash v_2\, z : T_{22}\{z/y\}$ by (T_VAR) and (T_APP).
Since $y{:}T_{21} \vdash T_{22}$, we have $z{:}T_{21}, y{:}T_{21} \vdash T_{22}$ and thus $y{:}T_{11}, z{:}T_{21} \vdash T_{22}\{z/y\}$ by Lemmas C.3.2 and C.3.1. Since $y{:}T_{11}, z{:}T_{21} \vdash T_{12}$ by Lemma C.3.1, and $T_{12} \parallel T_{22}\{z/y\}$ by Lemma C.3.16, we have $y{:}T_{11}, z{:}T_{21} \vdash \langle T_{12} \Leftarrow T_{22}\{z/y\} \rangle^{\ell} : T_{22}\{z/y\} \to T_{12}$ by (T_CAST).
By Lemma C.3.1 and (T_APP), $y{:}T_{11}, z{:}T_{21} \vdash \langle T_{12} \Leftarrow T_{22}\{z/y\} \rangle^{\ell}\, (v_2\, z) : T_{12}$. By Lemma C.3.1 and (T_ABS), we have $y{:}T_{11} \vdash \lambda\, z{:}T_{21}.\langle T_{12} \Leftarrow T_{22}\{z/y\} \rangle^{\ell}\, (v_2\, z) : T_{21} \to T_{12}$. (Note that z does not occur T_{12}.) Since $y{:}T_{11} \vdash \langle T_{21} \Leftarrow T_{11} \rangle^{\ell}\, y : T_{21}$, by (T_APP) we have $y{:}T_{11} \vdash (\lambda\, z{:}T_{21}.\langle T_{12} \Leftarrow T_{22}\{z/y\} \rangle^{\ell}\, (v_2\, z))\, (\langle T_{21} \Leftarrow T_{11} \rangle^{\ell}\, y) : T_{12}$. By Lemma C.3.1 and (T_ABS), $\emptyset \vdash \lambda\, y{:}T_{11}.(\lambda\, z{:}T_{21}.\langle T_{12} \Leftarrow T_{22}\{z/y\} \rangle^{\ell}\, (v_2\, z))\, (\langle T_{21} \Leftarrow T_{11} \rangle^{\ell}\, y) : (y{:}T_{11} \to T_{12})$.
Since $y{:}T_{11} \to T_{12} \equiv T_2$, we have $(y{:}T_{11} \to T_{12})\{v_2/x\} \equiv T_2\{v_2/x\}$ by Lemma C.1.4 (3). Since $(y{:}T_{11} \to T_{12})\{v_2/x\} = y{:}T_{11} \to T_{12}$ and $\emptyset \vdash T_2\{v_2/x\}$ by Lemma C.3.14, we finish by (T_CONV).

Case (R_PROD): Similarly to the case for (R_FUN). We are given

$$\langle y{:}T_{11} \times T_{12} \Leftarrow y{:}T_{21} \times T_{22} \rangle^{\ell}\, (v_1, v_2) \rightsquigarrow (\lambda\, y{:}T_{11}.(y, \langle T_{12} \Leftarrow T_{22}\{v_1/y\} \rangle^{\ell}\, v_2))\, (\langle T_{11} \Leftarrow T_{21} \rangle^{\ell}\, v_1)$$

for some y, T_{11}, T_{12}, T_{21}, T_{22}, ℓ, v_1 and v_2. Without loss of generality, we can suppose that y is fresh. By Lemma C.3.10, we have $\emptyset \vdash y{:}T_{11} \times T_{12}$ and $\emptyset \vdash y{:}T_{21} \times T_{22}$ and $y{:}T_{11} \times T_{12} \parallel y{:}T_{21} \times T_{22}$ and $x{:}(y{:}T_{21} \times T_{22}) \to y{:}T_{11} \times T_{12} \equiv x{:}T_1 \to T_2$. Note that x does not occur in $y{:}T_{11} \times T_{12}$. By inversion of derivations, $\emptyset \vdash T_{11}$ and $\emptyset \vdash T_{21}$ and $y{:}T_{11} \vdash T_{12}$ and $y{:}T_{21} \vdash T_{22}$ and $T_{11} \parallel T_{21}$ and $T_{12} \parallel T_{22}$.
By Lemma C.3.11, we have $\emptyset \vdash v_1 : T'_{21}$ and $\emptyset \vdash v_2 : T'_{22}\{v_1/y\}$ and $y{:}T'_{21} \vdash T'_{22}$ and $y{:}T'_{21} \times T'_{22} \equiv \mathit{unref}(T_1)$ for some T'_{21} and T'_{22}. Since $y{:}T_{21} \times T_{22} \equiv T_1$ by Lemma C.3.4, we have $y{:}T_{21} \times T_{22} \equiv y{:}T'_{21} \times T'_{22}$, and thus $T_{21} \equiv T'_{21}$ and $T_{22} \equiv T'_{22}$ by Lemma C.3.5. Since $\emptyset \vdash T_{21}$, we have $\emptyset \vdash v_1 : T_{21}$ by Lemma C.1.1 (symmetry) and (T_CONV). Therefore, we have $\emptyset \vdash \langle T_{11} \Leftarrow T_{21} \rangle^{\ell}\, v_1 : T_{11}$ by (T_CAST) and (T_APP).
Since $y{:}T_{21} \vdash T_{22}$ and $\emptyset \vdash v_1 : T_{21}$, we have $y{:}T_{11} \vdash T_{22}\{v_1/y\}$ by Lemmas C.3.2 and C.3.1. By Lemma C.3.16, $T_{12} \parallel T_{22}\{v_1/y\}$. By (T_CAST), we have $y{:}T_{11} \vdash \langle T_{12} \Leftarrow T_{22}\{v_1/y\} \rangle^{\ell} : T_{22}\{v_1/y\} \to T_{12}$.
Since $T_{22} \equiv T'_{22}$, we have $T_{22}\{v_1/y\} \equiv T'_{22}\{v_1/y\}$ by Lemma C.1.4 (3). Since $\emptyset \vdash T_{22}\{v_1/y\}$ by Lemma C.3.2, we have $\emptyset \vdash v_2 : T_{22}\{v_1/y\}$ by

Lemma C.1.1 (symmetry) and (T_CONV). By Lemma C.3.1 and (T_APP), $y{:}T_{11} \vdash \langle T_{12} \Leftarrow T_{22}\{v_1/y\}\rangle^{\ell} v_2 : T_{12}$.
Let z be a fresh variable. Since $z{:}T_{11}, y{:}T_{11} \vdash \langle T_{12} \Leftarrow T_{22}\{v_1/y\}\rangle^{\ell} v_2 : T_{12}$ by Lemma C.3.1, we have $z{:}T_{11} \vdash (\langle T_{12} \Leftarrow T_{22}\{v_1/y\}\rangle^{\ell} v_2)\{z/y\} : T_{12}\{z/y\}$ by Lemma C.3.2. Since $z{:}T_{11} \vdash z : T_{11}$ by (T_VAR), and $z{:}T_{11}, y{:}T_{11} \vdash T_{12}$ by Lemmas C.3.1 and C.3.2, we have $z{:}T_{11} \vdash (z, (\langle T_{12} \Leftarrow T_{22}\{v_1/y\}\rangle^{\ell} v_2)\{z/y\}) : y{:}T_{11} \times T_{12}$ by Lemma C.3.1 and (T_PAIR). By Lemmas C.3.1 and C.3.2, $y{:}T_{11} \vdash (z, (\langle T_{12} \Leftarrow T_{22}\{v_1/y\}\rangle^{\ell} v_2)\{z/y\})\{y/z\} : (y{:}T_{11} \times T_{12})\{y/z\}$, that is,

$$y{:}T_{11} \vdash (y, (\langle T_{12} \Leftarrow T_{22}\{v_1/y\}\rangle^{\ell} v_2)) : (y{:}T_{11} \times T_{12}).$$

By Lemma C.3.1 and (T_ABS), $\emptyset \vdash \lambda\, y{:}T_{11}.(y, \langle T_{12} \Leftarrow T_{22}\{v_1/y\}\rangle^{\ell} v_2) : T_{11} \to y{:}T_{11} \times T_{12}$. By (T_APP), $\emptyset \vdash (\lambda\, y{:}T_{11}.(y, \langle T_{12} \Leftarrow T_{22}\{v_1/y\}\rangle^{\ell} v_2))\,(\langle T_{11} \Leftarrow T_{21}\rangle^{\ell} v_1) : y{:}T_{11} \times T_{12}$.
Since $y{:}T_{11} \times T_{12} \equiv T_2$ by Lemma C.3.5, we have $(y{:}T_{11} \times T_{12})\{v_2/x\} \equiv T_2\{v_2/x\}$ by Lemma C.1.4 (3). Since $(y{:}T_{11} \times T_{12})\{v_2/x\} = y{:}T_{11} \times T_{12}$ and $\emptyset \vdash T_2\{v_2/x\}$ by Lemma C.3.14, we finish by (T_CONV).

Case (R_FORGET): We are given $\langle T_1' \Leftarrow \{y{:}T_2' \mid e_2'\}\rangle^{\ell} v_2 \rightsquigarrow \langle T_1' \Leftarrow T_2'\rangle^{\ell} v_2$ for some T_1', y, T_2', e_2' and v_2. Without loss of generality, we can suppose that y is fresh. By Lemma C.3.10, we have $\emptyset \vdash T_1'$ and $\emptyset \vdash \{y{:}T_2' \mid e_2'\}$ and $T_1' \parallel \{y{:}T_2' \mid e_2'\}$ and $x{:}\{y{:}T_2' \mid e_2'\} \to T_1' \equiv x{:}T_1 \to T_2$. Note that x does not occur in T_1'. By inversion and Lemma C.3.15, $\emptyset \vdash T_2'$ and $T_1' \parallel T_2'$.
By (T_CAST), we have $\emptyset \vdash \langle T_1' \Leftarrow T_2'\rangle^{\ell} : T_2' \to T_1'$. Since $\{y{:}T_2' \mid e_2'\} \equiv T_1$ by Lemma C.3.4, we have $\emptyset \vdash v_2 : \{y{:}T_2' \mid e_2'\}$ by Lemma C.1.1 (symmetry) and (T_CONV). By (T_FORGET), $\emptyset \vdash v_2 : T_2'$. Thus, $\emptyset \vdash \langle T_1' \Leftarrow T_2'\rangle^{\ell} v_2 : T_1'$. Since $T_1' \equiv T_2$ by Lemma C.3.4, $T_1'\{v_2/x\} \equiv T_2\{v_2/x\}$ by Lemma C.1.4 (3). Since $T_1'\{v_2/x\} = T_1'$, we have $\emptyset \vdash \langle T_1' \Leftarrow T_2'\rangle^{\ell} v_2 : T_2\{v_2/x\}$ by Lemma C.3.14 and (T_CONV).

Case (R_PRECHECK): We are given $\langle \{y{:}T_1' \mid e_1'\} \Leftarrow T_2'\rangle^{\ell} v_2 \rightsquigarrow \langle\!\langle \{y{:}T_1' \mid e_1'\}, \langle T_1' \Leftarrow T_2'\rangle^{\ell} v_2\rangle\!\rangle^{\ell}$ for some y, T_1', e_1', T_2', ℓ and v_2. Without loss of generality, we can suppose that y is fresh. By Lemma C.3.10, we have $\emptyset \vdash \{y{:}T_1' \mid e_1'\}$ and $\emptyset \vdash T_2'$ and $\{y{:}T_1' \mid e_1'\} \parallel T_2'$ and $x{:}T_2' \to \{y{:}T_1' \mid e_1'\} \equiv x{:}T_1 \to T_2$. Note that x does not occur in $\{y{:}T_1' \mid e_1'\}$. By inversion and Lemma C.3.15, $\emptyset \vdash T_1'$ and $T_1' \parallel T_2'$.
By (T_CAST), we have $\emptyset \vdash \langle T_1' \Leftarrow T_2'\rangle^{\ell} : T_2' \to T_1'$. Since $T_2' \equiv T_1$ by Lemma C.3.4, we have $\emptyset \vdash v_2 : T_2'$ by (T_CONV). Thus, by (T_APP), $\emptyset \vdash \langle T_1' \Leftarrow T_2'\rangle^{\ell} v_2 : T_1'$. By (T_WCHECK), $\emptyset \vdash \langle\!\langle \{y{:}T_1' \mid e_1'\}, \langle T_1' \Leftarrow T_2'\rangle^{\ell} v_2\rangle\!\rangle^{\ell} : \{y{:}T_1' \mid e_1'\}$. Since $\{y{:}T_1' \mid e_1'\} \equiv T_2$ by Lemma C.3.4, we have $\{y{:}T_1' \mid e_1'\}\{v_2/x\} \equiv T_2\{v_2/x\}$. Since $\{y{:}T_1' \mid e_1'\}\{v_2/x\} = \{y{:}T_1' \mid e_1'\}$, we have $\emptyset \vdash \langle\!\langle \{y{:}T_1' \mid e_1'\}, \langle T_1' \Leftarrow T_2'\rangle^{\ell} v_2\rangle\!\rangle^{\ell} : T_2\{v_2/x\}$ by Lemma C.3.14 and (T_CONV).

