CPS Transformation with Affine Types for Call-By-Value Implicit Polymorphism

TARO SEKIYAMA, National Institute of Informatics, Japan
TAKESHI TSUKADA, Chiba University, Japan

Transformation of programs into continuation-passing style (CPS) reveals the notion of continuations, enabling many applications such as control operators and intermediate representations in compilers. Although type preservation makes CPS transformation more beneficial, achieving type-preserving CPS transformation for implicit polymorphism with call-by-value (CBV) semantics is known to be challenging. We identify the difficulty in the problem that we call scope intrusion. To address this problem, we propose a new CPS target language Λ^{open} that supports two additional constructs for polymorphism: one only binds and the other only generalizes type variables. Unfortunately, their unrestricted use makes Λ^{open} unsafe due to undesired generalization of type variables. We thus equip Λ^{open} with affine types to allow only the type-safe generalization. We then define a CPS transformation from Curry-style CBV System F to type-safe Λ^{open} and prove that the transformation is meaning and type preserving. We also study parametricity of Λ^{open} as it is a fundamental property of polymorphic languages and plays a key role in applications of CPS transformation. To establish parametricity, we construct a parametric, step-indexed Kripke logical relation for Λ^{open} and prove that it satisfies the Fundamental Property as well as soundness with respect to contextual equivalence.

CCS Concepts: • **Mathematics of computing** → **Lambda calculus**; • **Software and its engineering** → **Functional languages**; **Control structures**; **Polymorphism**.

Additional Key Words and Phrases: continuation-passing style, polymorphism, affine types, parametricity

ACM Reference Format:
Taro Sekiyama and Takeshi Tsukada. 2021. CPS Transformation with Affine Types for Call-By-Value Implicit Polymorphism. *Proc. ACM Program. Lang.* 5, ICFP, Article 95 (August 2021), 30 pages. https://doi.org/10.1145/3473600

1 INTRODUCTION

1.1 Background: CPS Transformation for Implicit Polymorphism

Transformation of programs into continuation-passing style (CPS) is a technique to reveal the notion of continuations [Reynolds 1993], enabling many applications such as the enforcement of specific evaluation strategies [Plotkin 1975; Reynolds 1972], control operators [Biernacki et al. 2015; Danvy and Filinski 1990; Felleisen et al. 1986; Hillerström et al. 2017; Reynolds 1972], and intermediate representations (IRs) in compilers [Appel 1992; Cong et al. 2019; Fluet and Weeks 2001; Kennedy 2007; Leißa et al. 2015]. Although the study of CPS transformation initially focused on its semantics—specifically, the *meaning-preserving* aspects—with little attention on its typing [Fischer 1972; Plotkin 1975; Reynolds 1972], *type-preserving* CPS transformation has been proven beneficial since its discovery by Meyer and Wand [1985] in the simply typed setting. For example, type-preserving CPS transformation guides type systems for control operators [Danvy and Filinski

Authors' addresses: Taro Sekiyama, tsekiyama@acm.org, National Institute of Informatics, Japan; Takeshi Tsukada, tsukada@math.s.chiba-u.ac.jp, Chiba University, Japan.

2475-1421/2021/8-ART95
https://doi.org/10.1145/3473600

1989; Kameyama and Yonezawa 2008], enables typed CPS IRs in compilers, which are useful for optimization, debugging, and certification of generated assembly/machine code [Morrisett et al. 1999], and is key in certain program verification techniques [Kobayashi 2013]. Addressing more programming features enables more programming languages to avail these advantages.

In this work, we address a long-standing open problem with CPS transformation: whether type-preserving CPS transformation for implicitly polymorphic call-by-value languages is achievable. Harper and Lillibridge [1993a] investigated extensions of simply-typed CPS transformation to two classes of polymorphism—"standard" and "ML-like"—each with call-by-value (CBV) and call-by-name (CBN) variants. "Standard" polymorphism corresponds to what we call *explicit* polymorphism, where type abstraction and application appear explicitly as constructs, as in System F [Girard 1972; Reynolds 1974], and evaluation does not proceed beneath type abstractions. "ML-like" polymorphism is called *implicit*, where evaluation *does* proceed beneath type abstractions (or, equivalently from the semantic perspective, type abstraction and application implicitly appear in source programs as in Curry-style System F [Leivant 1983]). Harper and Lillibridge provided type-preserving CPS transformations for both CBV and CBN explicit polymorphism and for CBN implicit polymorphism. However, their treatment of CBV implicit polymorphism depends on the value restriction [Tofte 1990], i.e., it restricts polymorphic expressions to be values. More significantly, they also showed that achieving meaning- and type-preserving CBV CPS transformation is difficult even for a pure language if no restriction is imposed on implicit polymorphism [Harper and Lillibridge 1993b]; we will discuss a reason of the difficulty in Section 7.

Unfortunately, the dependence on the value restriction in the previous work is not very satisfactory. Since the discovery of the value restriction,[1] many approaches to handling non-value polymorphic expressions in effectful languages have been proposed [Appel and MacQueen 1991; Garrigue 2004; Hoang et al. 1993; Kammar and Pretnar 2017; Leroy and Weis 1991; Sekiyama and Igarashi 2019; Sekiyama et al. 2020]. The dependence on the value restriction prevents languages with these advanced approaches from receiving the benefits of type-preserving CPS transformation. For example, OCaml [Leroy et al. 2020] employs the relaxed value restriction [Garrigue 2004] due to its several advantages over the value restriction. If type-preserving CPS transformation depends on the value restriction, CPS (or CPS-like) compilers of OCaml could not avail benefits such as typed IRs and compiler certification.

This problem is pervasive, specifically in languages with type inference, which involve implicit polymorphism inherently. Further, CBV is also common in languages with imperative (or effectful) constructs. Both strict languages, such as ML-family ones, and non-strict languages, such as Haskell, may involve CBV fragments (e.g., via monads in the latter [Peyton Jones and Wadler 1993]). A study of CPS transformation for CBV implicit polymorphism is valuable for these various languages.

1.2 This Work

Our ultimate, long-term goal is to achieve meaning- and type-preserving CPS transformation for implicitly polymorphic CBV languages that do not employ the value restriction.

As a first step towards it, this work focuses on Curry-style CBV System F, a *pure* language with *unrestricted* implicit polymorphism. To this end, we must implement the evaluation strategy of implicit polymorphism via CPS transformation. However, we discover that this requirement causes the problem that we call *scope intrusion*, which enables type variables to escape from their scope and makes defining type-preserving CPS transformation a challenge.

[1] The value restriction was originally introduced to resolve the unsoundness issue with polymorphic type systems in the presence of mutable references.

To address this challenge, we propose a new polymorphic CPS target language Λ^{open} that supports two additional constructs for polymorphism: *restrictions*, which only bind type variables, and *open type abstractions*, which only generalize the type variables introduced by the former. These constructs enable us to solve scope intrusion, but their unrestricted use makes the language unsafe due to the undesired generalization of type variables. Specifically, without restriction, they would allow the same type variable to be generalized *twice* and, as a result, would break the assumption for type safety that the occurrences of the same type variable should be replaced with the same type. We prevent this undesired generalization using *affine types* [Maraist et al. 1995], which can ensure that type variables are generalized only once.[2] Setting Λ^{open} as the target, we provide meaning- and type-preserving CPS transformation for Curry-style CBV System F.

We further study *parametricity* of Λ^{open}. Parametricity [Reynolds 1983; Wadler 1989], a fundamental property in polymorphic languages, ensures that abstraction is never violated, establishes powerful reasoning principles, and enhances CPS transformation. For example, parametricity enables reasoning about CPS-transformed terms [Thielecke 2003, 2004], verified CPS compilers [Ahmed and Blume 2011], and typed CPS IRs for dependently typed languages [Bowman et al. 2018]. Inspired by these applications (and as a sanity check of Λ^{open}), we construct a parametric, step-indexed Kripke logical relation for Λ^{open} and prove that it satisfies the Fundamental Property. Our logical relation is compatible with term constructors in Λ^{open}, which further enables us to prove that it is sound with respect to contextual equivalence [Morris 1969]. These results could enhance our CPS transformation furthermore, but further investigation is left as future work.

The contributions of this work can be summarized as follows.

- We define a type-safe polymorphic language Λ^{open} with restrictions, open type abstractions, and affine types.
- Employing Λ^{open} as a target, we provide a CPS transformation for Curry-style CBV System F and prove that it is meaning and type preserving.[3]
- We construct a step-indexed Kripke logical relation for Λ^{open} and prove that it satisfies the Fundamental Property and soundness with respect to contextual equivalence.

The rest of this paper is organized as follows. Section 2 reviews Curry-style CBV System F, which is referred to as $\lambda_v^\forall$. Section 3 presents an overview of this work, describing the scope intrusion problem and how we address it. Section 4 defines the CPS target language Λ^{open} and Section 5 provides the meaning- and type-preserving CPS transformation from $\lambda_v^\forall$ to Λ^{open}. Section 6 shows a logical relation for Λ^{open} and its properties. Section 7 discusses related work and Section 8 concludes.

This paper may omit the formal definitions of certain well-known notions, auxiliary lemmas, and detailed proofs. The full definitions, lemmas, and proofs are found in the supplementary material.

2 $\lambda_V^\forall$: CURRY-STYLE CBV SYSTEM F

This section introduces $\lambda_v^\forall$, which is the CBV λ-calculus extended with polymorphic types. The syntax, semantics, and type system of $\lambda_v^\forall$ are standard, shown in Figure 1. The remainder of this section summarizes them to make this paper self-contained.

Syntax. We use the metavariables x, y, z, f, k for variables and α, β, γ for type variables. $\lambda_v^\forall$ is parameterized over constants, ranged over by c, and base types, ranged over by ι. Types, ranged

[2] We decided to use affine types, not linear types, because affine types are more flexible in that, when we extend the source language with control operators, they could enable CPS transformation that allows discarding continuations captured by the control operators, while linear types would require the exact use of the continuations. However, linear types would also help us attain the objective of this study; a target language with linear typing can be defined as Λ^{open}.

[3] In this work, "meaning preservation" implies that the meaning of a whole program is preserved. Considering meaning preservation on expressions, as in previous work on compiler certification [Ahmed and Blume 2011], is left for future work.