Case (R_DATATYPE): We are given

$$\langle \tau_1\langle e_1'\rangle \Leftarrow \tau_2\langle e_2'\rangle\rangle^{\ell}\, C_2\langle e'\rangle v \rightsquigarrow C_1\langle e_1'\rangle(\langle T_1''\{e_1'/y_1\} \Leftarrow T_2''\{e_2'/y_2\}\rangle^{\ell} v)$$

for some τ_1, e_1', τ_2, e_2', ℓ, C_2, e', v, C_1, T_1'', y_1, T_2'', and y_2 such that $\tau_1 \neq \tau_2$

or τ_1 is not monomorphic, and $C_1 = \delta(\langle \tau_1\langle e_1'\rangle \Leftarrow \tau_2\langle e_2'\rangle\rangle^\ell\ C_2\langle e'\rangle v)$ and, for $i \in \{1,2\}$, $\mathit{ArgTypeOf}(\tau_i) = y_i{:}T_i'$ and $\mathit{CtrArgOf}(C_i) = T_i''$.
Since the constructor choice function δ is well-formed, we find that $C_1 \in \mathit{CompatCtrsOf}(\tau_1, C_2)$, that is, $C_1 \in \mathit{CtrsOf}(\tau_1)$ and $T_1'' \parallel T_2''$ from well-formedness of the type definition environment. Also, $y_1{:}T_1' \vdash T_1''$ and $y_2{:}T_2' \vdash T_2''$ from well-formedness of the type definition environment.
By Lemma C.3.16, $T_1''\{e_1'/y_1\} \parallel T_2''\{e_2'/y_2\}$. By Lemma C.3.10, we have $\emptyset \vdash \tau_1\langle e_1'\rangle$ and $\emptyset \vdash \tau_2\langle e_2'\rangle$ and $x{:}\tau_2\langle e_2'\rangle \to \tau_1\langle e_1'\rangle \equiv x{:}T_1 \to T_2$. Note that x does not occur in $\tau_1\langle e_1'\rangle$. By inversion of derivations, and Lemma C.3.2, we have $\emptyset \vdash T_1''\{e_1'/y_1\}$ and $\emptyset \vdash T_2''\{e_2'/y_2\}$. Thus by (T_CAST), $\emptyset \vdash \langle T_1''\{e_1'/y_1\} \Leftarrow T_2''\{e_2'/y_2\}\rangle^\ell : T_2''\{e_2'/y_2\} \to T_1''\{e_1'/y_1\}$. By Lemma C.3.12, $\emptyset \vdash v : T_2''\{e'/y_2\}$ and $\tau_2\langle e'\rangle \equiv \mathit{unref}(T_1)$. Since $\tau_2\langle e_2'\rangle \equiv \mathit{unref}(T_1)$ by Lemmas C.3.4 and C.3.8, we have $\tau_2\langle e'\rangle \equiv \tau_2\langle e_2'\rangle$ by Lemma C.3.4 and Lemma C.1.1 (transitivity). Thus, $e' \equiv e_2'$ by Lemma C.3.6. Since $T_2''\{e'/y_2\} \equiv T_2''\{e_2'/y_2\}$ by Lemma C.1.3 (3), we have $\emptyset \vdash v : T_2''\{e_2'/y_2\}$ by (T_CONV). By (T_APP), we have $\emptyset \vdash \langle T_1''\{e_1'/y_1\} \Leftarrow T_2''\{e_2'/y_2\}\rangle^\ell\ v : T_1''\{e_1'/y_1\}$. By inversion of $\emptyset \vdash \tau_1\langle e_1'\rangle$, we have $\emptyset \vdash e_1' : T_1'$. Thus, by (T_CTR), $\emptyset \vdash C_1\langle e_1'\rangle(\langle T_1''\{e_1'/y_1\} \Leftarrow T_2''\{e_2'/y_2\}\rangle^\ell\ v) : \tau_1\langle e_1'\rangle$.
By Lemma C.3.4, we have $\tau_1\langle e_1'\rangle \equiv T_2$. Since $\tau_1\langle e_1'\rangle\{C_2\langle e'\rangle v/x\} = \tau_1\langle e_1'\rangle$, we have $\tau_1\langle e_1\rangle \equiv T_2\{C_2\langle e'\rangle v/x\}$ by Lemma C.1.4 (3). By Lemma C.3.14 and (T_CONV), we finish.

Case (R_DATATYPEMONO): We are given $\langle \tau \Leftarrow \tau\rangle^\ell\ v_2 \rightsquigarrow v_2$ for some τ, ℓ and v_2. By Lemma C.3.10, $x{:}\tau \to \tau \equiv x{:}T_1 \to T_2$. Note that x does not occur in τ. By Lemma C.3.4, $\tau \equiv T_1$ and $\tau \equiv T_2$, and so $T_1 \equiv T_2$ by Lemma C.1.1. Since $T_1\{v_2/x\} = T_1$ by Lemma C.3.14, $T_1 \equiv T_2\{v_2/x\}$ by Lemma C.1.4 (3). Since $\emptyset \vdash v_2 : T_1$, we have $\emptyset \vdash v_2 : T_2\{v_2/x\}$ by Lemma C.3.14 and (T_CONV).

Case (R_DATATYPEFAIL): We are given $\langle \tau_1\langle e_1'\rangle \Leftarrow \tau_2\langle e_2'\rangle\rangle^\ell\ v_2 \rightsquigarrow \Uparrow\ell$ for some τ_1, e_1', τ_2, e_2', ℓ and v_2. By Lemma C.3.14 and (T_BLAME), we finish.

Case (T_PROJ1): We are given $\emptyset \vdash e_1.1 : T$ for some e_1. By inversion, we have $\emptyset \vdash e_1 : x{:}T \times T_2$ for some x and T_2. The term steps only by (R_PROJ1): $(v_1, v_2).1 \rightsquigarrow v_1$ for some v_1 and v_2 such that $e_1 = (v_1, v_2)$.
By Lemma C.3.11, we have $\emptyset \vdash v_1 : T_1'$ and $x{:}T_1' \times T_2' \equiv x{:}T \times T_2$ for some T_1' and T_2'. By Lemma C.3.5, we have $T_1' \equiv T$. Since $\emptyset \vdash T$ by Lemma C.3.14, we have $\emptyset \vdash v_1 : T$ by (T_CONV).

Case (T_PROJ2): We are given $\emptyset \vdash e_2.2 : T_2\{e_2.1/x\}$ for some e_2, T_2 and x. By inversion, we have $\emptyset \vdash e_2 : x{:}T_1 \times T_2$ for some T_1. The term steps only by (R_PROJ2): $(v_1, v_2).2 \rightsquigarrow v_2$ for some v_1 and v_2 such that $e_2 = (v_1, v_2)$.
By Lemma C.3.11, we have $\emptyset \vdash v_2 : T_2'\{v_1/x\}$ and $x{:}T_1' \times T_2' \equiv x{:}T_1 \times T_2$ for some T_1' and T_2'. Since $(v_1, v_2).1 \longrightarrow v_1$ by (E_RED)/(R_PROJ1), we have $T_2'\{(v_1, v_2).1/x\} \equiv T_2'\{v_1/x\}$ by Lemmas C.1.2 and C.1.3 (3). Since $T_2' \equiv T_2$ by Lemma C.3.5, we have $T_2'\{(v_1, v_2).1/x\} \equiv T_2\{(v_1, v_2).1/x\}$ by Lemma C.1.4 (3), and thus $T_2'\{v_1/x\} \equiv T_2\{(v_1, v_2).1/x\}$ by Lemma C.1.1 (symmetry and transitivity). Since $\emptyset \vdash T_2\{(v_1, v_2).1/x\}$ by Lemma C.3.14, we have $\emptyset \vdash v_2 : T_2\{(v_1, v_2).1/x\}$ by (T_CONV).

Case (T_MATCH): We are given $\emptyset \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i \in \{1,\ldots,n\}} : T$ for some e_0 and $\overline{C_i\ x_i \to e_i}^{\,i \in \{1,\ldots,n\}}$. By inversion, we have $\emptyset \vdash e_0 : \tau\langle e''\rangle$

and $\emptyset \vdash T$ and $\mathit{CtrsOf}(\tau) = \overline{C_i}^{\,i \in \{1,\ldots,n\}}$ and $\mathit{ArgTypeOf}(\tau) = y{:}T'$ and, for $i \in \{1,\ldots,n\}$, $\mathit{CtrArgOf}(C_i) = T_i$ and $x_i{:}T_i\,\{e''/y\} \vdash e_i : T$. The term steps only by (R_MATCH):

$$\mathsf{match}\ C_j\langle e'''\rangle v'\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i \in \{1,\ldots,n\}} \rightsquigarrow e_j\,\{v'/x_j\}$$

for some $j \in \{1,\ldots,n\}$, e''', v' such that $e_0 = C_j\langle e'''\rangle v'$.

By Lemma C.3.12, we have $\emptyset \vdash v' : T_j\,\{e'''/y\}$ and $\tau\langle e'''\rangle \equiv \tau\langle e''\rangle$. Since $e''' \equiv e''$ by Lemma C.3.6, we have $T_j\,\{e'''/y\} \equiv T_j\,\{e''/y\}$ by Lemma C.1.3 (3). Since $x_j{:}T_j\,\{e''/y\} \vdash e_j : T$, we have $\emptyset \vdash T_j\,\{e''/y\}$ by Lemma C.3.14 and inversion. Thus we have $\emptyset \vdash v' : T_j\,\{e''/y\}$ by (T_CONV). Since x_j does not occur in T, we have $\emptyset \vdash e_j\,\{v'/x_j\} : T$ by Lemma C.3.2.

Case (T_IF): We are given $\emptyset \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ for some e_1, e_2 and e_3. By inversion, we have $\emptyset \vdash e_2 : T$ and $\emptyset \vdash e_3 : T$. Only two reduction rules can be applied to the term: (R_IFTRUE) and (R_IFFALSE). The case of (R_IFTRUE) follows from $\emptyset \vdash e_2 : T$, and (R_IFFALSE) from $\emptyset \vdash e_3 : T$.

Case (T_WCHECK): We are given $\emptyset \vdash \langle\!\langle \{x{:}T_1 \mid e_1\}, e_2 \rangle\!\rangle^{\ell} : \{x{:}T_1 \mid e_1\}$ for some x, T_1, e_1, e_2 and ℓ. By inversion, we have $\emptyset \vdash \{x{:}T_1 \mid e_1\}$ and $\emptyset \vdash e_2 : T_1$. The term steps only by (R_CHECK): $\langle\!\langle \{x{:}T_1 \mid e_1\}, v_2 \rangle\!\rangle^{\ell} \rightsquigarrow \langle \{x{:}T_1 \mid e_1\}, e_1\,\{v_2/x\}, v_2 \rangle^{\ell}$ for some v_2 such that $e_2 = v_2$.

From $\emptyset \vdash \{x{:}T_1 \mid e_1\}$, we find that $x{:}T_1 \vdash e_1 : \mathsf{bool}$. By Lemma C.3.2, $\emptyset \vdash e_1\,\{v_2/x\} : \mathsf{bool}$. Because $e_1\,\{v_2/x\} \longrightarrow^* e_1\,\{v_2/x\}$, we finish.

Case (T_ACHECK): We are given $\emptyset \vdash \langle \{x{:}T_1 \mid e_1\}, e_2, v \rangle^{\ell} : \{x{:}T_1 \mid e_1\}$ for some x, T_1, e_1, e_2 and v. By inversion, we have $\emptyset \vdash \{x{:}T_1 \mid e_1\}$ and $\emptyset \vdash v : T_1$ and $e_1\,\{v/x\} \longrightarrow^* e_2$. Only two reduction rules can be applied to the term: (R_OK) and (R_FAIL). The case of (R_OK) follows from (T_EXACT), and (R_FAIL) from (T_BLAME).

Case (T_CONV): By inversion, we have $\emptyset \vdash e : T'$ and $T' \equiv T$ and $\emptyset \vdash T$ for some T'. If e steps to e', then we have $\emptyset \vdash e' : T'$ by the IH. By (T_CONV), we finish.

2. By induction on the typing derivation. If $e \longrightarrow \Uparrow\ell$ by (E_BLAME), then we finish by Lemma C.3.14 and (T_BLAME). In the following, we suppose that e steps by (E_RED).

Case (T_CONST), (T_VAR), (T_BLAME), (T_ABS), (T_CAST), (T_FORGET) or (T_EXACT): Trivial because e does not step in the evaluation relation.

Case (T_APP): We are given $\emptyset \vdash e_1\ e_2 : T_2\,\{e_2/x\}$ for some e_1, e_2, T_2 and x. By inversion, we have $\emptyset \vdash e_1 : x{:}T_1 \to T_2$ and $\emptyset \vdash e_2 : T_1$ for some T_1.

If e_1 is not a value, then $e_1 \longrightarrow e_1'$ for some e_1' (noting e_1 is not a blaming; if so, (E_BLAME) is applied to $e_1\ e_2$, but it is contradictory). By the IH, $\emptyset \vdash e_1' : x{:}T_1 \to T_2$ and thus $\emptyset \vdash e_1'\ e_2 : T_2\,\{e_2/x\}$ by (T_APP).

If e_1 is a value but e_2 is not, then $e_2 \longrightarrow e_2'$ for some e_2'. By the IH, $\emptyset \vdash e_2' : T_1$ and thus $\emptyset \vdash e_1\ e_2' : T_2\,\{e_2'/x\}$ by (T_APP). Because $T_2\,\{e_2'/x\} \equiv T_2\,\{e_2/x\}$ by Lemmas C.1.2, C.1.3 (3) and C.1.1, we have $\emptyset \vdash e_1\ e_2' : T_2\,\{e_2/x\}$ by Lemma C.3.14 and (T_CONV).

Otherwise, if e_1 and e_2 are values, then we finish by the case (1).

Case (T_PAIR): We are given $\emptyset \vdash (e_1, e_2) : x{:}T_1 \times T_2$ for some e_1, e_2, x, T_1 and T_2. By inversion, we have $\emptyset \vdash e_1 : T_1$ and $\emptyset \vdash e_2 : T_2\{e_1/x\}$ and $x{:}T_1 \vdash T_2$.
If e_1 is not a value, then $e_1 \longrightarrow e_1'$ for some e_1'. By the IH, $\emptyset \vdash e_1' : T_1$ and thus $\emptyset \vdash T_2\{e_1'/x\}$ by Lemma C.3.2. Because $T_2\{e_1/x\} \equiv T_2\{e_1'/x\}$ by Lemmas C.1.2 and C.1.3 (3), we have $\emptyset \vdash e_2 : T_2\{e_1'/x\}$ by (T_CONV). Thus, by (T_PAIR), $\emptyset \vdash (e_1', e_2) : x{:}T_1 \times T_2$.
If e_1 is a value but e_2 is not, then $e_2 \longrightarrow e_2'$ for some e_2'. By the IH, $\emptyset \vdash e_2' : T_2\{e_1/x\}$ and thus $\emptyset \vdash (e_1, e_2') : x{:}T_1 \times T_2$.
Otherwise, if e_1 and e_2 are values, then so is (e_1, e_2).

Case (T_PROJ1): We are given $\emptyset \vdash e_1.1 : T$ for some e_1. By inversion, we have $\emptyset \vdash e_1 : x{:}T \times T_2$ for some x and T_2. If e_1 is not a value, then $e_1 \longrightarrow e_1'$ for some e_1'. By the IH, $\emptyset \vdash e_1' : x{:}T \times T_2$ and thus $\emptyset \vdash e_1'.1 : T$ by (T_PROJ1). Otherwise, if e_1 is a value, we finish by the case (1).