Syntax

$$\begin{array}{llll} \text{Variables} & x, y, z, f, k & \text{Type variables} & \alpha, \beta, \gamma \\ \text{Base types} & \iota ::= \mathsf{bool} \mid \mathsf{int} \mid \ldots & \text{Constants} & c ::= \mathsf{true} \mid \mathsf{false} \mid 0 \mid + \mid \ldots \\ \text{Terms} & e ::= x \mid c \mid \lambda x.e \mid e_1\, e_2 & \text{Types} & \tau ::= \alpha \mid \iota \mid \tau_1 \to \tau_2 \mid \forall \alpha.\tau \\ \text{Typing contexts} & \Theta ::= \emptyset \mid \Theta, x:\tau \mid \Theta, \alpha & \text{Values} & w ::= x \mid c \mid \lambda x.e \end{array}$$

Semantics $\boxed{e_1 \longrightarrow_F e_2}$

$$c_1\, c_2 \longrightarrow_F \zeta(c_1, c_2) \qquad (\lambda x.e)\, w \longrightarrow_F e[w/x] \qquad \frac{e_1 \longrightarrow_F e_1'}{e_1\, e_2 \longrightarrow_F e_1'\, e_2} \qquad \frac{e_2 \longrightarrow_F e_2'}{w_1\, e_2 \longrightarrow_F w_1\, e_2'}$$

Type system $\boxed{\vdash \Theta}$ $\boxed{\Theta \vdash e : \tau}$

$$\frac{}{\vdash \emptyset} \qquad \frac{\vdash \Theta \quad \Theta \vdash \tau \quad x \notin dom(\Theta)}{\vdash \Theta, x:\tau} \qquad \frac{\vdash \Theta \quad \alpha \notin dom(\Theta)}{\vdash \Theta, \alpha}$$

$$\frac{\vdash \Theta \quad x:\tau \in \Theta}{\Theta \vdash x : \tau} \qquad \frac{\vdash \Theta}{\Theta \vdash c : ty^{\to}(c)} \qquad \frac{\Theta, x:\tau_1 \vdash e : \tau_2}{\Theta \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Theta \vdash e_1 : \tau_1 \to \tau_2 \quad \Theta \vdash e_2 : \tau_1}{\Theta \vdash e_1\, e_2 : \tau_2} \qquad \frac{\Theta, \alpha \vdash e : \tau}{\Theta \vdash e : \forall \alpha.\tau} \qquad \frac{\Theta \vdash e : \forall \alpha.\tau_2 \quad \Theta \vdash \tau_1}{\Theta \vdash e : \tau_2[\tau_1/\alpha]}$$

Fig. 1. Syntax, semantics, and type system of $\lambda_v^{\forall}$.

over by τ, consist of type variables, base types, function types $\tau_1 \to \tau_2$, and polymorphic types $\forall \alpha.\tau$ (which bind α in τ). Terms, ranged over by e, consist of variables, constants, lambda abstractions $\lambda x.e$ (which bind x in e), and applications $e_1\, e_2$. Variables, constants, and lambda abstractions are also called values, ranged over by w. Typing contexts, ranged over by Θ, are finite sequences of bindings of the form $x:\tau$, which assigns the type τ to the variable x, or α. We write $dom(\Theta)$ for the set of variables and type variables bound by Θ. The notions of free variables and free type variables are defined as usual. We write $ftv(\tau)$ for the set of free type variables in the type τ. A type is closed if and only if it contains no free type variable. We also write $e_1[e_2/x]$ for the term obtained by substituting e_2 for free variable x in e_1 in a capture-avoiding manner. Similarly, $\tau_1[\tau_2/\alpha]$ is the type obtained by applying capture-avoiding type substitution $[\tau_2/\alpha]$ to τ_1. Note that there are no syntactic constructs for type abstraction and type application because they appear only implicitly.

Semantics. The semantics is defined by the evaluation relation $\longrightarrow_F$, which is a binary relation over terms. The first two rules in Figure 1 define reduction. The reduction of a constant application $c_1\, c_2$ depends on the metafunction ζ, which maps pairs of constants to constants. A function application $(\lambda x.e)\, w$ reduces to $e[w/x]$ as usual (β-reduction). It requires the argument w to be a value because this work considers the CBV semantics. The other two rules determine the subterms to be reduced, and indicate that the evaluation proceeds from left to right.

Type system. The type system consists of two judgments. Well-formedness judgment $\vdash \Theta$ states that typing context Θ is well formed, that is, Θ binds the same term and type variable at most once and assigns a well-formed type to a term variable. A type τ is well formed under a typing context Θ, written as $\Theta \vdash \tau$, if and only if Θ binds all the free type variables in τ. Typing judgment $\Theta \vdash e : \tau$ states that e is typed at τ under Θ. All the typing rules are standard. The metafunction $ty^{\to}$ assigns a closed first-order type of the form $\iota_1 \to \ldots \to \iota_n \to \iota_{n+1}$ to each constant. We assume that, for each constant c, the type $ty^{\to}(c)$ is consistent with the denotation given by ζ. More formally, the following holds for any c_1, c_2, and τ: $\zeta(c_1, c_2)$ is well defined and $ty^{\to}(\zeta(c_1, c_2)) = \tau$ if and only if

$ty^{\rightarrow}(c_1) = \iota \rightarrow \tau$ and $ty^{\rightarrow}(c_2) = \iota$ for some ι. A polymorphic type quantifies a type variable that does not occur free in a typing context, and can be instantiated with a well-formed type.

3 OVERVIEW

This section illustrates scope intrusion, which hinders type-preserving CBV CPS transformation for unrestricted implicit polymorphism, and then provides an overview of our approach to it.

3.1 Challenge: Scope Intrusion

In contrast with the simplicity of the definition, CPS transformation involves several factors. To clarify the aspects of CBV CPS transformation that make extension to implicit polymorphism difficult, we begin by reviewing the factorization of CBV CPS transformation.

Danvy [1992] found that CBV CPS transformation can be factorized into three stages: naming intermediate results, sequencing computation, and rendering access to continuations explicit. Sabry and Felleisen [1992] discovered that the first two stages can be expressed in a source language via two reduction rules on source terms: β_{flat}, which names intermediate results, and β_{lift}, which lifts subterms to be evaluated upward.

Let's take a close look at β_{lift}, which is problematic in implicit polymorphism. The rule β_{lift} depends on the notion of *evaluation contexts*, which are a syntactic device to express continuations in operational semantics [Sabry and Felleisen 1992] and identify subterms to be evaluated. Let E be an evaluation context. We write: $E[e]$ for the term obtained by filling the hole $[\,]$ of E with a term e; and $fv(E)$ for the set of free variables in E. Then, β_{lift} is expressed by:

$$E[(\lambda x.e_1)\, e_2] \Longrightarrow (\lambda x.E[e_1])\, e_2 \qquad (\text{if } E \neq [\,] \text{ and } x \notin fv(E))$$

($e_1 \Longrightarrow e_2$ means that e_1 is reduced to e_2). This rule reveals how CPS transformation enforces a specific evaluation order (determined by evaluation contexts here) independently of the evaluation strategy in the target language.[4] For simplicity, let e_2 be a value w_2. Then, β_{lift} reduces $E[(\lambda x.e_1)\, w_2]$ to $(\lambda x.E[e_1])\, w_2$. Both CBV and CBN reduce the resulting term to the same term $E[e_1][w_2/x]$ because neither of them evaluates under the λ nor evaluates the argument value w_2 further. In this way, β_{lift} enables the redex determined by an evaluation context to be reduced first in both CBV and CBN. CPS transformation inherently incorporates both β_{lift} and β_{flat} [Danvy and Hatcliff 1992; Sabry and Felleisen 1992]. Thus, β_{lift} should be type preserving for CPS transformation to be so.

Unfortunately, naive support for implicit polymorphism makes β_{lift} non-type-preserving. To demonstrate this, we make type abstraction and application explicit as in System F: we write $\Lambda\alpha.e$ and $e\,\tau$ for explicit type abstractions and applications, respectively. Explicit polymorphism with these constructs can simulate the semantics of implicit polymorphism by allowing evaluation contexts in which the holes occur under binders Λ of type variables. For example, $\Lambda\alpha.[\,]$ is a valid evaluation context and, as it allows reduction under the Λ, a term $\Lambda\alpha.((\lambda x.x)\,(\lambda x.x))$ can be reduced to $\Lambda\alpha.\lambda x.x$ by β-reduction. Inspired by Harper and Lillibridge [1993a], we call such polymorphism *ML-like* explicit polymorphism.

Consider β_{lift} in ML-like explicit polymorphism. Because $E[\Lambda\alpha.[\,]]$ is a valid evaluation context, β_{lift} would allow the following transformation:

$$E[\Lambda\alpha.((\lambda x.e_1)\, e_2)] \Longrightarrow (\lambda x.E[\Lambda\alpha.e_1])\, e_2 \qquad (\text{if } x \notin fv(E))\,.$$

This transformation *intrudes* the scope of the type variable α. It is necessary because the evaluation context E expects the hole to be filled with a polymorphic term. However, the intrusion causes two typing issues in the resulting term. The first issue is that the resulting term does not bind the type variable α in e_2. The second, more serious issue is that, while the original term binds

[4] The other rule β_{flat} is also important for applying β_{lift} to weak head normal forms within lambda abstractions.

the variable x *inside* the scope of α, the resulting term binds it *outside* the scope of α. This change may make it impossible for x to have the type required for the resulting term to be well typed. For example, let $e_1 = \mathsf{id}\,(\alpha \to \alpha)\,x$, where id is a polymorphic identity function of the type $\forall\beta.\beta \to \beta$. This term is well typed if and only if x is assigned the type $\alpha \to \alpha$. The subterm $\Lambda\alpha.((\lambda x.e_1)\,e_2)$ in the original term can be well typed (if e_2 is of $\alpha \to \alpha$), because x is bound within the scope of α and, thus, it can be assigned the type $\alpha \to \alpha$. However, the subterm $\lambda x.E[\Lambda\alpha.e_1]$ in the resulting term should be ill typed because the type of x cannot reference α.

Note that this scope intrusion problem does not occur if the source language employs the value restriction or "standard" explicit polymorphism [Harper and Lillibridge 1993a], where all type abstractions are values and evaluation does not proceed beneath type abstractions, because they can exclude the evaluation contexts that cause the scope intrusion problem, such as $E[\Lambda\alpha.\,[\,]\,]$. Harper and Lillibridge [1993a] found that a variant of ML-like explicit polymorphism with CBN semantics can also exclude the problematic evaluation contexts. However, obtaining such a variant for CBV is difficult because evaluation beneath type abstractions appears unavoidable there.

3.2 Our Solution

To resolve the scope intrusion problem, we propose new constructs for polymorphism. The constructs enable type-preserving CBV CPS transformation, but their unrestricted use is unsafe. We restrict their use to be safe by introducing affine types. In what follows, we present an overview of these ideas. We use the metavariable M for terms in a language with the new constructs for polymorphism to distinguish them from those in System F.

3.2.1 New Constructs: Restriction and Open Type Abstraction. In this section, we informally introduce two constructs: *restriction* and *open type abstraction*. A restriction $\nu\alpha.\,M$ simply introduces a fresh type variable α and binds it in the term M. An *open type abstraction* $\Lambda^\circ\langle\,\alpha, M\,\rangle$ generalizes (but does not bind) the type variable α, which may occur free in M. For example, an open type abstraction $\Lambda^\circ\langle\,\alpha, \lambda x.x\,\rangle$ can be given type $\forall\alpha.\alpha \to \alpha$ by assigning the type variable α to x and generalizing it; note that α must be bound by an outer ν. Intuitively, an open type abstraction $\Lambda^\circ\langle\,\alpha, M\,\rangle$ is evaluated to $\Lambda^\circ\langle\,\alpha, V\,\rangle$ where V is the evaluation result of the body M, and then to $\Lambda\alpha.V$ (thus, explicit type abstractions are still supported). Open type abstractions can generalize only type variables bound by ν and not those bound by Λ.

The benefit of using the new constructs is recognized in generalizing type variables in open terms (i.e., terms containing free variables). For example, consider a term $\lambda x.\Lambda^\circ\langle\,\alpha, \mathsf{id}\,(\alpha \to \alpha)\,x\,\rangle$ (recall that id is of $\forall\beta.\beta \to \beta$). This function can be given the type $(\alpha \to \alpha) \to \forall\alpha.(\alpha \to \alpha)$ by assigning the type $\alpha \to \alpha$ to x because the body $\mathsf{id}\,(\alpha \to \alpha)\,x$ has $\alpha \to \alpha$ and the open type abstraction generalizes α in the body. Note that open type abstractions can generalize type variables in the types assigned to variables even if the variables are bound outside the open type abstractions. By contrast, as seen in Section 3.1, a term $\lambda x.\Lambda\alpha.\mathsf{id}\,(\alpha \to \alpha)\,x$ with ordinary type abstraction cannot have such a type, because the Λ determines the scope of α and x is bound outside of it.

Now, let's see how the new constructs resolve the scope intrusion problem. First, we express a context $\Lambda\alpha.\,[\,]$ in a source program by $\nu\alpha.\,\Lambda^\circ\langle\,\alpha,\ [\,]\ \rangle$ in the target language of transformation. Thus, the problematic evaluation context $E[\Lambda\alpha.\,[\,]\,]$ is expressed by $E[\nu\alpha.\,\Lambda^\circ\langle\,\alpha,\ [\,]\ \rangle]$. Then, we can give CPS transformation such that a function application $(\lambda x.M_1)\,M_2$ placed in the hole is lifted in a type-safe manner. It consists of two steps.

$$E[\nu\alpha.\,\Lambda^\circ\langle\,\alpha, (\lambda x.M_1)\,M_2\,\rangle] \Longrightarrow \nu\alpha.\,E[\Lambda^\circ\langle\,\alpha, (\lambda x.M_1)\,M_2\,\rangle] \Longrightarrow \nu\alpha.\,((\lambda x.E[\Lambda^\circ\langle\,\alpha, M_1\,\rangle])\,M_2)$$

where we suppose that α and x do not occur free in E. The first step extrudes the ν onto the top of the program whereas the second intrudes the Λ° under the λ by considering the evaluation

context $E[\Lambda^\circ\langle\,\alpha,\ [\,]\,\rangle]$. We can observe that this new transformation still enforces the evaluation order determined by evaluation contexts, as follows. The evaluation of a restriction term $\nu\alpha.\,M$ proceeds by simply evaluating its body M. Thus, the evaluation of the resulting term begins with the reduction of the redex $(\lambda x.\,\ldots)\,M_2$ if M_2 is a value.

This transformation is type preserving. Suppose that the original term is well typed. Then, it is found that M_2 has some type τ_2, M_1 has some type τ_1 under the type assignment that gives x the type τ_2, and E requires the hole to be filled with a term of the polymorphic type $\forall\alpha.\tau_1$ (in general, restriction $\nu\alpha.\,M$ has the same type as its body M). In addition, $\lambda x.M_1$ and M_2 must be placed in a context that binds α. Considering the resulting term, the ν at the top-level binds α. Thus, M_2 can be of the type τ_2. The application $(\lambda x.\,\ldots)\,M_2$ in the resulting term allows (and requires) the bound variable x to have the same type as M_2; thus, x is assigned τ_2. This type assignment allows M_1 to be of the type τ_1. Then, the open type abstraction $\Lambda^\circ\langle\,\alpha, M_1\,\rangle$ has the polymorphic type $\forall\alpha.\tau_1$ by generalizing α in the type τ_1 of M_1. Thus, it can be used to fill the hole of E. The type of the entire resulting term is the same as that of the original term, which is given by E. Thus, the transformation is type preserving. Section 5 defines a CBV CPS transformation based on this idea.

3.2.2 Restricting Generalization by Affine Types. Although open type abstractions are key to type-preserving CBV CPS transformation for implicit polymorphism, their unrestricted use allows type variables to be generalized more than once, which results in allowing a term that is well typed but gets stuck. For example, let us consider a term $M \stackrel{\text{def}}{=} \Lambda^\circ\langle\,\alpha, \lambda x.\Lambda^\circ\langle\,\alpha, \lambda y.x\,\rangle\,\rangle$. This term generalizes the type variable α twice and would have the type $\forall\alpha.\alpha \to \forall\alpha.(\alpha \to \alpha)$ by assigning α to both x and y. This type indicates that application of M to any value returns a polymorphic identity function. Thus, for example, we can expect that a term $(M\,\mathsf{int}\,0)\,\mathsf{bool}\,\mathsf{true}$ is of the type bool and returns a Boolean value. However, considering the underlying untyped term of M (i.e., eliminating the Λ°s), we can find that it would return the first argument value, integer 0. This result indicates that we can provide a program that is well typed but gets stuck.

In general, the type checking with the standard type abstraction mechanism as in System F assumes that a type variable is only associated with a single type within its scope by instantiation. However, the unrestricted use of open type abstractions breaks this assumption: it causes a type variable to be associated with multiple different types.

Our solution to this problem is to constrain the use of open type abstractions so that every type variable is generalized at most once. We enforce this constraint using *affine types* [Maraist et al. 1995], which are a general type-based technique to ensure that a value is not used more than once. Our type system manages both values and type variables in an affine manner to prevent more than one open type abstraction from generalizing the same type variable.

4 $\Lambda^{\mathbf{OPEN}}$: CPS TARGET LANGUAGE

This section defines the target language Λ^{open} of our CPS transformation. Λ^{open} is a polymorphic λ-calculus with restrictions, open type abstractions, and affine types as sketched in Section 3.2.

4.1 Syntax

Figure 2 presents the syntax of Λ^{open}, where polymorphism is made explicit.

Types, ranged over by A, B, C, and D, include type variables, base types, and polymorphic types as in $\lambda_u^{\forall}$. The other two type constructors originate from linear logic [Girard 1987] (although the present work considers affine typing, not linear typing). An affine function type $A \multimap B$ is given to functions that produce a value of the type B by referring to argument variables of the type A *at most once*. We call variables that can be used at most once *affine* and those that can be used multiple times *unrestricted*. Affine variables can be bound to any value, but unrestricted ones can only be

$$
\begin{array}{llll}
\text{Types} & A, B, C, D & ::= & \alpha \mid \iota \mid \forall\alpha.A \mid A \multimap B \mid !A \\
\text{Terms} & M & ::= & x \mid c \mid \lambda x.M \mid M_1\, M_2 \mid !M \mid \mathsf{let}\, !x = M_1 \,\mathsf{in}\, M_2 \mid \\
 & & & \nu\alpha.\, M \mid \Lambda^{\circ}\langle\, \alpha, M \,\rangle \mid \Lambda\alpha.M \mid M\, A \\
\text{Uses} & \pi & ::= & \mathbf{0} \mid \mathbf{1} \mid \omega \\
\text{Typing contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x :^{\pi} A \mid \Gamma, \alpha^{\pi}
\end{array}
$$

Well-formedness rules $\boxed{\vdash \Gamma}$

$$
\frac{}{\vdash \emptyset} \qquad \frac{\vdash \Gamma \quad \Gamma \vdash A \quad x \notin dom(\Gamma)}{\vdash \Gamma, x :^{\pi} A} \qquad \frac{\vdash \Gamma \quad \alpha \notin dom(\Gamma) \quad \pi \neq \omega}{\vdash \Gamma, \alpha^{\pi}}
$$

Typing rules $\boxed{\Gamma \vdash M : A}$

$$
\frac{\vdash \Gamma}{\Gamma \vdash x : \Gamma(x)}\ \text{T\_Var} \qquad \frac{\vdash \Gamma}{\Gamma \vdash c : ty(c)}\ \text{T\_Const}
$$

$$
\frac{\Gamma, x :^{\mathbf{1}} A \vdash M : B}{\Gamma \vdash \lambda x.M : A \multimap B}\ \text{T\_Abs} \qquad \frac{\Gamma_1 \vdash M_1 : A \multimap B \quad \Gamma_2 \vdash M_2 : A}{\Gamma_1 + \Gamma_2 \vdash M_1\, M_2 : B}\ \text{T\_App}
$$

$$
\frac{\vdash \Gamma \quad \omega\Gamma \vdash M : A}{\Gamma \vdash !M : !A}\ \text{T\_Bang} \qquad \frac{\Gamma_1 \vdash M_1 : !B \quad \Gamma_2, x :^{\omega} B \vdash M_2 : A}{\Gamma_1 + \Gamma_2 \vdash \mathsf{let}\, !x = M_1 \,\mathsf{in}\, M_2 : A}\ \text{T\_LetBang}
$$

$$
\frac{\Gamma, \alpha^{\mathbf{1}} \vdash M : A \quad \Gamma \vdash A}{\Gamma \vdash \nu\alpha.\, M : A}\ \text{T\_Nu} \qquad \frac{\Gamma \vdash M : \forall\alpha.B \quad \Gamma \vdash A}{\Gamma \vdash M\, A : B[A/\alpha]}\ \text{T\_TApp}
$$

$$
\frac{\Gamma_1, \alpha^{\mathbf{0}}, \Gamma_2 \vdash M : !A}{\Gamma_1, \alpha^{\mathbf{1}}, \Gamma_2 \vdash \Lambda^{\circ}\langle\, \alpha, M \,\rangle : !\forall\alpha.A}\ \text{T\_Gen} \qquad \frac{\vdash \Gamma \quad \omega\Gamma, \alpha^{\mathbf{0}} \vdash M : A}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.A}\ \text{T\_TAbs}
$$

Fig. 2. Syntax and type system of Λ^{open}.

bound to unrestricted values. We deem values affine by default and require values used more than once to be of an "of-course" type. An of-course type $!A$ is given to unrestricted values of the type A.