Case (T_PROJ2): We are given $\emptyset \vdash e_2.2 : T_2\{e_2.1/x\}$ for some e_2, T_2 and x. By inversion, we have $\emptyset \vdash e_2 : x{:}T_1 \times T_2$ for some T_1. If e_2 is not a value, then $e_2 \longrightarrow e_2'$ for some e_2'. By the IH, $\emptyset \vdash e_2' : x{:}T \times T_2$ and thus $\emptyset \vdash e_2'.2 : T_2\{e_2'.1/x\}$ by (T_PROJ2). Because $T_2\{e_2'.1/x\} \equiv T_2\{e_2.1/x\}$ by Lemmas C.1.2, C.1.3 (3) and C.1.1, we have $\emptyset \vdash e_2'.2 : T_2\{e_2.1/x\}$ by Lemma C.3.14 and (T_CONV). Otherwise, if e_2 is a value, we finish by the case (1).

Case (T_IF): We are given $\emptyset \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ for some e_1, e_2 and e_3. By inversion, we have $\emptyset \vdash e_1 : \mathsf{bool}$ and $\emptyset \vdash e_2 : T$ and $\emptyset \vdash e_3 : T$. If e_1 is not a value, $e_1 \longrightarrow e_1'$ for some e_1'. By the IH, $\emptyset \vdash e_1' : \mathsf{bool}$ and thus $\emptyset \vdash \mathsf{if}\ e_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ by (T_IF). Otherwise, if e_1 is a value, then we finish by the case (1).

Case (T_CTR): We are given $\emptyset \vdash C\langle e_1\rangle e_2 : \tau\langle e_1\rangle$ for some C, e_1, e_2 and τ. By inversion, we have $\mathit{TypSpecOf}(C) = x{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle x\rangle$ and $\emptyset \vdash e_1 : T_1$ and $\emptyset \vdash e_2 : T_2\{e_1/x\}$ and $\emptyset \vdash \tau\langle e_1\rangle$. If e_2 is not a value, then $e_2 \longrightarrow e_2'$ for some e_2'. By the IH, $\emptyset \vdash e_2' : T_2\{e_1/x\}$ and thus $\emptyset \vdash C\langle e_1\rangle e_2' : \tau\langle e_1\rangle$ by (T_CTR). Otherwise, if e_2 is a value, then so is $C\langle e_1\rangle e_2$.

Case (T_MATCH): We are given $\emptyset \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i} : T$. By inversion, we have $\emptyset \vdash e_0 : \tau\langle e''\rangle$ and $\emptyset \vdash T$ and $\mathit{CtrsOf}(\tau) = \overline{C_i}^{\,i}$ and $\mathit{ArgTypeOf}(\tau) = y{:}T'$ and, for all i, $\mathit{CtrArgOf}(C_i) = T_i$ and $x_i{:}T_i\{e''/y\} \vdash e_i : T$. If e_0 is not a value, then $e_0 \longrightarrow e_0'$ for some e_0'. By the IH, $\emptyset \vdash e_0' : \tau\langle e''\rangle$ and thus $\emptyset \vdash \mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i} : T$ by (T_MATCH). Otherwise, if e_0 is a value, then we finish by the case (1).

Case (T_WCHECK): We are given $\emptyset \vdash \langle\!\langle \{x{:}T_1 \mid e_1\}, e_2\rangle\!\rangle^{\ell} : \{x{:}T_1 \mid e_1\}$ for some x, T_1, e_1, e_2 and ℓ. By inversion, we have $\emptyset \vdash \{x{:}T_1 \mid e_1\}$ and $\emptyset \vdash e_2 : T_1$. If e_2 is not a value, then $e_2 \longrightarrow e_2'$ for some e_2'. By the IH, $\emptyset \vdash e_2' : T_1$ and thus $\emptyset \vdash \langle\!\langle \{x{:}T_1 \mid e_1\}, e_2'\rangle\!\rangle^{\ell} : \{x{:}T_1 \mid e_1\}$ by (T_WCHECK). Otherwise, if e_2 is a value, then we finish by the case (1).

Case (T_ACHECK): We are given $\emptyset \vdash \langle\{x{:}T_1 \mid e_1\}, e_2, v\rangle^{\ell} : \{x{:}T_1 \mid e_1\}$ for some x, T_1, e_1, e_2, v and ℓ. By inversion, we have $\emptyset \vdash \{x{:}T_1 \mid e_1\}$ and $\emptyset \vdash v : T_1$ and $\emptyset \vdash e_2 : \mathsf{bool}$ and $e_1\{v/x\} \longrightarrow^* e_2$. If e_2 is not a value, then $e_2 \longrightarrow e_2'$ for some e_2'. By the IH, $\emptyset \vdash e_2' : \mathsf{bool}$. Because $e_1\{v/x\} \longrightarrow^* e_2'$, we have $\emptyset \vdash \langle\{x{:}T_1 \mid e_1\}, e_2', v\rangle^{\ell} : \{x{:}T_1 \mid e_1\}$. Otherwise, if e_2 is a value, then we finish by the case (1).

Case (T_CONV): By inversion, we have $\emptyset \vdash e : T'$ and $T' \equiv T$ and $\emptyset \vdash T$ for some T'. Since $e \longrightarrow e'$, we have $\emptyset \vdash e' : T'$ by the IH. By (T_CONV), $\emptyset \vdash e' : T$. □

Definition 13. *We define a function* refines *from types to sets of pairs of a bound variable and a term, as follows.*

$$\begin{array}{lcl} \mathit{refines}\,(\{x{:}T \mid e\}) & = & \{(x,e)\} \cup \mathit{refines}\,(T) \\ \mathit{refines}\,(T) & = & \emptyset \qquad \textit{(If } T \textit{ is not a refinement type.)} \end{array}$$

In addition, we write $\vdash v : \mathit{refines}\,(T)$ *if (1)* v *is a closed value, and (2) for any* $(x,e) \in \mathit{refines}\,(T)$, $e\{v/x\} \longrightarrow^* \mathsf{true}$.

Lemma C.3.17.

(1) If $T_1 \Rrightarrow T_2$, *then* $\vdash v : \mathit{refines}\,(T_1)$ *iff* $\vdash v : \mathit{refines}\,(T_2)$.

(2) If $T_1 \equiv T_2$, *then* $\vdash v : \mathit{refines}\,(T_1)$ *iff* $\vdash v : \mathit{refines}\,(T_2)$.

Proof.

1. From $T_1 \Rrightarrow T_2$, there exist some T, x, e_1' and e_2' such that $T_1 = T\{e_1'/x\}$ and $T_2 = T\{e_2'/x\}$ and $e_1' \longrightarrow e_2'$. By induction on T.

   Case $T = \mathsf{bool}$, $y{:}T_1' \to T_2'$, $y{:}T_1' \times T_2'$, or $\tau\langle e\rangle$: Obvious because $\mathit{refines}\,(T_1)$ and $\mathit{refines}\,(T_2)$ are empty.

   Case $T = \{y{:}T' \mid e'\}$: Without loss of generality, we suppose that y is a fresh variable. Since $T'\{e_1'/x\} \Rrightarrow T'\{e_2'/x\}$, it suffices to show that $e'\{e_1'/x\}\{v/y\} \longrightarrow^* \mathsf{true}$ iff $e'\{e_2'/x\}\{v/y\} \longrightarrow^* \mathsf{true}$ by the IH. For $i \in \{1,2\}$, since v and e_i' are closed values (recall that the evaluation relation is defined over closed terms), we have $e'\{e_i'/x\}\{v/y\} = e'\{v/y\}\{e_i'/x\}$. Since $e'\{v/y\}\{e_1'/x\} \Rrightarrow e'\{v/y\}\{e_2'/x\}$, we finish by Lemma 29.

2. By induction on $T_1 \equiv T_2$.

   Case $T_1 \Rrightarrow T_2$: By the case (1).

   Case transitivity and symmetry: By the IH(s).

□

Lemma C.3.18. *If* $\emptyset \vdash v : T$, *then* $\vdash v : \mathit{refines}\,(T)$.

Proof. By induction on $\emptyset \vdash v : T$.

Case (T_CONST), (T_ABS), (T_CAST), (T_PAIR) or (T_CTR): Obvious because $\mathit{refines}\,(T) = \{\}$.

Case (T_VAR), (T_BLAME), (T_APP), (T_PROJ1), (T_PROJ2), (T_MATCH), (T_IF), (T_WCHECK) or (T_ACHECK): Contradictory.

Case (T_CONV): By inversion, we have $\emptyset \vdash v : T'$ for some T' such that $T' \equiv T$. By the IH and Lemma C.3.17 (2), we finish.

Case (T_FORGET): By inversion, we have $\emptyset \vdash v : \{x{:}T \mid e\}$ for some x and e. By the IH, we finish.

Case (T_EXACT): We are given $\emptyset \vdash v : \{x{:}T' \mid e'\}$ for some x, T' and e'. By inversion, we have $\emptyset \vdash v : T'$ and $e'\{v/x\} \longrightarrow^* \mathsf{true}$. Since $\mathit{refines}(\{x{:}T' \mid e'\}) = \mathit{refines}(T') \cup \{(x, e')\}$, we finish by the IH. □

Theorem 9 (Type Soundness). *If* $\emptyset \vdash e : T$, *then*

1. $e \longrightarrow^* v$ *for some* v *such that* $\emptyset \vdash v : T$ *and* $\vdash v : \mathit{refines}(T)$;
2. $e \longrightarrow^* \Uparrow \ell$ *for some* ℓ; *or*
3. *there is an infinite sequence of evaluation* $e \longrightarrow e_1 \longrightarrow \cdots$.

Proof. Suppose that $e \longrightarrow^* e'$ for some e' such that e' cannot reduce. We show the theorem by mathematical induction on the number of evaluation steps of e.

1. 0: We know that e cannot reduce. Since $\emptyset \vdash e : T$, we find that e is a value or a blaming by Lemma 27. Moreover, if e is a value, then $\vdash e : \mathit{refines}(T)$ by Lemma C.3.18.
2. $i+1$: We are given $e \longrightarrow e'' \longrightarrow^i e'$ for some e''. By Lemma 28 (2), $\emptyset \vdash e'' : T$ and thus we finish by the IH. □

C.4 Translation

In this section, as a proof of correctness of our translation, we show that: the translation generates a well-formed datatype (Theorem 10); a cast from a refinement type to the generated datatype always succeeds (Lemma C.4.9); and a cast from the datatype to the refinement type also succeeds (Lemma C.4.12). In this section, we assume a few things. First, type definition environments include int list. Second, we make type definition environments and constructor choice functions explicit sometimes; we write $\langle\Sigma, \delta\rangle; \Gamma \vdash e : T$, $\langle\Sigma, \delta\rangle; \Gamma \vdash T$, and $\langle\Sigma, \delta\rangle \vdash \Gamma$ to expose both in typing judgments and $\delta \vdash e_1 \longrightarrow e_2$ and $\delta \vdash e_1 \longrightarrow^* e_2$ to expose constructor choice functions in evaluation. We still assume that they are well formed. We write $v\downarrow_\tau$ for $\langle\Sigma, \delta\rangle \vdash v\downarrow_\tau$ if Σ and δ are not important or clear from the context. Finally, we assume that the input predicate function F takes the form

$$\mathtt{fix}\, f(y{:}T, x{:}\mathsf{int\ list}) = \mathsf{match}\ x\ \mathsf{with}\ [\,] \to e_1 \mid z_1 :: z_2 \to e_2$$

and it is translatable under Σ. We refer to metasymbols (f, y, x, e_1, etc.) in the definition of F as ones with subscript F. For example, y in F is written as y^F when we want to emphasize that it is from F.

C.4.1 Static Correctness

We first show that the new datatype generated from a translatable function by the translation algorithm is well formed.

Lemma C.4.1 (Type Definition Weakening). *Let* ς *be a type definition.*

(1) If $\langle\Sigma, \delta\rangle; \Gamma \vdash e : T$, *then* $\langle\Sigma, \varsigma, \delta\rangle; \Gamma \vdash e : T$.

(2) If $\langle\Sigma, \delta\rangle; \Gamma \vdash T$, *then* $\langle\Sigma, \varsigma, \delta\rangle; \Gamma \vdash T$.

(3) If $\langle\Sigma, \delta\rangle \vdash \Gamma$, *then* $\langle\Sigma, \varsigma, \delta\rangle \vdash \Gamma$.

Proof. Straightforward by induction on each derivation. □

Definition 14 (Free Variables in Typing Contexts)**.** *We write* $\mathsf{FV}(\Gamma)$ *to denote the set of free variables in a typing context* Γ*. Formally, it is defined as follows:*

$$\begin{array}{lcl} \mathsf{FV}(\emptyset) & = & \emptyset \\ \mathsf{FV}(\Gamma, x{:}T) & = & \mathsf{FV}(\Gamma) \cup (\mathsf{FV}(T) \backslash \mathsf{dom}(\Gamma)) \end{array}$$

where $\mathsf{dom}(\Gamma)$ *means the set of binding variables in* Γ*.*

Lemma C.4.2 (Strengthening)**.**

(1) If $\Gamma_1, x{:}T', \Gamma_2 \vdash e : T$ *and* $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e)$*, then* $\Gamma_1, \Gamma_2 \vdash e : T$*.*

(2) If $\Gamma_1, x{:}T', \Gamma_2 \vdash T$ *and* $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(T)$*, then* $\Gamma_1, \Gamma_2 \vdash T$*.*

(3) If $\vdash \Gamma_1, x{:}T', \Gamma_2$ *and* $x \notin \mathsf{FV}(\Gamma_2)$*, then* $\vdash \Gamma_1, \Gamma_2$*.*

Proof. By induction on each derivation. The interesting cases are for (T_ABS), (T_APP) and (T_MATCH).

1. By case analysis on the rule applied last.

   Case (T_CONST): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash c : \mathsf{bool}$. By inversion, we have $\vdash \Gamma_1, x{:}T', \Gamma_2$. By the IH, $\vdash \Gamma_1, \Gamma_2$ and thus $\Gamma_1, \Gamma_2 \vdash c : \mathsf{bool}$ by (T_CONST).

   Case (T_VAR): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash y : T$. By inversion, we have $\vdash \Gamma_1, x{:}T', \Gamma_2$ and $y{:}T \in \Gamma_1, x{:}T', \Gamma_2$. By the IH, $\vdash \Gamma_1, \Gamma_2$. We find that $x \neq y$ from $x \notin \mathsf{FV}(y)$. Thus, $\Gamma_1, \Gamma_2 \vdash y : T$ by (T_VAR).