Terms, ranged over by M, follow those of the linear λ-calculus [Maraist et al. 1995], augmented with the constructs for polymorphism. They contain terms from the λ-calculus (variables, constants, functions, and applications) and terms for operating unrestricted values ($!M$ and $\mathsf{let}\, !x = M_1 \,\mathsf{in}\, M_2$). An unrestricted term $!M$ ensures that the evaluation result of M is unrestricted by restricting M to refer only to unrestricted variables. A term $\mathsf{let}\, !x = M_1 \,\mathsf{in}\, M_2$, where x is bound in M_2, evaluates M_2 after binding the unrestricted variable x to the unrestricted value produced by M_1. The remaining four constructs are for polymorphism. A restriction $\nu\alpha.\, M$ introduces a fresh type variable α and binds it in M. An open type abstraction $\Lambda^{\circ}\langle\, \alpha, M \,\rangle$ generalizes the type variable α that may occur free in M. Note that it does *not* bind α. A type abstraction $\Lambda\alpha.M$ and type application $M\, A$ come from System F. We call type abstractions of this form *closed*.

We introduce certain syntactic notation as in $\lambda_v^{\forall}$. We write $ftv(A)$ for the set of free type variables in the type A. Terms and types obtained by applying capture-avoiding substitution, such as $M_1[M_2/x]$, $A_1[A_2/\alpha]$, and $M_1[A_2/\alpha]$, are defined as usual, except for $M_1[A_2/\alpha]$: the term $M_1[A_2/\alpha]$ is well defined if and only if α is not generalized in M_1 (i.e., M_1 involves no subterm of the form $\Lambda^{\circ}\langle\, \alpha, M' \,\rangle$) or A_2 is a type variable.

4.2 Type System

This section formalizes the type system informally illustrated in Section 3.2. Figure 2 gives the definition of typing contexts and inference rules for the type system.

Typing contexts, ranged over by Γ, are sequences of bindings of variables coupled with types and those of type variables. Unlike in $\lambda_v^\forall$, bindings in Λ^{open} accompany *uses* [Igarashi and Kobayashi 1997; Turner et al. 1995], ranged over by π, which determine the usage of variables and type variables. A use attached to a variable determines how many times the variable can be referenced. Variables with the use $\mathbf{0}$ may never be referenced; those with $\mathbf{1}$ may be only once; and those with ω may be any number of times. A use attached to a type variable determines whether it can be generalized. Type variables with $\mathbf{0}$ may never be generalized and those with $\mathbf{1}$ may be only once. The type system ensures that type variables are not equipped with ω. Uses given to type variables restrict the generalization of the type variables but do not restrict references to them. For example, a type variable α occurs in $(M\,\alpha)\,\alpha$ twice, but this term can be typechecked if M has a type $\forall\beta.\forall\gamma.A$.

The type system of Λ^{open} consists of two judgments: well-formedness judgment $\vdash \Gamma$ and typing judgment $\Gamma \vdash M : A$. The well-formedness rules are identical to the rules in $\lambda_v^\forall$ except that it is ensured that the use ω is never attached to type variables. Figure 2 uses the same notation as in $\lambda_v^\forall$: $dom(\Gamma)$ is the set of variables and type variables bound by the typing context Γ; and $\Gamma \vdash A \stackrel{\text{def}}{=} ftv(A) \subseteq dom(\Gamma)$. We also write Γ_1, Γ_2 for the concatenation of Γ_1 and Γ_2.

Typing rules are in a syntax-directed manner. The rule (T_Var) assigns the type $\Gamma(x)$ to a variable x; we write $\Gamma(x)$ for the type A such that $x :^\pi A \in \Gamma$ for some $\pi \neq \mathbf{0}$. Thus, (T_Var) disallows reference to variables having the use $\mathbf{0}$. The premise $\vdash \Gamma$ ensures that $\Gamma(x)$ is uniquely determined. The rule (T_Const) gives a constant c the type $ty(c)$, which is the same as $ty^{\to}(c)$ except that the type constructor $\to$ is replaced by $\multimap$. The rule (T_App) for applications is standard except for the use of *typing context addition* $+$, which is conventional in some calculi involving linearity [Atkey 2018; Cervesato and Pfenning 1996; Igarashi and Kobayashi 1997; Montagu and Rémy 2009]. It merges the uses of bindings in two typing contexts, defined as follows.

Definition 1 (Typing Context Addition). *The binary operation $+$ on uses is defined as follows:*

$$\mathbf{0} + \pi \stackrel{\text{def}}{=} \pi + \mathbf{0} \stackrel{\text{def}}{=} \pi \qquad \omega + \pi \stackrel{\text{def}}{=} \pi + \omega \stackrel{\text{def}}{=} \mathbf{1} + \mathbf{1} \stackrel{\text{def}}{=} \omega \qquad \textit{(for any } \pi\textit{)}\,.$$

The binary operation $+$ on typing contexts is defined as follows:

$$\begin{array}{lcl} \emptyset + \emptyset & \stackrel{\text{def}}{=} & \emptyset \\ (\Gamma_1, x :^{\pi_1} A) + (\Gamma_2, x :^{\pi_2} A) & \stackrel{\text{def}}{=} & (\Gamma_1 + \Gamma_2), x :^{\pi_1+\pi_2} A \\ (\Gamma_1, \alpha^{\pi_1}) + (\Gamma_2, \alpha^{\pi_2}) & \stackrel{\text{def}}{=} & (\Gamma_1 + \Gamma_2), \alpha^{\pi_1+\pi_2} \quad \textit{(if } \pi_1 + \pi_2 \neq \omega\textit{)}\,. \end{array}$$

The rule (T_App) builds the typing context in the conclusion judgment by merging the typing contexts Γ_1 for function M_1 and Γ_2 for argument M_2. The definition of use addition indicates that variables referenced by both M_1 and M_2 are equipped with the use ω in the conclusion judgment. By contrast, it is impossible for both M_1 and M_2 to generalize the same type variable. If both generalize a type variable α, the typing contexts Γ_1 and Γ_2 would assign the use $\mathbf{1}$ to α. However, such typing contexts cannot be merged because the addition result of the uses for a type variable must not be ω. This is how the type system prevents the same type variable from being generalized more than once. The rule (T_Bang) is for unrestricted terms. As unrestricted values may be duplicated, we prevent them from referring to affine variables and from generalizing affine type variables. The typing context $\omega\Gamma$ is obtained by changing the use $\mathbf{1}$ given to bindings in Γ to $\mathbf{0}$.

Definition 2 (Unrestricted Typing Contexts). *Given Γ, a typing context $\omega\Gamma$ is defined by induction on Γ as follows.*

$$\begin{array}{lcllcl} \omega\emptyset & \stackrel{\text{def}}{=} & \emptyset & \omega(\Gamma, \alpha^\pi) & \stackrel{\text{def}}{=} & \omega\Gamma, \alpha^{\mathbf{0}} \\ \omega(\Gamma, x :^\omega A) & \stackrel{\text{def}}{=} & \omega\Gamma, x :^\omega A & \omega(\Gamma, x :^\pi A) & \stackrel{\text{def}}{=} & \omega\Gamma, x :^{\mathbf{0}} A \quad \textit{(if } \pi \neq \omega\textit{)} \end{array}$$

The premise $\vdash \Gamma$ in (T_Bang) ensures that no type variable is assigned ω in Γ. The rule (T_LetBang) typechecks a term $\mathsf{let}\,!x = M_1\,\mathsf{in}\,M_2$. The variable x will be bound to the evaluation result of M_1. As M_1 is of an of-course type $!B$, the result should be unrestricted. Thus, (T_LetBang) allows M_2 to refer to x multiple times by assigning the use ω to x. The uses of other free variables and free type variables are merged by typing context addition as in (T_App). The rule (T_Nu) is applied to a restriction $\nu\alpha.\,M$. Because type variables may be generalized once, the bound type variable α is assigned the use $\mathbf{1}$. The premise $\Gamma \vdash A$ ensures that α does not leak outside the binder. The rule (T_TApp) for type applications is standard, requiring type arguments to be well formed.

The rule (T_Gen) for open type abstractions $\Lambda^\circ\langle\,\alpha, M\,\rangle$ requires (1) the typing context $\Gamma_1, \alpha^{\mathbf{1}}, \Gamma_2$ in the conclusion judgment to assign the use $\mathbf{1}$ to α so that α can be generalized, and (2) the typing context $\Gamma_1, \alpha^{\mathbf{0}}, \Gamma_2$ in the premise to assign $\mathbf{0}$ to α for preventing generalization of α in M. Further, (T_Gen) states that the abstraction $\Lambda^\circ\langle\,\alpha, M\,\rangle$ is unrestricted. This is necessary for proving type preservation of CPS transformation. Intuitively, in a resulting term of our CPS transformation, a polymorphic term is expressed by an open type abstraction. Because terms in $\lambda_v^\forall$ are unrestricted, the open type abstraction must also evaluate to an unrestricted value. The of-course types assigned by (T_Gen) ensure this requirement, and for consistency, the body M is also required to be unrestricted.

The last rule (T_TAbs) typechecks closed type abstractions. The notable point of this rule is the use of $\omega\Gamma$ in the premise; it means that the body of a closed type abstraction generalizes no type variable bound outside it. The enforcement of this discipline is key to proving subject reduction. To see it, assume that closed type abstractions can generalize type variables bound outside. Then, for example, a closed type abstraction $M \stackrel{\text{def}}{=} \Lambda\alpha.\Lambda^\circ\langle\,\beta, \mathsf{id}\,\alpha\,\rangle$ would be of a type $\forall\alpha.\forall\beta.\alpha \multimap \alpha$ (where id is a term of $\forall\alpha.\alpha \multimap \alpha$). By (T_TApp), type application $M\,\beta$ would be typed at $(\forall\beta.\alpha \multimap \alpha)[\beta/\alpha] = \forall\gamma.\beta \multimap \beta$ for some fresh γ. Subject reduction means that the reduction result of $M\,\beta$ also has the same type as $M\,\beta$. As shown in the next section, $M\,\beta$ is reduced to the term $\Lambda^\circ\langle\,\beta, \mathsf{id}\,\beta\,\rangle$, which is obtained by simply substituting the argument β for α in the body $\Lambda^\circ\langle\,\beta, \mathsf{id}\,\alpha\,\rangle$ of M. However, this resulting term is of $\forall\beta.\beta \multimap \beta$ and cannot have the expected type $\forall\gamma.\beta \multimap \beta$. Thus, subject reduction would not hold if we allow the bodies of closed type abstractions to generalize type variables that are bound outside of them. Conversely, we can avoid the unsatisfactory situation by disallowing it. A key lemma for type substitution, which is formulated by Lemma 2 in Section 4.4, is that, given a type A and a term M of type B, the term $M[A/\alpha]$ has the desired type $B[A/\alpha]$ *if* M generalizes no type variable occurring in A. As the use of $\omega\Gamma$ in (T_TAbs) ensures that closed type abstractions generalize no type variable that occurs in substituted types, we can use this lemma to prove that reduction of type applications is type preserving.