   Case (T_BLAME): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \Uparrow \ell : T$. By inversion, we have $\vdash \Gamma_1, x{:}T', \Gamma_2$ and $\emptyset \vdash T$. By the IH, $\vdash \Gamma_1, \Gamma_2$ and thus $\Gamma_1, \Gamma_2 \vdash \Uparrow \ell : T$ by (T_BLAME).

   Case (T_ABS): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \mathtt{fix}\, f(y{:}T_1){:}T_2 = e_2 : y{:}T_1 \to T_2$. Without loss of generality, we can suppose that f and y are fresh for x. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2, f{:}(y{:}T_1 \to T_2), y{:}T_1 \vdash e_2 : T_2$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(\mathtt{fix}\, f(y{:}T_1){:}T_2 = e_2)$, we find that $x \notin \mathsf{FV}(\Gamma_2, f{:}(y{:}T_1 \to T_2), y{:}T_1) \cup \mathsf{FV}(e_2)$. Note that, thanks to type annotation T_2 in the lambda abstraction, we can find $x \notin \mathsf{FV}(T_2)$. Thus, by the IH, $\Gamma_1, \Gamma_2, f{:}(y{:}T_1 \to T_2), y{:}T_1 \vdash e_2 : T_2$. By (T_ABS), we finish.

   Case (T_CAST): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \langle T_1 \Leftarrow T_2 \rangle^\ell : T_2 \to T_1$. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash T_1$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash T_2$ and $T_1 \parallel T_2$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(\langle T_1 \Leftarrow T_2 \rangle^\ell)$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(T_1) \cup \mathsf{FV}(T_2)$. Thus, by the IHs, $\Gamma_1, \Gamma_2 \vdash T_1$ and $\Gamma_1, \Gamma_2 \vdash T_2$. By (T_CAST), we finish.

   Case (T_APP): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash e_1\, e_2 : T_2\{e_2/y\}$. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash e_1 : y{:}T_1 \to T_2$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash e_2 : T_1$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_1\, e_2)$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2)$. Thus, by the IHs, $\Gamma_1, \Gamma_2 \vdash e_1 : y{:}T_1 \to T_2$ and $\Gamma_1, \Gamma_2 \vdash e_2 : T_1$. By (T_APP), we finish.

   Case (T_PAIR): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash (e_1, e_2) : y{:}T_1 \times T_2$. Without loss of generality, we can suppose that y is fresh for x. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash e_1 : T_1$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash e_2 : T_2\{e_1/y\}$ and $\Gamma_1, x{:}T', \Gamma_2, y{:}T_1 \vdash T_2$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}((e_1, e_2))$, we find that

$x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2)$. Thus, by the IHs, $\Gamma_1, \Gamma_2 \vdash e_1 : T_1$ and $\Gamma_1, \Gamma_2 \vdash e_2 : T_2\{e_1/y\}$. By Lemma C.3.14, $x \notin \mathsf{FV}(T_1) \cup \mathsf{FV}(T_2)$. Thus, by the IH, $\Gamma_1, \Gamma_2, y{:}T_1 \vdash T_2$. By (T_PAIR), we finish.

Case (T_PROJ1): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash e_1.1 : T$. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash e_1 : y{:}T_1 \times T_2$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_1.1)$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_1)$. Thus, by the IH, $\Gamma_1, \Gamma_2 \vdash e_1 : y{:}T_1 \times T_2$. By (T_PROJ1), we finish.

Case (T_PROJ2): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash e_2.2 : T_2\{e_2.1/y\}$. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash e_2 : y{:}T_1 \times T_2$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_2.2)$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_2)$. Thus, by the IH, $\Gamma_1, \Gamma_2 \vdash e_2 : y{:}T_1 \times T_2$. By (T_PROJ2), we finish.

Case (T_CTR): We are given $[G1, x : T', G2| - Ce1e2 : te1]$. By inversion, we have $\mathit{TypSpecOf}(C) = y{:}T_1 \rightarrowtail T_2 \rightarrowtail \tau\langle y\rangle$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash e_1 : T_1$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash e_2 : T_2\{e_1/y\}$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash \tau\langle e_1\rangle$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(C\langle e_1\rangle e_2)$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2)$. Thus, by the IHs, $\Gamma_1, \Gamma_2 \vdash e_1 : T_1$ and $\Gamma_1, \Gamma_2 \vdash T_2\{e_1/y\}$ and $\Gamma_1, \Gamma_2 \vdash \tau\langle e_1\rangle$. By (T_CTR), we finish.

Case (T_MATCH): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ y_i \to e_i}^{\,i} : T$. We can suppose that each y_i is fresh for x. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash e_0 : \tau\langle e'\rangle$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash T$ and $\mathit{CtrsOf}(\tau) = \overline{C_i}^{\,i}$ and $\mathit{ArgTypeOf}(\tau) = y{:}T''$ and for any i, $\mathit{CtrArgOf}(C_i) = T_i$ and $\Gamma_1, x{:}T', \Gamma_2, y_i{:}T_i\{e'/y\} \vdash e_i : T$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(\mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ y_i \to e_i}^{\,i})$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_0) \cup \bigcup_i \mathsf{FV}(e_i)$. Thus, by the IH, $\Gamma_1, \Gamma_2 \vdash e_0 : \tau\langle e'\rangle$. By Lemma C.3.14 and its inversion, $x \notin \mathsf{FV}(e')$. From well-formedness of the type definition environment, $x \notin \mathsf{FV}(T_i)$. Thus, by the IHs, for any i, $\Gamma_1, \Gamma_2, y_i{:}T_i\{e'/y\} \vdash e_i : T$. By Lemma C.3.14, $x \notin \mathsf{FV}(T)$ (noting τ has at least one constructor from well-formedness of the type definition environment). By the IH, $\Gamma_1, \Gamma_2 \vdash T$. By (T_MATCH), we finish.

Case (T_IF): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathsf{bool}$. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash e_1 : \mathsf{bool}$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash e_2 : T$ and $\Gamma_1, x{:}T', \Gamma_2 \vdash e_3 : T$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3)$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \cup \mathsf{FV}(e_3)$. By the IHs, $\Gamma_1, \Gamma_2 \vdash e_1 : \mathsf{bool}$ and $\Gamma_1, \Gamma_2 \vdash e_2 : T$ and $\Gamma_1, \Gamma_2 \vdash e_3 : T$. By (T_IF), we finish.

Case (T_ACHECK): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \langle\{y{:}T_1 \mid e_1\}, e_2, v\rangle^{\ell} : \{y{:}T_1 \mid e_1\}$. By inversion, we have $\vdash \Gamma_1, x{:}T', \Gamma_2$ and $\emptyset \vdash \{y{:}T_1 \mid e_1\}$ and $\emptyset \vdash v : T_1$ and $\emptyset \vdash e_2 : \mathsf{bool}$ and $e_1\{v/y\} \longrightarrow^* e_2$. By the IH, $\vdash \Gamma_1, \Gamma_2$. By (T_ACHECK), we finish.

Case (T_WCHECK): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \langle\!\langle\{y{:}T_1 \mid e_1\}, e_2\rangle\!\rangle^{\ell} : \{y{:}T_1 \mid e_1\}$. By inversion, we have $\vdash \Gamma_1, x{:}T', \Gamma_2$ and $\emptyset \vdash \{y{:}T_1 \mid e_1\}$ and $\emptyset \vdash e_2 : T_1$. By the IH, $\vdash \Gamma_1, \Gamma_2$. By (T_ACHECK), we finish.

Case (T_CONV): By inversion, we have $\vdash \Gamma_1, x{:}T', \Gamma_2$ and $\emptyset \vdash e : T''$ and $T'' \equiv T$ and $\emptyset \vdash T$. By the IH, $\vdash \Gamma_1, \Gamma_2$. By (T_CONV), we finish.

Case (T_FORGET): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash v : T$. By inversion, we have $\vdash \Gamma_1, x{:}T', \Gamma_2$ and $\emptyset \vdash v : \{y{:}T \mid e'\}$. By the IH, $\vdash \Gamma_1, \Gamma_2$. By (T_FORGET), we finish.

Case (T_EXACT): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash v : \{y{:}T'' \mid e''\}$. By inversion, we have $\vdash \Gamma_1, x{:}T', \Gamma_2$ and $\emptyset \vdash v : T''$ and $\emptyset \vdash \{y{:}T'' \mid e''\}$ and $e''\{v/y\} \longrightarrow^*$ true. By the IH, $\vdash \Gamma_1, \Gamma_2$. By (T_EXACT), we finish.

2. By case analysis on the rule applied last.

   Case (WT_BASE): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \mathsf{bool}$. By the IH and (WT_BASE), we finish.

   Case (WT_FUN): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash y : T_1 \to T_2$. Without loss of generality, we can suppose that y is fresh for x. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash T_1$ and $\Gamma_1, x{:}T', \Gamma_2, y{:}T_1 \vdash T_2$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(y{:}T_1 \to T_2)$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(T_1) \cup \mathsf{FV}(T_2)$. By the IHs, $\Gamma_1, \Gamma_2 \vdash T_1$ and $\Gamma_1, \Gamma_2, y{:}T_1 \vdash T_2$. By (WT_FUN), we finish.

   Case (WT_PROD): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash y{:}T_1 \times T_2$. Without loss of generality, we can suppose that y is fresh for x. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash T_1$ and $\Gamma_1, x{:}T', \Gamma_2, y{:}T_1 \vdash T_2$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(y{:}T_1 \times T_2)$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(T_1) \cup \mathsf{FV}(T_2)$. By the IHs, $\Gamma_1, \Gamma_2 \vdash T_1$ and $\Gamma_1, \Gamma_2, y{:}T_1 \vdash T_2$. By (WT_PROD), we finish.

   Case (WT_REFINE): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \{y{:}T'' \mid e''\}$. Without loss of generality, we can suppose that y is fresh for x. By inversion, we have $\Gamma_1, x{:}T', \Gamma_2 \vdash T''$ and $\Gamma_1, x{:}T', \Gamma_2, y{:}T'' \vdash e'' : \mathsf{bool}$. Since $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(\{y{:}T'' \mid e''\})$, we find that $x \notin \mathsf{FV}(\Gamma_2) \cup \mathsf{FV}(T'') \cup \mathsf{FV}(e'')$. Thus, by the IHs, $\Gamma_1, \Gamma_2 \vdash T''$ and $\Gamma_1, \Gamma_2, y{:}T'' \vdash e'' : \mathsf{bool}$. By (WT_REFINE), we finish.

   Case (WT_DATATYPE): We are given $\Gamma_1, x{:}T', \Gamma_2 \vdash \tau\langle e' \rangle$. By the IH and (WT_DATATYPE), we finish.

3. By case analysis on the rule applied last.

   Case (WC_EMPTY): Obvious.

   Case (WC_EXTENDVAR): If $\Gamma_2 = \emptyset$, then, by inversion, we have $\vdash \Gamma_1$ and thus we finish. Otherwise, if $\Gamma_2 = \Gamma_2', y{:}T''$, then, by inversion, $\vdash \Gamma_1, x{:}T', \Gamma_2'$ and $\Gamma_1, x{:}T', \Gamma_2' \vdash y : T''$. By the IHs and (WC_EXTENDVAR), we finish. □

Lemma C.4.3 (Application Inversion). *If* $\Gamma \vdash e_1\ e_2 : T$, *then*

- $\Gamma \vdash e_1 : x{:}T_1 \to T_2$,
- $\Gamma \vdash e_2 : T_1$, *and*
- $T_2\{e_2/x\} \equiv T$

for some x, T_1 *and* T_2.

Proof. Similarly to Lemma C.3.9, by induction on the typing derivation. Only two rules can be applied to the application.

Case (T_APP): Since $T = T_2\{e_2/x\}$, we have $T_2\{e_2/x\} \equiv T$ by Lemma C.1.1 (reflexivity). By inversion, we finish.

Case (T_CONV): By inversion, we have $\emptyset \vdash e_1\ e_2 : T'$ and $T' \equiv T$ for some T'. By the IH, we have $\emptyset \vdash e_1 : x{:}T_1 \to T_2$ and $\emptyset \vdash e_2 : T_1$ and $T_2\{e_2/x\} \equiv T'$. We have $T_2\{e_2/x\} \equiv T$ by Lemma C.1.1 (transitivity). By Lemma C.3.1, we finish. □

Lemma C.4.4 (Variable Inversion). *If* $\Gamma \vdash x : T$, *then* $\vdash \Gamma$ *and* $x{:}T \in \Gamma$.

Proof. Obvious because only (T_VAR) can drive $\Gamma \vdash x : T$. □

Lemma C.4.5. *Let F be a translatable function, e be a subterm of ${e_2}^F$, $\Gamma_1 = {f^F}{:}T^F \to \mathsf{int\ list} \to \mathsf{bool}, {y^F}{:}T^F, {z_1}^F{:}\mathsf{int}$, and Γ_2 be a typing context. If $\Gamma_1, \Gamma_2 \vdash e : \mathsf{bool}$ and $(e_{\mathit{opt}_0}, e_0) \in \mathit{GenContracts}\,(e)$, then:*

- *for any e', if $e_{\mathit{opt}_0} = \mathit{Some}\ e'$, then ${y^F}{:}T^F, {z_1}^F{:}\mathsf{int} \vdash e' : T^F$; and*
- $\Gamma_1, \Gamma_2 \vdash e_0 : \mathsf{bool}$.

Proof. By structural induction on e with case analysis on $\Gamma_1, \Gamma_2 \vdash e : \mathsf{bool}$.

Case (T_CONST): Obvious because $\mathit{GenContracts}\,(\mathsf{true}) = \{(\mathit{None}, \mathsf{true})\}$ and $\mathit{GenContracts}\,(\mathsf{false}) = \emptyset$.

Case (T_VAR), (T_ABS), (T_CAST), (T_APP), (T_PAIR), (T_PROJi) for $i \in \{1, 2\}$, (T_CTR), (T_FORGET), (T_EXACT), (T_BLAME), (T_ACHECK), and (T_WCHECK): Obvious because $\mathit{GenContracts}\,(e) = \{(\mathit{None}, e)\}$.

Case (T_IF): We are given $\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathsf{bool}$. By inversion, we have $\Gamma_1, \Gamma_2 \vdash e_1 : \mathsf{bool}$ and $\Gamma_1, \Gamma_2 \vdash e_2 : \mathsf{bool}$ and $\Gamma_1, \Gamma_2 \vdash e_3 : \mathsf{bool}$. There are three cases which we have to consider.

Case $e_1 = {f^F}\ {e_1'}\ {{z_2}^F}$ where $\mathsf{FV}\,(e_1') \subseteq \{{y^F}, {z_1}^F\}$: Then,

$$\begin{array}{l}\mathit{GenContracts}\,(e) = \\ \quad \{(\mathit{Some}\ e_1', e_2)\} \cup \\ \quad \{(e_{\mathbf{opt}}, \mathsf{if}\ {f^F}\ {e_1'}\ {{z_2}^F}\ \mathsf{then\ false\ else}\ e_3') \mid (e_{\mathbf{opt}}, e_3') \in \mathit{GenContracts}\,(e_3)\}\end{array}$$

We first show ${y^F}{:}T^F, {z_1}^F{:}\mathsf{int} \vdash e_1' : T^F$. Since $\Gamma_1, \Gamma_2 \vdash {f^F}\ {e_1'}\ {{z_2}^F} : \mathsf{bool}$, we find that $\Gamma_1, \Gamma_2 \vdash f^F : x{:}T_1 \to T_2$ and $\Gamma_1, \Gamma_2 \vdash e_1' : T_1$ for some x, T_1 and T_2, by applying Lemma C.4.3 twice. By Lemma C.4.4, $x{:}T_1 \to T_2 = T^F \to \mathsf{int\ list} \to \mathsf{bool}$ since ${f^F}{:}x{:}T_1 \to T_2 \in \Gamma_1$. Thus, $T_1 = T^F$ and so $\Gamma_1, \Gamma_2 \vdash e_1' : T^F$. Since $\mathsf{FV}\,(e_1') \subseteq \{{y^F}, {z_1}^F\}$, and $f^F \notin \mathsf{FV}\,(T^F)$ by Lemma C.3.14, we have ${y^F}{:}T^F, {z_1}^F{:}\mathsf{int} \vdash e_1' : T^F$ by Lemma C.4.2 (1). In addition, we have $\Gamma_1, \Gamma_2 \vdash e_2 : \mathsf{bool}$ from the premise of the typing derivation.

Let $(e_{\mathbf{opt}}, e_3') \in \mathit{GenContracts}\,(e_3)$. It suffices to show that (1) for any e', if $e_{\mathbf{opt}} = \mathit{Some}\ e'$, then ${y^F}{:}T^F, {z_1}^F{:}\mathsf{int} \vdash e' : T^F$ and (2) $\Gamma_1, \Gamma_2 \vdash \mathsf{if}\ {f^F}\ {e_1'}\ {{z_2}^F}\ \mathsf{then\ false\ else}\ e_3' : \mathsf{bool}$. The case (1) is shown by the IH. The case (2) is obvious by (T_IF) because $\Gamma_1, \Gamma_2 \vdash \mathsf{false} : \mathsf{bool}$ by Lemmas C.3.14 and C.3.1 and $\Gamma_1, \Gamma_2 \vdash e_3' : \mathsf{bool}$ by the IH.

Case $e_1 \neq {f^F}\ {e_1'}\ {{z_2}^F}$ for any e_1' such that $\mathsf{FV}\,(e_1') \subseteq \{{y^F}, {z_1}^F\}$, and a term of the form ${f^F}\ {e_1'}\ {{z_2}^F}$ for some e_1' occurs in e_2 or e_3: Similarly to the above. We have

$$\begin{array}{l}\mathit{GenContracts}\,(e) = \\ \quad \{(e_{\mathbf{opt}}, \mathsf{if}\ e_1\ \mathsf{then}\ e_2'\ \mathsf{else\ false}) \mid (e_{\mathbf{opt}}, e_2') \in \mathit{GenContracts}\,(e_2)\} \cup \\ \quad \{(e_{\mathbf{opt}}, \mathsf{if}\ e_1\ \mathsf{then\ false\ else}\ e_3') \mid (e_{\mathbf{opt}}, e_3') \in \mathit{GenContracts}\,(e_3)\}.\end{array}$$

Since $\Gamma_1, \Gamma_2 \vdash e_2 : \mathsf{bool}$ and $\Gamma_1, \Gamma_2 \vdash e_3 : \mathsf{bool}$, we finish by the IHs.

Case otherwise: Obvious because $\mathit{GenContracts}\,(e) = \{(\mathit{None}, e)\}$.

Case (T_MATCH): Similarly to the case for (T_IF). We are given $\Gamma_1, \Gamma_2 \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i}^{\,i \in \{1,\ldots,n\}} : \mathsf{bool}$. By inversion, we have $\Gamma_1, \Gamma_2 \vdash e_0 :$

$\tau\langle e'\rangle$ and $\mathit{ArgTypeOf}(\tau) = x':T'$ and, for any $i \in \{1, ..., n\}$, $\mathit{CtrArgOf}(C_i) = T_i$ and $\Gamma_1, \Gamma_2, x_i{:}T_i\,\{e'/x'\} \vdash e_i : \mathsf{bool}$ for some τ, e', x', T', and $\overline{T_i}^{\,i \in \{1,...,n\}}$.

If some e_i contains a term of the form ${f^F}\ {e_1'}\ {z_2}^F$ for some e_1', then we have

$$\begin{array}{l}\mathit{GenContracts}(e) = \\ \quad \bigcup_{j \in \{1,...,n\}} \{(e_{\mathbf{opt}}, \mathsf{match}\ e_0\ \mathsf{with}\ \overline{C_i\ x_i \to e_i''}^{\,i \in \{1,...,n\}}) \mid \\ \qquad\qquad (e_{\mathbf{opt}}, e_j'') \in \mathit{GenContracts}(e_j) \wedge \forall i \neq j\ .\ e_i'' = \mathsf{false}\}.\end{array}$$

We finish by the IHs with the fact that, for any i, $\Gamma_1, \Gamma_2, x_i{:}T_i\,\{e'/x'\} \vdash \mathsf{false} : \mathsf{bool}$ by Lemmas C.3.14 and C.3.1, and so $\Gamma_1, \Gamma_2, x_i{:}T_i\,\{e'/x'\} \vdash e_i'' : \mathsf{bool}$.

Otherwise, obvious because $\mathit{GenContracts}(e) = \{(\mathit{None}, e)\}$.

Case (T_CONV): By inversion, we have $\emptyset \vdash e : T$ and $T \equiv \mathsf{bool}$. If $e = \mathsf{false}$, then obvious because $\mathit{GenContracts}(\mathsf{false}) = \emptyset$. Otherwise, since ${f^F}$ (and ${z_2}^F$) does not occur in e, we have $\mathit{GenContracts}(e) = \{(\mathit{None}, e)\}$ (even if $e = \mathsf{true}$) and so we finish.

□

Theorem 10 (Translation Generates Well-Formed Datatype)**.** *Let Σ be a well-formed type definition environment and F be a translatable function under Σ. Then, the type definition $\mathit{Trans}(F)$ is well formed under Σ.*

Proof. By definition, $\mathit{Trans}(F) = \mathsf{type}\,\tau\,\langle y^F{:}T^F\rangle = D \parallel [\,] : \{z{:}\mathsf{unit} \mid {e_1}^F\} \mid \overline{D_i \parallel (::) : T_i}^{\,i}$ where z is fresh. It suffices to show that the type definition satisfies five conditions from definition of well-formedness of type definition under type definition environment.

(a) We show that τ has constructors more than zero, which is obvious.

(b) We show that $\Sigma; \emptyset \vdash T^F$. Since F is well typed, we have $\Sigma; \emptyset \vdash T^F$ by Lemma C.3.14 and its inversion.

(c) We show that (1) $\Sigma, \mathit{Trans}(F); y^F{:}T^F \vdash \{z{:}\mathsf{unit} \mid {e_1}^F\}$ and (2) $\Sigma, \mathit{Trans}(F); y^F{:}T^F \vdash T_i$ for any i.

(1) Since F is translatable under Σ, we have $(\Sigma, \emptyset); y^F{:}T^F \vdash {e_1}^F : \mathsf{bool}$. By Lemma C.4.1, $(\Sigma, \mathit{Trans}(F), \emptyset); y^F{:}T^F \vdash {e_1}^F : \mathsf{bool}$. By Lemma C.3.1 and (T_REFINE), $(\Sigma, \mathit{Trans}(F), \emptyset); y^F{:}T^F \vdash \{z{:}\mathsf{unit} \mid {e_1}^F\}$.

(2) By definition of *GenContracts*, T_i is defined based on $\mathit{GenContracts}({e_2}^F)$. Let $(e_{\mathbf{opt}}, e) \in \mathit{GenContracts}({e_2}^F)$ and $\Gamma = f^F{:}T^F \to \mathsf{int\ list} \to \mathsf{bool}, y^F{:}T^F, {z_1}^F{:}\mathsf{int}, {z_2}^F{:}\mathsf{int\ list}$. Since F is translatable under Σ, we have $(\Sigma, \emptyset); \Gamma \vdash {e_2}^F : \mathsf{bool}$. By Lemma C.4.5, $(\Sigma, \emptyset); \Gamma \vdash e : \mathsf{bool}$. Since $(\Sigma, \emptyset); \emptyset \vdash F : T^F \to \mathsf{int\ list} \to \mathsf{bool}$, we have $(\Sigma, \emptyset); y^F{:}T^F, {z_1}^F{:}\mathsf{int}, {z_2}^F{:}\mathsf{int\ list} \vdash e\,\{F/f^F\} : \mathsf{bool}$ by Lemma C.3.2. Note that T^F is closed by Lemma C.3.14 and its inversion. By Lemma C.4.1,

$$(\Sigma, \mathit{Trans}(F), \emptyset); y^F{:}T^F, {z_1}^F{:}\mathsf{int}, {z_2}^F{:}\mathsf{int\ list} \vdash e\,\{F/f^F\} : \mathsf{bool}.$$

By case analysis on $e_{\mathbf{opt}}$, letting $\Gamma' = y^F{:}T^F, {z_1}^F{:}\mathsf{int}$.

Case $e_{\mathbf{opt}} = \mathit{Some}\ e''$: By Lemma C.3.1 and (T_ABS),

$$(\Sigma, \mathit{Trans}\,(F), \emptyset); \Gamma' \vdash \lambda\, {z_2}^F{:}\mathsf{int\ list}.e\,\{F/f^F\} \ :\ \mathsf{int\ list} \to \mathsf{bool}.$$

By Lemmas C.4.5 and C.4.1,

$$(\Sigma, \mathit{Trans}\,(F), \emptyset); \Gamma' \vdash e'' \ :\ T^F.$$

Thus,

$$(\Sigma, \mathit{Trans}\,(F), \emptyset); \Gamma' \vdash \tau\langle e''\rangle$$

by (WT_DATATYPE). By (C_DATATYPE), $\Sigma, \mathit{Trans}\,(F) \vdash \tau\langle e''\rangle \parallel \mathsf{int\ list}$. Since $(\Sigma, \mathit{Trans}\,(F), \emptyset); \Gamma' \vdash \mathsf{int\ list}$ by Lemmas C.3.14 and C.3.1 and (WT_DATATYPE), we find

$$(\Sigma, \mathit{Trans}\,(F), \emptyset); \Gamma' \vdash \langle \mathsf{int\ list} \Leftarrow \tau\langle e''\rangle\rangle^{\ell} \ :\ \tau\langle e''\rangle \to \mathsf{int\ list}$$

for any ℓ, by (T_CAST). By Lemma C.3.1, (T_VAR) and (T_APP), we have

$$(\Sigma, \mathit{Trans}\,(F), \emptyset); \Gamma', {z_2}^F{:}\tau\langle e''\rangle \vdash \langle \mathsf{int\ list} \Leftarrow \tau\langle e''\rangle\rangle^{\ell}\ {z_2}^F \ :\ \mathsf{int\ list}.$$

Letting $e_0 = (\lambda\, {z_2}^F{:}\mathsf{int\ list}.e\,\{F/f^F\})\,(\langle \mathsf{int\ list} \Leftarrow \tau\langle e''\rangle\rangle^{\ell}\ {z_2}^F)$, we have

$$(\Sigma, \mathit{Trans}\,(F), \emptyset); \Gamma', {z_2}^F{:}\tau\langle e''\rangle \vdash e_0 \ :\ \mathsf{bool}$$

by Lemma C.3.1 and (T_APP). Note that e_0 can be written as $\mathsf{let}\ {z_2}^F = \langle \mathsf{int\ list} \Leftarrow \tau\langle e''\rangle\rangle^{\ell}\ {z_2}^F\ \mathsf{in}\ e\,\{F/f^F\}$. Letting $T_0 = \tau\langle e''\rangle$, we have

$$(\Sigma, \mathit{Trans}\,(F), \emptyset); \Gamma' \vdash \{{z_2}^F{:}T_0 \mid e_0\}.$$

by (WT_REFINE). Thus, by (WT_PROD),

$$(\Sigma, \mathit{Trans}\,(F), \emptyset); y^F{:}T^F \vdash {z_1}^F \ :\ \mathsf{int} \times \{{z_2}^F{:}T_0 \mid e_0\}.$$

Note that $T_i = {z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}T_0 \mid e_0\}$.

Case $e_{\mathbf{opt}} = \mathit{None}$: By (WT_REFINE) and (WT_PROD), we have

$$(\Sigma, \mathit{Trans}\,(F), \emptyset); y^F{:}T^F \vdash {z_1}^F \ :\ \mathsf{int} \times \{{z_2}^F{:}\mathsf{int\ list} \mid e\,\{F/f^F\}\}.$$

Note that $T_i = {z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}\mathsf{int\ list} \mid e\,\{F/f^F\}\}$.

(d) We show that Σ includes int list, which is proven by the assumption.