4.3 Semantics

This section gives the call-by-value, small-step semantics of Λ^{open}. We start with introducing new syntactic classes—results, values, evaluation contexts, and extrusion contexts—and then define the reduction and evaluation relations using them. All the definitions are shown in Figure 3.

4.3.1 New Syntax Classes. Results, ranged over by R, are evaluation results, being values possibly enclosed by ν-binders. Values, ranged over by V, are constants, lambda abstractions, unrestricted results, or closed type abstractions. In our semantics, results are first-class, i.e., variables are replaced by results at run time (hence, our semantics is call-by-*result* more precisely). Evaluation contexts are standard in a call-by-value, left-to-right semantics [Felleisen and Hieb 1992].

Extrusion contexts, ranged over by $\mathbb{E}$, are introduced to define the semantics of restrictions: their evaluation starts with evaluating the bodies and then extrudes the ν-binders upwards if they are at a redex position. Extrusion contexts identify contexts under which ν-binders are extruded. For example, an application $(\nu\alpha.\,\lambda x.x)\,1$ can be reduced to $\nu\alpha.\,((\lambda x.x)\,1)$ by extruding the ν-binder

$$\begin{array}{rrl} \text{Results } R & ::= & V \mid \nu\alpha.\, R \qquad \text{Values } V ::= c \mid \lambda x.M \mid !R \mid \Lambda\alpha.M \\ \text{Evaluation contexts } E & ::= & [\,]\ M_2 \mid R_1\ [\,] \mid \mathsf{let}\,!x = [\,]\ \mathsf{in}\, M_2 \mid \\ & & \Lambda^\circ\langle\, \alpha,\, [\,] \,\rangle \mid [\,]\ A \mid \nu\alpha.\ [\,] \mid\, ![\,] \\ \text{Extrusion contexts } \mathbb{E} & ::= & [\,]\ R_2 \mid \Lambda^\circ\langle\, \alpha,\, [\,] \,\rangle \mid [\,]\ A \end{array}$$

Reduction rules $\boxed{M_1 \rightsquigarrow M_2}$

$$\begin{array}{rcll} c_1\,(\nu\overline{\alpha}.\, c_2) & \rightsquigarrow & \nu\overline{\alpha}.\, \zeta(c_1, c_2) & \text{R\_Const} \\ \mathsf{let}\,!x = \nu\overline{\alpha}.\, !R\,\mathsf{in}\, M & \rightsquigarrow & M[\nu\overline{\alpha}.\, R/x] & \text{R\_Bang} \\ (\Lambda\alpha.M)\,A & \rightsquigarrow & M[A/\alpha] & \text{R\_TBeta} \end{array} \qquad \begin{array}{rcll} (\lambda x.M)\,R & \rightsquigarrow & M[R/x] & \text{R\_Beta} \\ \Lambda^\circ\langle\, \alpha, !R \,\rangle & \rightsquigarrow & !\Lambda\alpha.R & \text{R\_Closing} \end{array}$$

Evaluation rules $\boxed{M_1 \longrightarrow M_2}$

$$\frac{M_1 \rightsquigarrow M_2}{M_1 \longrightarrow M_2}\ \text{E\_Red} \qquad \frac{M_1 \longrightarrow M_2}{E[M_1] \longrightarrow E[M_2]}\ \text{E\_Eval} \qquad \frac{\alpha \notin \mathit{ftv}(\mathbb{E})}{\mathbb{E}[\nu\alpha.\, R] \longrightarrow \nu\alpha.\, \mathbb{E}[R]}\ \text{E\_Extr}$$

Fig. 3. Semantics of Λ^{open}.

under the extrusion context $[\,]\ 1$. This mechanism of extrusion can also be seen in the previous work that handles constructs similar to restrictions [Milner et al. 1992; Montagu and Rémy 2009].

Extrusion contexts are more restrictive than evaluation contexts for the following reasons. First, it excludes $R_1\ [\,]$ and $\mathsf{let}\,!x = [\,]\ \mathsf{in}\, M_2$, which are contexts with the hole at argument positions, because our semantics is call-by-"result": results at argument positions will be substituted for variables. Second, we disallow the extrusion of ν-binders beneath the restriction constructor for preventing meaningless non-terminating computation such as $\nu\alpha.\, \nu\beta.\, R \longrightarrow \nu\beta.\, \nu\alpha.\, R \longrightarrow \nu\alpha.\, \nu\beta.\, R \longrightarrow \cdots$. Third, we disallow the extrusion of ν-binders beneath the !-constructor because allowing it may result in the violation of the affine discipline. For example, a term $!\nu\alpha.\, \lambda x.\Lambda^\circ\langle\, \alpha, M \,\rangle$ would be reduced to $\nu\alpha.\, !\lambda x.\Lambda^\circ\langle\, \alpha, M \,\rangle$ if the ν-binder under the !-constructor could be extruded. The resulting term indicates that the value $\lambda x.\Lambda^\circ\langle\, \alpha, M \,\rangle$ can be duplicated. However, this duplication causes the type variable α to be generalized more than once.

4.3.2 Reduction and Evaluation. The semantics consists of two relations over terms that contain no free variables (but may contain free type variables): the reduction relation $\rightsquigarrow$ for basic computation and the evaluation relation $\longrightarrow$ for reducing subterms and extruding ν-binders. They are defined as the smallest relations satisfying the rules in Figure 3. We write $\overline{\alpha}$ for a sequence of type variables $\alpha_1, \cdots, \alpha_n$ and $\nu\overline{\alpha}.\, M$ for $\nu\alpha_1.\, \cdots \nu\alpha_n.\, M$ when $\overline{\alpha} = \alpha_1, \cdots, \alpha_n$.

The reduction rules for constant, function, and type applications are standard in the λ-calculus or System F (except for the existence of ν-binders). The rule (R_Bang) for let-expressions comes from the linear λ-calculus, dropping the !-constructor from substituted results. The substituted results must retain ν-binders for subject reduction. The rule (R_Closing) converts an open type abstraction $\Lambda^\circ\langle\, \alpha, !R \,\rangle$ to an unrestricted closed type abstraction $!\Lambda\alpha.R$, as (T_Gen) indicates. Because R is unrestricted, it never generalizes type variables bound outside of itself; thus, R satisfies the requirement that (T_TAbs) imposes on the bodies of well-typed closed type abstractions.

Evaluation subsumes reduction (E_Red). Subterms are evaluated if they are placed under an evaluation context (E_Eval). The rule (E_Extr) extrudes a ν-binder under an extrusion context $\mathbb{E}$. The side condition, which uses the notation $\mathit{ftv}(\mathbb{E})$ denoting the set of free type variables in $\mathbb{E}$, prevents the extruded ν-binder from capturing free type variables in $\mathbb{E}$. This is necessary for proving subject reduction. To see it, consider a well-typed term $M \stackrel{\text{def}}{=} \nu\alpha.\, ((\nu\alpha.\, R_1)\, R_2)$ where both

results R_1 and R_2 require the type variable α to be assigned the use $\mathbf{1}$; thus, for the term M to be well typed, the type variable α in R_1 and α in R_2 must be bounded by different ν-binders. Let $\mathbb{E} = [\,]\ R_2$. Then, eliminating the side condition from (E_Extr) would allow the following reduction:

$$\nu\alpha.\,((\nu\alpha.\,R_1)\,R_2) = \nu\alpha.\,\mathbb{E}[\nu\alpha.\,R_1] \longrightarrow \nu\alpha.\,\nu\alpha.\,\mathbb{E}[R_1] = \nu\alpha.\,\nu\alpha.\,(R_1\,R_2)\,.$$

After the reduction, the type variable α in R_1 and α in R_2 are both bounded by the inner ν-binder. Hence, the reduction result would be ill typed, which implies that subject reduction would not hold. The side condition on (E_Extr) rejects this undesired reduction because the type variable α bound by the extruded, inner ν-binder occurs in R_2, that is, in $\mathbb{E}$. We can evaluate M by alpha-renaming the type variable α bound by the inner or outer ν-binder to a fresh β before the extrusion:

$$\nu\alpha.\,((\nu\alpha.\,R_1)\,R_2) = \nu\alpha.\,((\nu\beta.\,R_1[\beta/\alpha])\,R_2) \longrightarrow \nu\alpha.\,\nu\beta.\,(R_1[\beta/\alpha]\,R_2) \quad (\text{where } \beta \notin \mathit{ftv}(R_2))\,.$$

In the resulting term, β in $R_1[\beta/\alpha]$ and α in R_2 are bounded by the different ν-binders as desired.

4.4 Type Soundness

We show type soundness of Λ^{open} via progress and subject reduction [Wright and Felleisen 1994]. We use the metavariable Δ for typing contexts that consist only of type variable bindings. The relation $\longrightarrow^*$ is the reflexive, transitive closure of $\longrightarrow$.

Lemma 1 (Progress). *If $\Delta \vdash M : A$, then $M = R$ for some R; or $M \longrightarrow M'$ for some M'.*

The proof of subject reduction depends on the following type substitution lemma. We write $\Gamma[A/\alpha]$ for a typing context obtained by applying $[A/\alpha]$ to the bindings in Γ.

Lemma 2 (Type substitution). *Suppose that $\pi = \mathbf{0}$ for any $\alpha^\pi \in \Gamma_1$. If $\Gamma_1 \vdash A$ and $\Gamma_1, \alpha^{\mathbf{0}}, \Gamma_2 \vdash M : B$, then $\Gamma_1, \Gamma_2[A/\alpha] \vdash M[A/\alpha] : B[A/\alpha]$.*

As explained at the end of Section 4.2, this lemma can be proven by ensuring that a term M involves no open type abstraction that generalizes type variables occurring in a substituted type A. It is indeed ensured by the condition of the lemma because, while type variables generalized by open type abstractions must have the use $\mathbf{1}$, the condition requires that M be typechecked under a typing context $\Gamma_1, \alpha^{\mathbf{0}}, \Gamma_2$ that assigns the use $\mathbf{0}$ to all the type variables occurring in A (which are in $\mathit{dom}(\Gamma_1)$). The type substitution lemma is used for proving that the reduction rule (R_TBeta) for type application $(\Lambda\alpha.M)\ A$ is type preserving. The typing rule (T_TAbs) ensures that the body M is typechecked under certain typing context $\omega\Delta$ that assigns only the use $\mathbf{0}$. Thus, we can apply the type substitution lemma to prove that $M[A/\alpha]$ has the same type as $(\Lambda\alpha.M)\ A$.