(e) We show that (1) $\Sigma, \mathit{Trans}\,(F) \vdash \{z{:}\mathsf{unit} \mid {e_1}^F\} \parallel \mathsf{unit}$ and (2) $\Sigma, \mathit{Trans}\,(F) \vdash T_i \parallel \mathsf{int} \times \mathsf{int\ list}$. The case (1) is obvious by (C_REFINEL) and reflexivity of the compatibility relation. The case (2) is straightforward because T_i takes either of the form ${z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}\mathsf{int\ list} \mid e_0\}$ or ${z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}\tau\langle e''\rangle \mid e_0\}$, and reflexivity of the compatibility relation and $\Sigma, \mathit{Trans}\,(F) \vdash \tau\langle e''\rangle \parallel \mathsf{int\ list}$. □

C.4.2 Dynamic Correctness

Next, we show correctness of translation in the dynamic aspect: casts between refinement types with a translatable function F and the datatype generated from F succeed

always. In particular, such casts convert "constructors" but not "structures." In this section, we assume that type definition environments include the datatype generated from a translatable function F.

Definition 15. *A constructor choice function δ is said to be trivial for τ when, if the type definition of τ takes the form* $\mathsf{type}\,\tau_1\,\langle x{:}T\rangle = \overline{C_i \parallel D_i : T_i}^{\,i}$ *and each D_i belongs to τ_2, then $\delta(\langle\tau_2\langle e_2\rangle \Leftarrow \tau_1\langle e_1\rangle\rangle^{\ell}\, C_i\langle e_3\rangle e_4) = D_i$ for any e_1, e_2, e_3, and e_4.*

We say that a constructor choice function is trivial when it is trivial for *Trans* (F).

Lemma C.4.6. *Let δ be a trivial choice function. Suppose that*

$$\mathit{Trans}\,(F) = \mathsf{type}\,\tau\,\langle y^F{:}T^F\rangle = D \parallel [\,] : \{z{:}\mathsf{unit} \mid {e_1}^F\} \mid \overline{D_i \parallel (::) : {z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}T_i \mid e_i\}}^{\,i}.$$

If $\emptyset \vdash \langle\mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\, v : \mathsf{int\ list}$ under δ, then $\langle\mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\, v \longrightarrow^* v'$ under δ for some v' which is obtained by replacing data constructors D and D_i of which v consists with $[\,]$ and $(::)$, respectively.

Proof. We proceed by structural induction on v. Since $\emptyset \vdash \langle\mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\, v : \mathsf{int\ list}$, we have $\emptyset \vdash \langle\mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell} : x{:}T_1' \to T_2'$ and $\emptyset \vdash v : T_1'$ and $T_2'\{v/x\} \equiv \mathsf{int\ list}$ for some x, T_1', and T_2' by Lemma C.4.3. By Lemma C.3.6, $T_2' = \mathsf{int\ list}$. By Lemmas C.3.10 and C.3.4, we have $\emptyset \vdash \tau\langle e\rangle$ and $\tau\langle e\rangle \equiv T_1'$. We perform case analysis on v by Lemmas C.3.13 (4) and C.3.12.

Case $v = D\langle e'\rangle v'$: Since δ is trivial, $\delta(\langle\mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\, D\langle e'\rangle v') = [\,]$. Thus, by (R_DATATYPE), (R_FORGET) and (R_BASE) with (E_RED),

$$\langle\mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\, D\langle e'\rangle v' \longrightarrow^* [\,].$$

Case $v = D_j\langle e'\rangle v'$: By Lemma C.3.12, $\emptyset \vdash v' : {z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}T_i \mid e_i\}\{e'/y^F\}$. By Lemmas C.3.13 (3) and C.3.11, $v' = (v_1, v_2)$ for some v_1 and v_2 such that $\emptyset \vdash v_1 : \mathsf{int}$ and $\emptyset \vdash v_2 : \{{z_2}^F{:}T_i \mid e_i\}\{e'/y^F, v_1/{z_1}^F\}$. Note that e' is a closed term. Since δ is trivial, $\delta(\langle\mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\, D_j\langle e'\rangle v') = (::)$. Thus, by (R_DATATYPE), (R_PROD), (R_BASE) and (R_FORGET) with (E_RED),

$$\langle\mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\, D_j\langle e'\rangle v' \longrightarrow^* v_1 :: (\langle\mathsf{int\ list} \Leftarrow T_i\{e'/y^F, v_1/{z_1}^F\}\rangle^{\ell}\, v_2).$$

From *Trans*, there are two cases we have to consider. If $T_i = \mathsf{int\ list}$, then $\langle\mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\, D_j\langle e'\rangle v' \longrightarrow^* v_1 :: v_2$ by (R_DATATYPEMONO). Otherwise, if $T_i = \tau\langle e''\rangle$ for some e'', then we finish by the IH, noting $\emptyset \vdash \langle\mathsf{int\ list} \Leftarrow T_i\{e'/y^F, v_1/{z_1}^F\}\rangle^{\ell}\, v_2 : \mathsf{int\ list}$, which follows from well-typedness of v_2, compatibility of int list and τ, (T_CAST), and (T_APP). □

Definition 16 (Notation). *Let σ be a (simultaneous) substitution. Then, we write $\sigma(e)$ to denote application of σ to e.*

Lemma C.4.7. *Let F be a translatable function, v, v_1 and v_2 be closed values, σ be a simultaneous substitution including $\{F/f^F, v/y^F, v_1/{z_1}^F, v_2/{z_2}^F\}$, and e be a subterm of ${e_2}^F$. If $\sigma(e) \longrightarrow^* \mathsf{true}$, then there is a unique pair $(e_{\mathit{opt}_0}, e_0) \in \mathit{GenContracts}\,(e)$ such that*

- $\sigma(e_0) \longrightarrow^* \mathsf{true}$ *and*
- *for any e', $e_{\mathit{opt}_0} = \mathit{Some}\ e'$ implies $F\,\sigma(e')\,v_2 \longrightarrow^* \mathsf{true}$.*

Proof. By structural induction on e.

Case $e = \mathsf{true}$: Obvious since $\mathit{GenContracts}\,(\mathsf{true}) = \{(\mathit{None}, \mathsf{true})\}$.

Case $e = \mathsf{false}$: Contradictory; $\sigma(e) \longrightarrow^* \mathsf{false}$.

Case $e = \mathsf{if}\, f^F\ e'\ {z_2}^F\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'$ where $\mathsf{FV}\,(e') \subseteq \{y^F, {z_1}^F\}$: By definition of *GenContracts*, we have

$$\begin{array}{l}\mathit{GenContracts}\,(e) = \\ \quad \{(\mathit{Some}\ e', e_2')\} \cup \\ \quad \{(e_{\mathbf{opt}_0}, \mathsf{if}\, f^F\ e'\ {z_2}^F\ \mathsf{then\ false\ else}\ e_3'') \mid (e_{\mathbf{opt}_0}, e_3'') \in \mathit{GenContracts}\,(e_3')\}.\end{array}$$

By case analysis on evaluation of $\sigma(f^F\ e'\ {z_2}^F) = F\,\sigma(e')\,v_2$. Note that the evaluation result is either true or false.

Case $F\,\sigma(e')\,v_2 \longrightarrow^* \mathsf{true}$: We have

$$\begin{array}{rcl}\sigma(\mathsf{if}\, f^F\ e'\ {z_2}^F\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3') & \longrightarrow^* & \mathsf{if\ true\ then}\ \sigma(e_2')\ \mathsf{else}\ \sigma(e_3') \\ & \longrightarrow & \sigma(e_2').\end{array}$$

Since $\sigma(e) \longrightarrow^* \mathsf{true}$, we find that $\sigma(e_2') \longrightarrow^* \mathsf{true}$. Because

$$\begin{array}{rcl}\sigma(\mathsf{if}\, f^F\ e'\ {z_2}^F\ \mathsf{then\ false\ else}\ e_3'') & \longrightarrow^* & \mathsf{if\ true\ then\ false\ else}\ \sigma(e_3'') \\ & \longrightarrow & \mathsf{false},\end{array}$$

pair $(\mathit{Some}\ e', e_2')$ is the unique one satisfying the property above.

Case $F\,\sigma(e')\,v_2 \longrightarrow^* \mathsf{false}$: We have

$$\begin{array}{rcl}\sigma(\mathsf{if}\, f^F\ e'\ {z_2}^F\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3') & \longrightarrow^* & \mathsf{if\ false\ then}\ \sigma(e_2')\ \mathsf{else}\ \sigma(e_3') \\ & \longrightarrow & \sigma(e_3').\end{array}$$

Since $\sigma(e) \longrightarrow^* \mathsf{true}$, we find that $\sigma(e_3') \longrightarrow^* \mathsf{true}$. By the IH, there is a unique pair $(e_{\mathbf{opt}}, e_3'') \in \mathit{GenContracts}\,(e_3')$ satisfying the above property. We have $\sigma(\mathsf{if}\, f^F\ e'\ {z_2}^F\ \mathsf{then\ false\ else}\ e_3'') \longrightarrow^* \mathsf{true}$. Since $F\,\sigma(e')\,v_2 \longrightarrow^* \mathsf{false}$, pair $(e_{\mathbf{opt}}, \mathsf{if}\, f^F\ e'\ {z_2}^F\ \mathsf{then\ false\ else}\ e_3'')$ is the unique one satisfying the property above.

Case $e = \mathsf{if}\, e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'$ where $e_1' \neq f^F\ e'\ {z_2}^F$ for any e' such that $\mathsf{FV}\,(e') \subseteq \{y^F, {z_1}^F\}$: By case analysis on evaluation of $\sigma(e_1')$. Note that the evaluation result is either true or false.

- Case $\sigma(e_1') \longrightarrow^* \mathsf{true}$: Since $\sigma(\mathsf{if}\, e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3') \longrightarrow^* \sigma(e_2') \longrightarrow^* \mathsf{true}$, there a unique pair $(e_{\mathbf{opt}}, e_2'') \in \mathit{GenContracts}\,(e_2')$ satisfying the above property, by the IH. Since $\sigma(\mathsf{if}\, e_1'\ \mathsf{then\ false\ else}\ e_3'') \longrightarrow^* \mathsf{false}$ for any e_3'', pair $(e_{\mathbf{opt}}, \mathsf{if}\, e_1'\ \mathsf{then}\ e_2''\ \mathsf{else\ false})$ is the unique one satisfying the property above.
- Case $\sigma(e_1') \longrightarrow^* \mathsf{false}$: Since $\sigma(\mathsf{if}\, e_1'\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3') \longrightarrow^* \sigma(e_3') \longrightarrow^* \mathsf{true}$, there a unique pair $(e_{\mathbf{opt}}, e_3'') \in \mathit{GenContracts}\,(e_3')$ satisfying the above property, by the IH. Since $\sigma(\mathsf{if}\, e_1'\ \mathsf{then}\ e_2''\ \mathsf{else\ false}) \longrightarrow^* \mathsf{false}$ for any e_2'', pair $(e_{\mathbf{opt}}, \mathsf{if}\, e_1'\ \mathsf{then\ false\ else}\ e_3'')$ is the unique one satisfying the property above.

Case $e = \mathsf{match}\ e_0'\ \mathsf{with}\ \overline{C_i\ x_i \to e_i'}^{\,i \in \{1,\ldots,n\}}$: Without loss of generality, we can suppose that each x_i is fresh for σ. Since $\sigma(e) \longrightarrow^* \mathsf{true}$, we find that $\sigma(e_0') \longrightarrow^* C_j\langle e'\rangle v'$ for some $j \in \{1,\ldots,n\}$, e' and v', and thus $\sigma(e_j')\,\{v'/x_j\} \longrightarrow^* \mathsf{true}$. By the IH, there is a unique pair $(e_{\mathbf{opt}}, e_j'') \in \mathit{GenContracts}\,(e_j')$ satisfying the above property. Since $\sigma(\mathsf{match}\ e_0'\ \mathsf{with}\ C_j\ x_j \to \mathsf{false} \mid \overline{C_i\ x_i \to e_i''}^{\,i \in \{1,\ldots,n\}\setminus\{j\}}) \longrightarrow^* \mathsf{false}$, pair $(e_{\mathbf{opt}}, \mathsf{match}\ e_0'\ \mathsf{with}\ C_j\ x_j \to e_j'' \mid \overline{C_i\ x_i \to \mathsf{false}}^{\,i \in \{1,\ldots,n\}\setminus\{j\}})$ is the unique one satisfying the property above.

Case otherwise: Obvious because $\mathit{Trans}\,(e) = \{(\mathit{None}, e)\}$.

□

In what follows, we compute constructor choice functions to convert data structures. Before it, we show that extensions of constructor choice functions are conservative with respect to evaluation results.

Lemma C.4.8. *Let δ' be an extension of constructor choice function δ. If $\delta \vdash e \longrightarrow^* v$, then $\delta' \vdash e \longrightarrow^* v$.*

Proof. From the two facts: (1) δ returns a constructor whenever taking cast applications in the evaluation $e \longrightarrow^* v$ and (2) δ' returns the same constructor as δ for cast applications contained by the domain of δ. □

Definition 17. *We write $\delta_1 \uplus \delta_2$ to denote the union of graphs of constructor choice functions δ_1 and δ_2 with disjoint domains.*

Lemma C.4.9 (From Refinement Types to Datatypes)**.** *Suppose that*

$$\mathit{Trans}\,(F) = \mathsf{type}\,\tau\,\langle y^F{:}T^F\rangle = D \parallel [\,] : \{z{:}\mathsf{unit} \mid {e_1}^F\} \mid \overline{D_i \parallel (::) : {z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}T_i \mid {e_i}\}}^{\,i}.$$

Let δ be a trivial constructor choice function such that $\delta(\langle\tau\langle e'\rangle \Leftarrow \mathsf{int\ list}\rangle^\ell\, v')$ is undefined for any e' and sublist v' of v.

If $\emptyset \vdash \langle\tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell\, v : \tau\langle e\rangle$ under δ, then there exists an extension δ' of δ such that $\langle\tau\langle e\rangle \Leftarrow \{l{:}\mathsf{int\ list} \mid F\ e\ l\}\rangle^\ell\, v \longrightarrow^* v'$ under δ' where v' is obtained by replacing some occurrences of data constructors $[\,]$ and $(::)$ of which v consists with D and one of $\overline{D_i}^{\,i}$, respectively.

Proof. By Lemma C.4.3, we have $\emptyset \vdash \langle\tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell : x_0{:}T_{01} \to T_{02}$ and $\emptyset \vdash v : T_{01}$ and $T_{02}\,\{v/x_0\} \equiv \tau\langle e\rangle$ for some x_0, T_{01} and T_{02}. By Lemmas C.3.10 and C.3.4 and (T_CONV), $\emptyset \vdash v : \{x{:}\mathsf{int\ list} \mid F\ e\ x\}$ and so $F\ e\ v \longrightarrow^* \mathsf{true}$ by Theorem 9 (noting that e is a closed term since $\emptyset \vdash \tau\langle e\rangle$ by Lemma C.3.14). Thus, $e \longrightarrow^* v'$ for some v'.