Lemma 3 (Subject Reduction). *If $\Delta \vdash M_1 : A$ and $M_1 \longrightarrow M_2$, then $\Delta \vdash M_2 : A$.*

Theorem 1 (Type Soundness). *If $\Delta \vdash M : A$ and $M \longrightarrow^* M'$ and there exists no M'' such that $M' \longrightarrow M''$, then $M' = R$ for some R such that $\Delta \vdash R : A$.*

5 CPS TRANSFORMATION FOR CURRY-STYLE CBV SYSTEM F

This section defines a CPS transformation for $\lambda_v^\forall$ and proves that it is meaning and type preserving.

5.1 Definition

Figure 4 defines CPS-transformations for types and terms in $\lambda_v^\forall$. $[\![\tau]\!]$ and $[\![\tau]\!]_{\mathrm{v}}$ transform type τ of terms and values in $\lambda_v^\forall$ to a type in Λ^{open}, respectively. $[\![\Theta \vdash e : \tau]\!] \Rightarrow R$ transforms a typing derivation for $\Theta \vdash e : \tau$ to a result R, defined as the smallest relation satisfying the rules at the bottom of Figure 4. Notice that it is a total, functional relation between derivations and results.

The transformation $[\![\tau]\!]$ shows that a term of type τ is transformed to a CPS term of the type $\forall\alpha.(![\![\tau]\!]_{\mathrm{v}} \multimap \alpha) \multimap \alpha$ for $\alpha \notin \mathit{ftv}(\tau)$. The resulting type abstracts over *answer types*, the types of the

Type transformation $\boxed{[\![\tau]\!]}$ (for terms) $\boxed{[\![\tau]\!]_{\mathrm{v}}}$ (for values)

$$[\![\tau]\!] \stackrel{\text{def}}{=} \forall\alpha.(!{[\![\tau]\!]_{\mathrm{v}}} \multimap \alpha) \multimap \alpha \quad (\alpha \notin ftv(\tau))$$

$$[\![\alpha]\!]_{\mathrm{v}} \stackrel{\text{def}}{=} \alpha \qquad [\![\iota]\!]_{\mathrm{v}} \stackrel{\text{def}}{=} \iota \qquad [\![\tau_1 \to \tau_2]\!]_{\mathrm{v}} \stackrel{\text{def}}{=} !{[\![\tau_1]\!]_{\mathrm{v}}} \multimap [\![\tau_2]\!] \qquad [\![\forall\alpha.\tau]\!]_{\mathrm{v}} \stackrel{\text{def}}{=} \forall\alpha.[\![\tau]\!]_{\mathrm{v}}$$

Typing derivation transformation $\boxed{[\![\Theta \vdash e : \tau]\!] \Rightarrow R}$

$$\frac{\vdash \Theta \quad x{:}\tau \in \Theta}{[\![\Theta \vdash x : \tau]\!] \Rightarrow \Lambda\alpha.\lambda k.k\,!x}\ \text{C\_Var} \qquad \frac{\vdash \Theta}{[\![\Theta \vdash c : ty^{\to}(c)]\!] \Rightarrow \Lambda\alpha.\lambda k.k\,[\![c : ty^{\to}(c)]\!]}\ \text{C\_Const}$$

$$\frac{[\![\Theta, x{:}\tau_1 \vdash e : \tau_2]\!] \Rightarrow R \quad y \text{ is fresh}}{[\![\Theta \vdash \lambda x.e : \tau_1 \to \tau_2]\!] \Rightarrow \Lambda\alpha.\lambda k.k\,!(\lambda y.\mathsf{let}\,!x = y\,\mathsf{in}\,R)}\ \text{C\_Abs}$$

$$\frac{[\![\Theta \vdash e_1 : \tau_1 \to \tau_2]\!] \Rightarrow R_1 \quad [\![\Theta \vdash e_2 : \tau_1]\!] \Rightarrow R_2 \quad x \text{ is fresh}}{[\![\Theta \vdash e_1\,e_2 : \tau_2]\!] \Rightarrow \Lambda\alpha.\lambda k.R_1\,\alpha\,(\lambda x.R_2\,\alpha\,(\lambda y.\mathsf{let}\,!z = x\,\mathsf{in}\,z\,y\,\alpha\,k))}\ \text{C\_App}$$

$$\frac{[\![\Theta, \beta \vdash e : \tau]\!] \Rightarrow R}{[\![\Theta \vdash e : \forall\beta.\tau]\!] \Rightarrow \Lambda\alpha.\lambda k.v\beta.\,R\,\alpha\,(\lambda x.k\,\Lambda^{\circ}\langle\beta, x\rangle)}\ \text{C\_TAbs}$$

$$\frac{[\![\Theta \vdash e : \forall\beta.\tau_2]\!] \Rightarrow R \quad \Theta \vdash \tau_1}{[\![\Theta \vdash e : \tau_2[\tau_1/\beta]]\!] \Rightarrow \Lambda\alpha.\lambda k.R\,\alpha\,(\lambda x.\mathsf{let}\,!y = x\,\mathsf{in}\,k\,!(y\,[\![\tau_1]\!]_{\mathrm{v}}))}\ \text{C\_TApp}$$

Fig. 4. CPS transformation.

final outputs (answers) of programs, by the type variable α. Meyer and Wand [1985] found that CPS transformation can be *parameterized* over answer types, that is, it does not require specific answer types but it shares an answer type during the transformation of a program. By contrast, our CPS transformation makes answer types *polymorphic* [Thielecke 2003]. We could define type-preserving CPS transformation that is parameterized over answer types as in Meyer and Wand. However, answer type polymorphism makes it easier to prove that our CPS transformation preserves the meanings of source terms of function types as well as those of base types. We will discuss this at the end of the next section. The CPS type $[\![\tau]\!]$ also indicates that CPS terms invoke continuations of the type $!{[\![\tau]\!]_{\mathrm{v}}} \multimap \alpha$ *at most once*. This reflects the fact that pure terms use continuations only linearly [Berdine et al. 2001, 2002; Hasegawa 2002]. This linearity is crucial in transforming polymorphic terms, as seen later. The argument type $!{[\![\tau]\!]_{\mathrm{v}}}$ of continuations indicates that they may duplicate the evaluation results of the terms. This reflects that any value in $\lambda_v^{\forall}$ is unrestricted.

The transformation $[\![\tau]\!]_{\mathrm{v}}$ for a type τ of values follows Harper and Lillibridge [1993b]. The transformation of polymorphic types reflects the fact that, in implicit polymorphism, type application of a polymorphic value triggers no computation and simply returns a value. The transformation of function types utilizes the standard encoding of unrestricted function types through the combination of affine function and of-course types [Girard 1987].

The transformation rules for $[\![\Theta \vdash e : \tau]\!] \Rightarrow R$ follow certain conventions. First, they suppose that variable k, which denotes continuations, never appears in source terms. Second, they produce CPS terms abstracting over answer types. Third, because continuations require arguments of an of-course type, CPS terms wrap expressions passed to continuations using the !-constructor, and remove the outermost !-constructor wrapping arguments of continuations by let when using them.

Now, let's take a closer look at the transformation rules. The rule (C_Var) transforms a variable x to $\lambda k.k\,!x$, where x is wrapped by the !-constructor. This is valid because all variables appearing in source terms are given the use ω in the CPS image. The rule (C_Const) produces a CPS term

that provides continuations with the CPS counterpart $[\![c : ty^{\to}(c)]\!]$ of the eta-expansion result of a constant c. In general, $[\![c : ty^{\to}(c)]\!]$ has the type $![\![ty^{\to}(c)]\!]_{\mathrm{v}}$ as expected by the continuations of c. We omit the full definition of $[\![c : ty^{\to}(c)]\!]$ because it would not be novel and is tedious. Interested readers are referred to the supplementary material. The rule (C_Abs) transforms lambda abstractions. The CPS function $\lambda y.\mathsf{let}\,!x = y\,\mathsf{in}\,R$ first binds the unrestricted variable x to an argument and then proceeds to R corresponding to the body of a lambda abstraction. The rule (C_App) defines CPS transformation of applications in CBV semantics [Plotkin 1975; Reynolds 1972]. As CPS terms are polymorphic for answer types, the CPS counterparts R_1 and R_2 of the subterms, as well as the CPS term returned by the application $z\,y$, are applied to the answer type variable α. The rule (C_TApp) handles type instantiation, producing a CPS term that instantiates the evaluation result of a polymorphic term. The type argument τ_1 is transformed to $[\![\tau_1]\!]_{\mathrm{v}}$ because type variables in the CPS image are supposed to range over CPS value types.

The rule (C_TAbs) for polymorphic terms is the crux of our CPS transformation. It is applied to a derivation for a typing judgment $\Theta \vdash e : \forall\beta.\tau$ with the premise $\Theta, \beta \vdash e : \tau$, and it produces a well-typed CPS term of $[\![\forall\beta.\tau]\!] = \forall\alpha.(!(\forall\beta.[\![\tau]\!]_{\mathrm{v}}) \multimap \alpha) \multimap \alpha$. Here, we show informally that the produced CPS term is indeed of the type; refer to the next section for the formal statement and to the supplementary material for the complete proof. For simplicity, suppose that Θ is empty (we will consider nonempty typing contexts in the next section). Let R be a result such that $[\![\beta \vdash e : \tau]\!] \Rightarrow R$. Then, we must show that

$$\emptyset \vdash \Lambda\alpha.\lambda k.\nu\beta.\, R\,\alpha\,(\lambda x.k\,\Lambda^{\circ}\langle\beta, x\rangle) : \forall\alpha.(!(\forall\beta.[\![\tau]\!]_{\mathrm{v}}) \multimap \alpha) \multimap \alpha\,.$$