We proceed by case analysis on v by Lemmas C.3.13 (4) and C.3.12.

Case $v = [\,]$: Let $\delta' = \delta \uplus \{\langle\tau\langle e\rangle \Leftarrow \mathsf{int\ list}\rangle^\ell\,[\,] \mapsto D\}$. Then, by (R_FORGET) and (R_DATATYPE) with (E_RED),

$$\delta' \vdash \langle\tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell\,[\,] \longrightarrow^* D\langle e\rangle(\langle\{z{:}\mathsf{unit} \mid {e_1}^F\,\{e/x\}\} \Leftarrow \mathsf{unit}\rangle^\ell\,())$$

Since $F\ v'\ v \longrightarrow^* \mathsf{true}$, we find that ${e_1}^F\,\{F/f^F, v'/y^F, v/x^F\} \longrightarrow^* \mathsf{true}$. Since F is translatable, we have $y{:}T \vdash {e_1}^F : \mathsf{bool}$ and so ${e_1}^F\,\{F/f^F, v'/y^F, v/x^F\} = {e_1}^F\,\{v'/y^F\}$. Thus, ${e_1}^F\,\{v'/y^F\} \longrightarrow^* \mathsf{true}$. Since $e \Rrightarrow^* v'$ by Lemma C.1.2, we have ${e_1}^F\,\{e/y^F\} \Rrightarrow^* {e_1}^F\,\{v'/y^F\}$ by Lemma C.1.5 (2). By Lemma 29 (2),

${e_1}^F\,\{e/y^F\} \longrightarrow^*$ true. Since $\delta' \vdash {e_1}^F\,\{e/y^F\} \longrightarrow^*$ true by Lemma C.4.8, we have

$$\delta' \vdash \langle \tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell\,[\,] \longrightarrow^*\ D\langle e\rangle()$$

by (R_PRECHECK), (R_BASE), (R_CHECK), and (R_OK) with (E_RED).

Case $v = (v_1 :: v_2)$: Since $F\ v'\ v \longrightarrow^*$ true, we find that

$${e_2}^F\,\{F/f^F, v'/y^F, v/{x}^F, v_1/{z_1}^F, v_2/{z_2}^F\} \longrightarrow^* \mathsf{true}.$$

Since F is translatable, ${f}^F{:}T^F \to \mathsf{int\ list} \to \mathsf{bool}, {y}^F{:}T^F, {z_1}^F{:}\mathsf{int}, {z_2}^F{:}\mathsf{int\ list} \vdash {e_2}^F : \mathsf{bool}$ and so

$${e_2}^F\,\{F/f^F, v'/y^F, v/{x}^F, v_1/{z_1}^F, v_2/{z_2}^F\} = {e_2}^F\,\{F/f^F, v'/y^F, v_1/{z_1}^F, v_2/{z_2}^F\}.$$

By Lemma C.4.7, there is a unique pair $(e_{\mathbf{opt}_0}, e_0) \in \mathit{GenContracts}\,({e_2}^F)$ satisfying the property stated in Lemma C.4.7. We perform case analysis on $e_{\mathbf{opt}_0}$.

Case $e_{\mathbf{opt}_0} = \mathit{Some}\ e_0'$: There exists some D_j such that

$$\mathit{CtrArgOf}(D_j) = {z_1}^F{:}\mathsf{int} \times T_j$$

where $T_j = \{{z_2}^F{:}\tau\langle e_0'\rangle \mid \mathsf{let}\ {z_2}^F = \langle \mathsf{int\ list} \Leftarrow \tau\langle e_0'\rangle\rangle^\ell\ {z_2}^F\ \mathsf{in}\ e_0\,\{F/f^F\}\}$. For any δ', if $\delta'(\langle \tau\langle e\rangle \Leftarrow \mathsf{int\ list}\rangle^\ell\,(v_1 :: v_2)) = D_j$, then by (R_FORGET), (R_DATATYPE), (R_PROD), and (R_BASE) with (E_RED),

$$\begin{aligned}\delta' \vdash \langle \tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell\,(v_1 :: v_2) \longrightarrow^* \\ D_j\langle e\rangle(v_1, \langle\!\langle T_j, \langle \tau\langle e_0'\rangle \Leftarrow \mathsf{int\ list}\rangle^\ell\ v_2\rangle\!\rangle^\ell\,\{e/y^F, v_1/{z_1}^F\}).\end{aligned}$$

Let $e_0'' = e_0'\,\{e/y^F, v_1/{z_1}^F\}$. By Lemmas C.4.5 and C.3.2, we have $\emptyset \vdash e_0'' : T^F$ since $\emptyset \vdash v_1 : \mathsf{int}$ by Lemma C.3.12, and $\emptyset \vdash e : T^F$ from inversion of $\emptyset \vdash \tau\langle e\rangle$. Thus, $x{:}\mathsf{int\ list} \vdash F\ e_0''\ x : \mathsf{bool}$ by Lemma C.3.1, (T_VAR) and (T_APP), and so $\emptyset \vdash \{x{:}\mathsf{int\ list} \mid F\ e_0''\ x\}$ by (WT_REFINE).
Since $e \longrightarrow^* v'$, we have $F\ e_0''\ v_2 \Rrightarrow^* F\ e_0'\,\{v'/y^F, v_1/{z_1}^F\}\ v_2$ by Lemmas C.1.2 and C.1.5 (2). Since $F\ e_0'\,\{v'/y^F, v_1/{z_1}^F\}\ v_2 \longrightarrow^*$ true by Lemma C.4.7, we have $F\ e_0''\ v_2 \longrightarrow^*$ true by Lemma 29 (2). Thus, by (T_EXACT), $\emptyset \vdash v_2 : \{x{:}\mathsf{int\ list} \mid F\ e_0''\ x\}$ since $\emptyset \vdash v_2 : \mathsf{int\ list}$ by Lemma C.3.12. Since $\tau\langle e_0''\rangle \parallel \{x{:}\mathsf{int\ list} \mid F\ e_0''\ x\}$ by (C_DATATYPE) and (C_REFINEL) (noting the compatibility relation is a equivalence one), and $\emptyset \vdash \tau\langle e_0''\rangle$ by (WT_DATATYPE), we have

$$\emptyset \vdash \langle \tau\langle e_0''\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e_0''\ x\}\rangle^\ell\ v_2 : \tau\langle e_0''\rangle$$

by (T_CAST) and (T_APP). By the IH, there exist some δ'' and v_2' such that

$$\delta'' \vdash \langle \tau\langle e_0''\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e_0''\ x\}\rangle^\ell\ v_2 \longrightarrow^*\ v_2'$$

and δ'' is an extension of δ, and v_2' is obtained by replacing data constructor $[\,]$ and $(::)$ of which v_2 consists with D and one of $\overline{D_i}^i$, respectively. Let $\delta''' = \{\langle \tau\langle e\rangle \Leftarrow \mathsf{int\ list}\rangle^\ell\,(v_1 :: v_2) \mapsto D_j\} \uplus \delta''$. Then,

$$\begin{aligned}\delta''' \vdash \langle \tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell\,(v_1 :: v_2) \longrightarrow^* \\ D_j\langle e\rangle(v_1, \langle\!\langle T_j\,\{e/y^F, v_1/{z_1}^F\}, v_2'\rangle\!\rangle^\ell).\end{aligned}$$

Since $\emptyset \vdash v_2' : \tau\langle e_0''\rangle$ by Theorem 9, we have $\emptyset \vdash \langle \mathsf{int\ list} \Leftarrow \tau\langle e_0''\rangle\rangle^\ell\, v_2' : \mathsf{int\ list}$ by (T_CAST) and (T_APP). By Lemma C.4.6, we have $\langle \mathsf{int\ list} \Leftarrow \tau\langle e_0''\rangle\rangle^\ell\, v_2' \longrightarrow^* v_2$ since δ is trivial. Since $e_0\,\{F/f^F, v'/y^F, v_1/{z_1}^F, v_2/{z_2}^F\} \longrightarrow^* \mathsf{true}$ by Lemma C.4.7, we have $e_0\,\{F/f^F, e/y^F, v_1/{z_1}^F, v_2/{z_2}^F\} \longrightarrow^* \mathsf{true}$ by Lemmas C.1.2 and C.1.5 (2) and Lemma 29 (2). Thus,

$$(\mathsf{let}\ {z_2}^F = \langle \mathsf{int\ list} \Leftarrow \tau\langle e_0'\rangle\rangle^\ell\, {z_2}^F\ \mathsf{in}\ e_0\,\{F/f^F\})\,\{e/y^F, v_1/{z_1}^F, v_2'/{z_2}^F\} \longrightarrow^* \mathsf{true}.$$

Therefore, by (R_CHECK) and (R_OK) with (E_RED) and Lemma C.4.8,

$$\delta''' \vdash \langle \tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell\,(v_1 :: v_2) \longrightarrow^* D_j\langle e\rangle(v_1, v_2').$$

Case $e_{\mathbf{opt}_0} = \mathit{None}$: There exists some D_j such that

$$\mathit{CtrArgOf}(D_j) = {z_1}^F{:}\mathsf{int} \times T_j$$

where $T_j = \{{z_2}^F{:}\mathsf{int\ list} \mid e_0\,\{F/f^F\}\}$. Let $\delta' = \delta \uplus \{\langle \tau\langle e\rangle \Leftarrow \mathsf{int\ list}\rangle^\ell\,(v_1 :: v_2) \mapsto D_j\}$. By (R_FORGET), (R_DATATYPE), (R_PROD), (R_BASE) with (E_RED),

$$\delta' \vdash \langle \tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell\,(v_1 :: v_2) \longrightarrow^* D_j\langle e\rangle(v_1, \langle\!\langle T_j, \langle \mathsf{int\ list} \Leftarrow \mathsf{int\ list}\rangle^\ell\, v_2\rangle\!\rangle^\ell\,\{e/y^F, v_1/{z_1}^F\}).$$

Since $\langle \mathsf{int\ list} \Leftarrow \mathsf{int\ list}\rangle^\ell\, v_2 \longrightarrow^* v_2$ by (R_DATATYPEMONO), we have

$$\delta' \vdash \langle \tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell\,(v_1 :: v_2) \longrightarrow^* D_j\langle e\rangle(v_1, \langle T_j, e_0\,\{F/f^F, v_2/{z_2}^F\}, v_2\rangle^\ell\,\{e/y^F, v_1/{z_1}^F\})$$

by (E_RED)/(R_CHECK). Since $e_0\,\{F/f^F, v'/y^F, v_1/{z_1}^F, v_2/{z_2}^F\} \longrightarrow^* \mathsf{true}$ by Lemma C.4.7, we have $e_0\,\{F/f^F, e/y^F, v_1/{z_1}^F, v_2/{z_2}^F\} \longrightarrow^* \mathsf{true}$ by Lemmas C.1.2 and C.1.5 (2) and Lemma 29 (2). Thus, by (E_RED)/(R_OK) and Lemma C.4.8,

$$\delta' \vdash \langle \tau\langle e\rangle \Leftarrow \{x{:}\mathsf{int\ list} \mid F\ e\ x\}\rangle^\ell\,(v_1 :: v_2) \longrightarrow^* D_j\langle e\rangle(v_1, v_2).$$

□

Lemma C.4.10. *Let F be a translatable function, e be a subterm of ${e_2}^F$, and σ be a simultaneous substitution including $\{F/f^F, e'/y^F, v_1/{z_1}^F, v_2/{z_2}^F\}$. If $(e_{\mathbf{opt}_0}, e_0) \in \mathit{GenContracts}(e)$ and $\sigma(e_0) \longrightarrow^* \mathsf{true}$ and $e_{\mathbf{opt}_0} = \mathit{Some}\ e''$ implies $F\,\sigma(e'')\,v_2 \longrightarrow^* \mathsf{true}$ for any e'', then $\sigma(e) \longrightarrow^* \mathsf{true}$.*

Proof. By structural induction on e.

Case $e = \mathsf{true}$: Obvious.

Case false: Contradictory; $\mathit{GenContracts}(\mathsf{false}) = \emptyset$.

Case $e = \mathsf{if}\ f^F\ e''\ {z_2}^F\ \mathsf{then}\ e_2'\ \mathsf{else}\ e_3'$ where $\mathsf{FV}(e'') \subseteq \{y^F, {z_1}^F\}$: There are two cases which we have to consider by case analysis on e_0.

Case $e_0 = e_2'$: Since $e_{\mathbf{opt}_0} = \mathit{Some}\, e''$, we have $F\,\sigma(e'')\,v_2 \longrightarrow^* \mathsf{true}$. Thus, $\sigma(\mathsf{if}\, {f^F\ e''\ z_2}^F\, \mathsf{then}\, e_0\, \mathsf{else}\, e_3') \longrightarrow^* \sigma(e_0) \longrightarrow^* \mathsf{true}$.

Case $e_0 = \mathsf{if}\, {f^F\ e''\ z_2}^F\, \mathsf{then}\, \mathsf{false}\, \mathsf{else}\, e_3''$ where $(e_{\mathbf{opt}_0}, e_3'') \in \mathit{GenContracts}\,(e_3')$: Since $\sigma(e_0) \longrightarrow^* \mathsf{true}$, we find that $F\,\sigma(e'')\,v_2 \longrightarrow^* \mathsf{false}$ and $\sigma(e_3'') \longrightarrow^* \mathsf{true}$. Since $(e_{\mathbf{opt}_0}, e_3'') \in \mathit{GenContracts}\,(e_3')$, we have $\sigma(e_3') \longrightarrow^* \mathsf{true}$ by the IH. Thus, $\sigma(\mathsf{if}\, {f^F\ e''\ z_2}^F\, \mathsf{then}\, e_2'\, \mathsf{else}\, e_3') \longrightarrow^* \mathsf{true}$.

Case $e = \mathsf{if}\, e_1'\, \mathsf{then}\, e_2'\, \mathsf{else}\, e_3'$ where $e_1' \neq {f^F\ e''\ z_2}^F$ for any e'' such that $\mathsf{FV}\,(e'') \subseteq \{y^F, {z_1}^F\}$: There are two cases which we have to consider by case analysis on e_0.