Let Γ be a typing context $\alpha^{\mathbf{0}}, k :^{\mathbf{1}} !(\forall\beta.[\![\tau]\!]_{\mathrm{v}}) \multimap \alpha$. Because term e only references type variable β, we can suppose R to be of the type $[\![\tau]\!] = \forall\alpha.(![\![\tau]\!]_{\mathrm{v}} \multimap \alpha) \multimap \alpha$ under the typing context $\omega\Gamma, \beta^{\mathbf{0}}$. Then, noting $\omega\Gamma + \Gamma = \Gamma$, we can construct the following derivation for the judgment in question:

$$\dfrac{\dfrac{\dfrac{\vdots}{\omega\Gamma, \beta^{\mathbf{0}} \vdash R\,\alpha : (![\![\tau]\!]_{\mathrm{v}} \multimap \alpha) \multimap \alpha}\,\mathrm{T\_TApp} \quad \dfrac{\mathcal{D}}{\Gamma, \beta^{\mathbf{1}} \vdash \lambda x.k\,\Lambda^{\circ}\langle\beta, x\rangle : ![\![\tau]\!]_{\mathrm{v}} \multimap \alpha}\,\mathrm{T\_Abs}}{\Gamma, \beta^{\mathbf{1}} \vdash R\,\alpha\,(\lambda x.k\,\Lambda^{\circ}\langle\beta, x\rangle) : \alpha}\,\mathrm{T\_App}}{\emptyset \vdash \Lambda\alpha.\lambda k.\nu\beta.\, R\,\alpha\,(\lambda x.k\,\Lambda^{\circ}\langle\beta, x\rangle) : \forall\alpha.(!(\forall\beta.[\![\tau]\!]_{\mathrm{v}}) \multimap \alpha) \multimap \alpha}\,\mathrm{T\_TAbs, T\_Abs, T\_Nu}$$

where the remaining derivation $\mathcal{D}$ is

$$\dfrac{\dfrac{}{\Gamma, \beta^{\mathbf{0}}, x :^{\mathbf{0}} ![\![\tau]\!]_{\mathrm{v}} \vdash k : !(\forall\beta.[\![\tau]\!]_{\mathrm{v}}) \multimap \alpha}\,\mathrm{T\_Var} \quad \dfrac{\dfrac{}{\omega\Gamma, \beta^{\mathbf{0}}, x :^{\mathbf{1}} ![\![\tau]\!]_{\mathrm{v}} \vdash x : ![\![\tau]\!]_{\mathrm{v}}}\,\mathrm{T\_Var}}{\omega\Gamma, \beta^{\mathbf{1}}, x :^{\mathbf{1}} ![\![\tau]\!]_{\mathrm{v}} \vdash \Lambda^{\circ}\langle\beta, x\rangle : !\forall\beta.[\![\tau]\!]_{\mathrm{v}}}\,\mathrm{T\_Gen}}{\Gamma, \beta^{\mathbf{1}}, x :^{\mathbf{1}} ![\![\tau]\!]_{\mathrm{v}} \vdash k\,\Lambda^{\circ}\langle\beta, x\rangle : \alpha}\,\mathrm{T\_App}$$

Further, we can observe that (C_TAbs) accurately reflects the semantics of implicit polymorphism because the entire CPS term and R obtained from the premise are computationally equivalent modulo type annotations. In the next section, we formalize this idea to show meaning preservation of the CPS transformation.

5.2 Preservation Properties

This section investigates typing and semantic properties of the CPS transformation. In particular, we show that it preserves the type and meaning of a source term. The full proofs of the statements presented in this section are found in the supplementary material.

First, we show that the CPS transformation is type preserving. We define transformation $[\![\Theta]\!]$ of typing contexts Θ as: $[\![\emptyset]\!] \stackrel{\mathrm{def}}{=} \emptyset$; $[\![\Theta, x : \tau]\!] \stackrel{\mathrm{def}}{=} [\![\Theta]\!], x :^{\omega} [\![\tau]\!]_{\mathrm{v}}$; and $[\![\Theta, \alpha]\!] \stackrel{\mathrm{def}}{=} [\![\Theta]\!], \alpha^{\mathbf{0}}$.

Lemma 4. *If* $[\![\Theta \vdash e : \tau]\!] \Rightarrow R$, *then* $[\![\Theta]\!] \vdash R : [\![\tau]\!]$.

Proof. By induction on the derivation of $[\![\Theta \vdash e : \tau]\!] \Rightarrow R$. □

Theorem 2 (Type Preservation of CPS Transformation). *If $\Theta \vdash e : \tau$, then there exists some R such that $[\![\Theta \vdash e : \tau]\!] \Rightarrow R$ and $[\![\Theta]\!] \vdash R : [\![\tau]\!]$.*

Proof. By Lemma 4, it suffices to show that, if $\Theta \vdash e : \tau$, then $[\![\Theta \vdash e : \tau]\!] \Rightarrow R$ for some R. This is easy to show because there is a one-to-one correspondence between the typing rules in $\lambda_v^{\forall}$ and the CPS transformation rules. □

Next, we show meaning preservation, a formal property stating that the CPS-transformed program behaves equivalently to a source program. To this end, we begin by defining type erasure, which translates terms in Λ^{open} to the untyped λ-calculus by erasing type annotations. Type erasure makes it possible to relate terms produced by our CPS transformation and those by (a variant of) the CBV CPS transformation given by Plotkin [1975]. Then, we show meaning preservation of our CPS transformation via that of Plotkin's CPS transformation.

Definition 3 (Type Erasure). *Type erasure* $\mathsf{erase}(M)$ *is a function that translates terms M in Λ^{open} to untyped terms in $\lambda_v^{\forall}$, defined by induction on M as follows.*

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{erase}(x) & \stackrel{\text{def}}{=} & x & \mathsf{erase}(c) & \stackrel{\text{def}}{=} & c \\
\mathsf{erase}(\lambda x.M) & \stackrel{\text{def}}{=} & \lambda x.\mathsf{erase}(M) & \mathsf{erase}(M_1\ M_2) & \stackrel{\text{def}}{=} & \mathsf{erase}(M_1)\ \mathsf{erase}(M_2) \\
\mathsf{erase}(!M) & \stackrel{\text{def}}{=} & \mathsf{erase}(M) & \mathsf{erase}(\mathsf{let}\ !x = M_1\ \mathsf{in}\ M_2) & \stackrel{\text{def}}{=} & (\lambda x.\mathsf{erase}(M_2))\ \mathsf{erase}(M_1) \\
\mathsf{erase}(v\alpha.\, M) & \stackrel{\text{def}}{=} & \mathsf{erase}(M) & \mathsf{erase}(\Lambda^{\circ}\langle\, \alpha, M\,\rangle) & \stackrel{\text{def}}{=} & \mathsf{erase}(M) \\
\mathsf{erase}(\Lambda\alpha.M) & \stackrel{\text{def}}{=} & \mathsf{erase}(M) & \mathsf{erase}(M\,A) & \stackrel{\text{def}}{=} & \mathsf{erase}(M)
\end{array}
$$

Type erasure is meaning preserving. We write $\longrightarrow_F^*$ for the reflexive, transitive closure of $\longrightarrow_F$. A term M is erasable if and only if, for any closed type abstraction $\Lambda\alpha.M'$ in M, the body M' is a result.

Theorem 3 (Meaning Preservation of Type Erasure). *Suppose that M is erasable.*

(1) If $M \longrightarrow^* M'$, then $\mathsf{erase}(M) \longrightarrow_F^* \mathsf{erase}(M')$. Furthermore, if M' is a result, then $\mathsf{erase}(M')$ is a value.

(2) If $\Delta \vdash M : A$ and $\mathsf{erase}(M) \longrightarrow_F^* e$, then $M \longrightarrow^* M'$ for some M' such that $\mathsf{erase}(M') = e$. Furthermore, if $e = w$, then $M' \longrightarrow^* R$ for some R such that $\mathsf{erase}(R) = w$.

The second case in Theorem 3 assumes that M is well typed because there is an ill-typed term M such that M gets stuck but $\mathsf{erase}(M)$ does not (for example, consider $M = (\lambda x.x)\ \mathsf{int}$).

Now, we relate our CPS transformation to a variant $(\!|\cdot|\!)$ of Plotkin's CBV CPS transformation for the untyped λ-calculus. The only difference between $(\!|\cdot|\!)$ and Plotkin's transformation is the treatment of functional constants. The former transforms a constant c of the type $ty^{\rightarrow}(c)$ at translation time—we express such transformation by $(\!|\, c : ty^{\rightarrow}(c)\,|\!)$, which returns CPS functions if c is a functional constant, as $[\![c : ty^{\rightarrow}(c)]\!]$. By contrast, the latter supposes that the transformation of functional constants is performed at run time (specifically, when the constants are applied). This is because we assume that the types of constants are known statically, while Plotkin worked in an untyped setting, where the types of constants are unknown. Nonetheless, the CPS terms obtained by the two transformations are semantically equivalent, and thus $(\!|\cdot|\!)$ is considered to enjoy the semantic properties of Plotkin's CBV CPS transformation. Except for the handling of functional constants, $(\!|\cdot|\!)$ is given a standard definition as in Plotkin [1975]. Hence, we omit its definition in the paper; see the supplementary material for details. We define the full CBV $\beta\eta$-reduction $\Longmapsto_{\beta\eta_v}$ as follows: $e_1 \Longmapsto_{\beta\eta_v} e_2$ if and only if e_2 is obtained by reducing a subterm (possibly under a λ) in e_1

with CBV β-reduction $(\lambda x.e)\, w \rightsquigarrow e[w/x]$ or CBV η-reduction $(\lambda x.w\, x) \rightsquigarrow w$ where x does not occur free in w. We write $\Longmapsto^*_{\beta\eta_v}$ for the reflexive, transitive closure of $\Longmapsto_{\beta\eta_v}$.

Lemma 5. *If* $[\![\Theta \vdash e : \tau]\!] \Rightarrow R$, *then* $\mathsf{erase}(R) \Longmapsto^*_{\beta\eta_v} (\!|\, e \,|\!)$.

Thus, the untyped CPS programs $\mathsf{erase}(R)\ (\lambda x.x)$ and $(\!|\, e \,|\!)\ (\lambda x.x)$ are related by $\Longmapsto^*_{\beta\eta_v}$. The following lemma proves that their evaluation results are equivalent modulo $\Longmapsto^*_{\beta\eta_v}$. A term e gets stuck if and only if there exists a term e' such that (1) $e \longrightarrow^*_F e'$, (2) e' does not take a further evaluation step, and (3) e' is not a value.

Lemma 6.

(1) If $e_1 \Longmapsto^*_{\beta\eta_v} e_2$ *and* $e_2 \longrightarrow^*_F w_2$ *and* e_1 *does not get stuck, then there exists some* w_1 *such that* $e_1 \longrightarrow^*_F w_1$ *and* $w_1 \Longmapsto^*_{\beta\eta_v} w_2$.
(2) If $e_1 \Longmapsto^*_{\beta\eta_v} e_2$ *and* $e_1 \longrightarrow^*_F w_1$, *then there exists some* w_2 *such that* $e_2 \longrightarrow^*_F w_2$ *and* $w_1 \Longmapsto^*_{\beta\eta_v} w_2$.

Lemma 6 (1) would not hold without the condition that e_1 does not get stuck. For example, consider $e_1 = \mathsf{succ}\,(\lambda x.1\, x)$ and $e_2 = \mathsf{succ}\, 1$. We can find $e_1 \Longmapsto_{\beta\eta_v} e_2$ and $e_2 \longrightarrow_F 2$, but the non-value term e_1 cannot be evaluated further.