Case $e_0 = \mathsf{if}\, e_1'\, \mathsf{then}\, e_2''\, \mathsf{else}\, \mathsf{false}$ where $(e_{\mathbf{opt}_0}, e_2'') \in \mathit{GenContracts}\,(e_2')$: Since $\sigma(e_0) \longrightarrow^* \mathsf{true}$, we find that $\sigma(e_1') \longrightarrow^* \mathsf{true}$ and $\sigma(e_2'') \longrightarrow^* \mathsf{true}$. Since $(e_{\mathbf{opt}_0}, e_2'') \in \mathit{GenContracts}\,(e_2')$, we have $\sigma(e_2') \longrightarrow^* \mathsf{true}$ by the IH. Thus, $\sigma(\mathsf{if}\, e_1'\, \mathsf{then}\, e_2'\, \mathsf{else}\, e_3') \longrightarrow^* \mathsf{true}$.

Case $e_0 = \mathsf{if}\, e_1'\, \mathsf{then}\, \mathsf{false}\, \mathsf{else}\, e_3''$ where $(e_{\mathbf{opt}_0}, e_3'') \in \mathit{GenContracts}\,(e_3')$: Since $\sigma(e_0) \longrightarrow^* \mathsf{true}$, we find that $\sigma(e_1') \longrightarrow^* \mathsf{false}$ and $\sigma(e_3'') \longrightarrow^* \mathsf{true}$. Since $(e_{\mathbf{opt}_0}, e_3'') \in \mathit{GenContracts}\,(e_3')$, we have $\sigma(e_3') \longrightarrow^* \mathsf{true}$ by the IH. Thus, $\sigma(\mathsf{if}\, e_1'\, \mathsf{then}\, e_2'\, \mathsf{else}\, e_3') \longrightarrow^* \mathsf{true}$.

Case $e = \mathsf{match}\, e_0'\, \mathsf{with}\, \overline{C_i\, x_i \to e_i'}^{\,i \in \{1,\ldots,n\}}$: For some j, we have $e_0 = \mathsf{match}\, e_0'\, \mathsf{with}\, C_j\, x_j \to e_j'' \mid \overline{C_i\, x_i \to \mathsf{false}}^{\,i \in \{1,\ldots,n\} \setminus \{j\}}$ where $(e_{\mathbf{opt}_0}, e_j'') \in \mathit{GenContracts}\,(e_j')$. Since $\sigma(e_0) \longrightarrow^* \mathsf{true}$, we have $\sigma(e_0') \longrightarrow^* C_j\langle e''\rangle v'$ and $\sigma(e_j'')\,\{v'/x_j\} \longrightarrow^* \mathsf{true}$ for some e'' and v'. By the IH, $\sigma(e_j')\,\{v'/x_j\} \longrightarrow^* \mathsf{true}$. Thus, $\sigma(\mathsf{match}\, e_0'\, \mathsf{with}\, \overline{C_i\, x_i \to e_i'}^{\,i \in \{1,\ldots,n\}}) \longrightarrow^* \mathsf{true}$.

Case otherwise: Obvious since $\mathit{GenContracts}\,(e) = \{(\mathit{None}, e)\}$. □

Lemma C.4.11. *Let F be a translatable function and δ be a trivial constructor choice function. If $v \downarrow_\tau$ and $\emptyset \vdash \langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^\ell\, v : \mathsf{int\ list}$, then $F\, e\, (\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^\ell\, v) \longrightarrow^* \mathsf{true}$.*

Proof. By structural induction on v. Suppose that

$$\mathit{Trans}\,(F) = \mathsf{type}\,\tau\,\langle y^F{:}T^F\rangle = D \parallel [\,] : \{z{:}\mathsf{unit} \mid {e_1}^F\} \mid \overline{D_i \parallel (::) : {z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}T_i \mid e_i\}}^{\,i}.$$

By Lemmas C.4.3 and C.3.10 and (T-Conv), we have $\emptyset \vdash v : \tau\langle e\rangle$. By Lemmas C.3.13 (4) and C.3.12, there are two cases which we have to consider by case analysis on v.

Case $v = D\langle e'\rangle v'$: Since $v \downarrow_\tau$, $e' \longrightarrow^* v''$ for some v''. By Lemmas C.3.12 and C.3.6, we have $\emptyset \vdash v' : \{z{:}\mathsf{unit} \mid {e_1}^F\,\{e'/y^F\}\}$ and $e' \equiv e$. By Theorem 9, we find that ${e_1}^F\,\{e'/y^F, v'/z\} = {e_1}^F\,\{e'/y^F\} \longrightarrow^* \mathsf{true}$. Since $\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^\ell\, D\langle e'\rangle v' \longrightarrow^* [\,]$ by Lemma C.4.6, we have

$$\begin{array}{rll}
 & F\, e\, (\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^\ell\, v) & \\
\equiv & F\, e'\, (\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^\ell\, v) & \text{(by Lemmas C.1.1 and C.1.5 (3))} \\
\longrightarrow^* & F\, v''\, (\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^\ell\, v) & \\
\longrightarrow^* & F\, v''\, [\,] & \\
\longrightarrow^* & {e_1}^F\,\{v''/y^F\} & \\
\equiv & {e_1}^F\,\{e'/y^F\}. & \text{(by Lemmas C.1.2, C.1.5 (3) and C.1.1)}
\end{array}$$

Thus, by Lemma C.2.25 (2),

$$F\ e\ (\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\ v) \longrightarrow^{*} \mathsf{true}.$$

Case $v = D_j\langle e'\rangle v'$: By definition of *Trans*, there is a unique pair $(e_{\mathbf{opt}_0}, e_0) \in \mathit{GenContracts}\,({e_2}^F)$ such that $\mathit{CtrArgOf}\,(D_j)$ is constructed from the pair. By case analysis on $e_{\mathbf{opt}_0}$.

Case $e_{\mathbf{opt}_0} = \mathit{Some}\ e_0'$: We have

$$\mathit{CtrArgOf}\,(D_j) = {z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}\tau\langle e_0'\rangle \mid \mathsf{let}\ {z_2}^F = \langle \mathsf{int\ list} \Leftarrow \tau\langle e_0'\rangle\rangle^{\ell}\ {z_2}^F\ \mathsf{in}\ e_0\,\{F/f^F\}\}.$$

By Lemmas C.3.12, C.3.13 (3), C.3.11 and C.3.6, we have $v' = (v_1, v_2)$ and $\emptyset \vdash v_1 : \mathsf{int}$ and $\emptyset \vdash v_2 : \{{z_2}^F{:}\tau\langle e_0'\rangle \mid \mathsf{let}\ {z_2}^F = \langle \mathsf{int\ list} \Leftarrow \tau\langle e_0'\rangle\rangle^{\ell}\ {z_2}^F\ \mathsf{in}\ e_0\,\{F/f^F\}\}\,\{e'/y^F, v_1/{z_1}^F\}$ and $e \equiv e'$ for some v_1 and v_2. By Lemma C.3.14, we have $\emptyset \vdash e : T^F$. Since $y^F{:}T^F, {z_1}^F{:}\mathsf{int} \vdash e_0' : T^F$ by Lemma C.4.5, we have $\emptyset \vdash e_0'\,\{e/y^F, v_1/{z_1}^F\} : T^F$. Since $\emptyset \vdash \tau\langle e_0'\rangle\,\{e/y^F, v_1/{z_1}^F\}$ by Theorem 10 and Lemma C.3.2 (2) and (T_FORGET), we have

$$\emptyset \vdash v_2 : \tau\langle e_0'\rangle\,\{e/y^F, v_1/{z_1}^F\}$$

by Lemma C.1.5 (3), (T_FORGET), and (T_CONV). Thus, we have $\emptyset \vdash \langle \mathsf{int\ list} \Leftarrow \tau\langle e_0'\,\{e/y^F, v_1/{z_1}^F\}\rangle\rangle^{\ell}\ v_2 : \mathsf{int\ list}$ by (T_FORGET), (T_CAST) and (T_APP). By Lemma C.4.6, there exists some v_2' such that

$$\langle \mathsf{int\ list} \Leftarrow \tau\langle e_0'\,\{e/y^F, v_1/{z_1}^F\}\rangle\rangle^{\ell}\ v_2 \longrightarrow^{*} v_2'.$$

By the IH, we have

$$F\ e_0'\,\{e/y^F, v_1/{z_1}^F\}\ (\langle \mathsf{int\ list} \Leftarrow \tau\langle e_0'\,\{e/y^F, v_1/{z_1}^F\}\rangle\rangle^{\ell}\ v_2) \longrightarrow^{*} \mathsf{true}.$$

Thus, there exists some v_0' such that $e_0'\,\{e/y^F, v_1/{z_1}^F\} \longrightarrow^{*} v_0'$ and $F\ v_0'\ v_2' \longrightarrow^{*} \mathsf{true}$. Since $F\ e_0'\,\{e/y^F, v_1/{z_1}^F\}\ v_2' \Rrightarrow^{*} F\ v_0'\ v_2'$ by Lemmas C.1.2 and C.1.5 (2), we have

$$F\ e_0'\,\{e/y^F, v_1/{z_1}^F\}\ v_2' \longrightarrow^{*} \mathsf{true}$$

by Lemma 29. By applying Lemma C.3.18 to v_2, we have $e_0\,\{F/f^F, e'/y^F, v_1/{z_1}^F, v_2'/{z_2}^F\} \longrightarrow^{*} \mathsf{true}$. Thus, by Lemmas C.1.5 (3) and C.2.25, we have

$$e_0\,\{F/f^F, e/y^F, v_1/{z_1}^F, v_2'/{z_2}^F\} \longrightarrow^{*} \mathsf{true}.$$

By Lemma C.4.10,

$${e_2}^F\,\{F/f^F, e/y^F, v_1/{z_1}^F, v_2'/{z_2}^F\} \longrightarrow^{*} \mathsf{true}.$$

Since $e' \longrightarrow^{*} v''$ for some v'' from $v\downarrow_{\tau}$, we have $v'' \equiv e$. By Lemmas C.1.5 (3) and C.2.25,

$${e_2}^F\,\{F/f^F, v''/y^F, v_1/{z_1}^F, v_2'/{z_2}^F\} \longrightarrow^{*} \mathsf{true}.$$

Thus,

$$\begin{array}{ll} & F\ e'\ (\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\ D_j\langle e'\rangle v') \\ \longrightarrow^* & F\ v''\ (\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\ D_j\langle e'\rangle v') \\ \longrightarrow^* & F\ v''\ (v_1 :: (\langle \mathsf{int\ list} \Leftarrow \tau\langle e_0'\,\{e/y^F, {v_1}/{z_1}^F\}\rangle\rangle^{\ell}\ v_2)) \\ \longrightarrow^* & F\ v''\ (v_1 :: v_2') \\ \longrightarrow^* & {e_2}^F\,\{F/f^F, v''/y^F, {v_1}/{z_1}^F, {v_2'}/{z_2}^F\} \\ \longrightarrow^* & \mathsf{true}. \end{array}$$

Case $e_{\mathbf{opt}_0} = \mathit{None}$: We have $\mathit{CtrArgOf}(D_j) = {z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}\mathsf{int\ list} \mid e_0\,\{F/f^F\}\}$. By Lemmas C.3.12, C.3.13 (3), C.3.11 and C.3.6, we have $\emptyset \vdash e' : T^F$ and $v' = (v_1, v_2)$ and $\emptyset \vdash v_1 : \mathsf{int}$ and $\emptyset \vdash v_2 : \{{z_2}^F{:}\mathsf{int\ list} \mid e_0\,\{F/f^F\}\}\,\{e'/y^F, {v_1}/{z_1}^F\}$ for some v_1 and v_2. By Lemma C.3.18, $e_0\,\{F/f^F, e'/y^F, {v_1}/{z_1}^F, {v_2}/{z_2}^F\} \longrightarrow^* \mathsf{true}$. By Lemma C.4.10, we have ${e_2}^F\,\{F/f^F, e'/y^F, {v_1}/{z_1}^F, {v_2}/{z_2}^F\} \longrightarrow^* \mathsf{true}$. Since $e' \longrightarrow^* v''$ for some v'' from $v\downarrow_\tau$, we have ${e_2}^F\,\{F/f^F, v''/y^F, {v_1}/{z_1}^F, {v_2}/{z_2}^F\} \longrightarrow^* \mathsf{true}$ by Lemmas C.1.2 and C.1.5 (2) and Lemma 29 (1). Thus,

$$F\ e'\ (\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\ D_j\langle e'\rangle v') \longrightarrow^* F\ v''\ (v_1 :: v_2) \longrightarrow^* \mathsf{true}.$$

□

Lemma C.4.12 (From Datatypes to Refinement Types)**.** *Suppose that*

$$\mathit{Trans}\,(F) = \mathsf{type}\,\tau\,\langle y^F{:}T^F\rangle = D \parallel [\,] : \{z{:}\mathsf{unit} \mid {e_1}^F\} \mid \overline{D_i \parallel (::) : {z_1}^F{:}\mathsf{int} \times \{{z_2}^F{:}T_i \mid e_i\}}^{\,i}.$$

Let δ be a trivial constructor choice function.

If $v\downarrow_\tau$ and $\emptyset \vdash v : \tau\langle e\rangle$, then $\langle\{x{:}\mathsf{int\ list} \mid F\ e\ x\} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\ v \longrightarrow^* v'$ for some v' obtained by replacing data constructor D and D_i in v with $[\,]$ and $(::)$, respectively.

Proof. Since $\emptyset \vdash \tau\langle e\rangle$ Lemma C.3.14 and $\mathsf{int\ list} \parallel \tau\langle e\rangle$, we have $\emptyset \vdash \langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\ v : \mathsf{int\ list}$ by (T_CAST) and (T_APP). By Lemma C.4.6, $\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\ v \longrightarrow^* v'$ for some v' which satisfies the property in the statement above. By Lemma C.4.11, we have $F\ e\ (\langle \mathsf{int\ list} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\ v) \longrightarrow^* \mathsf{true}$. Thus, letting v'' be a value such that $e \longrightarrow^* v''$, we find that $F\ v''\ v' \longrightarrow^* \mathsf{true}$. By Lemmas C.1.2 and C.1.5 (2) and Lemma 29 (2), $F\ e\ v' \longrightarrow^* \mathsf{true}$. Thus, by (R_PRECHECK) and (R_OK) with (E_RED),

$$\langle\{x{:}\mathsf{int\ list} \mid F\ e\ x\} \Leftarrow \tau\langle e\rangle\rangle^{\ell}\ v \longrightarrow^* v'.$$

□