The last auxiliary property is meaning preservation of $(\!|\, \cdot \,|\!)$, taken from Plotkin [1975]. The function Ψ converts values in $\lambda_v^{\forall}$ into CPS, defined as: $\Psi(c) \stackrel{\text{def}}{=} (\!|\, c : ty^{\rightarrow}(c) \,|\!)$ and $\Psi(\lambda x.e) \stackrel{\text{def}}{=} \lambda x.(\!|\, e \,|\!)$.

Corollary 1 (Meaning preservation of $(\!|\, \cdot \,|\!)$).

(1) if $e \longrightarrow^*_F w$, *then* $(\!|\, e \,|\!)\ (\lambda x.x) \longrightarrow^*_F \Psi(w)$.
(2) if $(\!|\, e \,|\!)\ (\lambda x.x) \longrightarrow^*_F w'$, *then* $e \longrightarrow^*_F w$ *for some* w *such that* $w' = \Psi(w)$.

Proof. By the indifference and simulation properties of $(\!|\, \cdot \,|\!)$, and the equivalence of the small-step and big-step CBV semantics for $\lambda_v^{\forall}$. These properties have been proven by Plotkin [1975]. □

Finally, we show meaning preservation of our CPS transformation. We write $w \Rightarrow R$ if and only if $\mathsf{erase}(R) \Longmapsto^*_{\beta\eta_v} \Psi(w)$; this relates the evaluation result of a term in $\lambda_v^{\forall}$ and that of its CPS counterpart in Λ^{open}.

Theorem 4 (Meaning Preservation of CPS Transformation). *Suppose that* $[\![\emptyset \vdash e : \tau]\!] \Rightarrow R$.

(1) If $e \longrightarrow^*_F w$, *then* $R\,![\![\tau]\!]_{\mathsf{v}}\ (\lambda x.x) \longrightarrow^* R'$ *for some* R' *such that* $w \Rightarrow R'$.
(2) If $R\,![\![\tau]\!]_{\mathsf{v}}\ (\lambda x.x) \longrightarrow^* R'$, *then* $e \longrightarrow^*_F w$ *for some* w *such that* $w \Rightarrow R'$.

Proof. First, we note that application of Theorem 3 in this proof requires R to be erasable, which is easily shown from the definition of the CPS transformation.

(1) By Lemma 5, $\mathsf{erase}(R)\ (\lambda x.x) \Longmapsto^*_{\beta\eta_v} (\!|\, e \,|\!)\ (\lambda x.x)$. By Corollary 1 (1) with the assumption, $(\!|\, e \,|\!)\ (\lambda x.x) \longrightarrow^*_F \Psi(w)$. Because $\emptyset \vdash R\,![\![\tau]\!]_{\mathsf{v}}\ (\lambda x.x) : !\,[\![\tau]\!]_{\mathsf{v}}$ by Lemma 4, we can show that $\mathsf{erase}(R\,![\![\tau]\!]_{\mathsf{v}}\ (\lambda x.x)) = \mathsf{erase}(R)\ (\lambda x.x)$ does not get stuck using Theorem 3 (with a few auxiliary lemmas). Hence, by Lemma 6 (1), there exists some w' such that $\mathsf{erase}(R)\ (\lambda x.x) \longrightarrow^*_F w'$ and $w' \Longmapsto^*_{\beta\eta_v} \Psi(w)$. By Theorem 3, $R\,![\![\tau]\!]_{\mathsf{v}}\ (\lambda x.x) \longrightarrow^* R'$ for some R' such that $\mathsf{erase}(R') = w'$. As $\mathsf{erase}(R') = w' \Longmapsto^*_{\beta\eta_v} \Psi(w)$, we have $w \Rightarrow R'$.

(2) By Lemma 5, $\mathsf{erase}(R)\ (\lambda x.x) \Longmapsto^*_{\beta\eta_v} (\!|\, e \,|\!)\ (\lambda x.x)$. By Theorem 3 with the assumption, we have $\mathsf{erase}(R)\ (\lambda x.x) \longrightarrow^*_F \mathsf{erase}(R')$. As $\mathsf{erase}(R')$ is a value, there exists some w' such that $(\!|\, e \,|\!)\ (\lambda x.x) \longrightarrow^*_F w'$ and $\mathsf{erase}(R') \Longmapsto^*_{\beta\eta_v} w'$ by Lemma 6 (2). By Corollary 1 (2), $e \longrightarrow^*_F w$ for some w such that $w' = \Psi(w)$. As $\mathsf{erase}(R') \Longmapsto^*_{\beta\eta_v} w' = \Psi(w)$, we have $w \Rightarrow R'$. □

The proof of Theorem 4 (1) utilizes answer type polymorphism to show that $R\,!\llbracket\tau\rrbracket_{\mathrm{v}}\,(\lambda x.x)$ is well typed. If we were to adapt answer type *parameterization*, this property—the application of R to $\lambda x.x$ is well typed—would be more difficult to show. To see it, suppose R to be produced by CPS transformation parameterized over answer types. Then, R would be of the type $(!\llbracket\tau\rrbracket_{\mathrm{v}} \multimap \alpha) \multimap \alpha$, in which α is an *answer type parameter*. One may expect that the application $R\,(\lambda x.x)$ can be well typed by instantiating α with $!\llbracket\tau\rrbracket_{\mathrm{v}}$. This instantiation would make R typed at $((!\llbracket\tau\rrbracket_{\mathrm{v}} \multimap \alpha) \multimap \alpha)[!\llbracket\tau\rrbracket_{\mathrm{v}}/\alpha] = (!\llbracket\tau\rrbracket_{\mathrm{v}}[!\llbracket\tau\rrbracket_{\mathrm{v}}/\alpha] \multimap !\llbracket\tau\rrbracket_{\mathrm{v}}) \multimap !\llbracket\tau\rrbracket_{\mathrm{v}}$. However, $!\llbracket\tau\rrbracket_{\mathrm{v}}[!\llbracket\tau\rrbracket_{\mathrm{v}}/\alpha]$ and $!\llbracket\tau\rrbracket_{\mathrm{v}}$ would be different if τ is a function type, because then $!\llbracket\tau\rrbracket_{\mathrm{v}}$ would involve the parameter α. Thus, the instantiation of α to $!\llbracket\tau\rrbracket_{\mathrm{v}}$ could not make $R\,(\lambda x.x)$ well typed in general. Unfortunately, we could not find type instantiation that generally makes $R\,(\lambda x.x)$ well typed. Note that $R\,(\lambda x.x)$ could be well typed if the type τ of the program involves no function type. However, allowing τ to involve function types provides a more accurate, general relationship between direct style and CPS semantics, and it is indeed established by using answer type polymorphism.

6 LOGICAL RELATION AND PARAMETRICITY

This section defines a parametric, step-indexed Kripke logical relation for Λ^{open} and proves that it satisfies the Fundamental Property and soundness with respect to contextual equivalence.

6.1 Main Idea

We construct a Kripke logical relation, which is a logical relation indexed by *possible worlds* [Pitts and Stark 1993]. Kripke logical relations enable reasoning principles that exploit constraints on run-time environments, such as heap invariants, and worlds keep track of the constraints on the states of related terms. In the present work, worlds track type variables that may occur free in related terms as well as type substitutions for them.

Because running terms in Λ^{open} may contain free type variables bound by ν, our logical relation is defined for such terms. To track type variables that may be referenced by related terms, worlds contain typing contexts Δ that include only type variables. Typing contexts in worlds are used not only to determine type variables that related terms may reference, but also to build contexts under which related terms are tested. For example, consider checking that terms M_1 and M_2 are logically related at a function type $A \multimap B$. A common approach to the checking is to test whether applications $M_1\,M_1'$ and $M_2\,M_2'$ are logically related for any arguments M_1' and M_2' that are logically related at the argument type A. In our logical relation, the construction of the test arguments M_1' and M_2' is constrained not only by the argument type A but also by a typing context under which M_1 and M_2 should be related. For example, if M_1 and M_2 should be related under typing context $\alpha^{\mathbf{1}}$, then it suffices to consider only arguments M_1' and M_2' that are well typed under $\alpha^{\mathbf{0}}$, because otherwise $M_1\,M_1'$ and $M_2\,M_2'$ may result in being ill typed. Considering the uses of type variables enables the exclusion of invalid contexts.

As usual in the work on parametricity [Reynolds 1983], worlds also include interpretations ρ that map type variables α (that were bound by Λ) to triples of the form (A_1, A_2, r). The types A_1 and A_2 are those substituted for α during the evaluation of related terms, respectively. The relation r is a *relational interpretation* of α, determining which results are related at the type α. Logical relations for languages with parametricity relate values V_1 and V_2 of a polymorphic type $\forall\alpha.A$ if and only if, for any types B_1 and B_2 and any relational interpretation r, type applications $V_1\,B_1$ and $V_2\,B_2$ are related at the type A under the interpretation that maps α to (B_1, B_2, r). The flexibility on the choice of B_1, B_2, and r ensures that polymorphic values behave independently of type arguments and enables powerful reasoning principles such as free theorems [Wadler 1989].

A key ingredient of Kripke logical relations is a world extension relation $\sqsupseteq$, which tracks the possible transition of program states represented by worlds. When a world W_1 may evolve into

W_2 (which is written $W_2 \sqsupseteq W_1$), terms related in the world W_1 may be involved in the state of W_2. Thus, a Kripke logical relation is required to be *monotonic* under $\sqsupseteq$. A carefully designed world extension relation enables us to prove that evaluation preserves desired invariants on related terms.

Following the semantics in Λ^{open}, our world extension relation allows worlds to be extended in three ways. First, it allows typing contexts to contain more type variables over time, because substitution may place terms under ν- or Λ-binders, that is, where more type variables may be referenced. Second, it allows the uses of type variables in typing contexts to increase over time. For example, consider a term $\nu\alpha.\,((\lambda x.x)\ R)$. The result R may be typechecked under a typing context $\alpha^{\mathbf{0}}$, but the evaluation produces the term $\nu\alpha.\,R$ in which R is typechecked under $\alpha^{\mathbf{1}}$. Allowing the increase of the uses acknowledges this computation. Third, the world extension relation allows an interpretation ρ to additionally map type variables of the use $\mathbf{0}$ in the current world. This comes from the run-time nature of Λ^{open} that type variables with the use $\mathbf{0}$ may result in being bound by closed type abstractions as computation proceeds, and thus may be replaced by some types. For example, consider a term $\nu\alpha.\,\Lambda^{\circ}\langle\alpha, !R\rangle$ in which the type variable α is bound by ν and is assigned the use $\mathbf{0}$ in typechecking result R. This term is evaluated to $\nu\alpha.\,!\Lambda\alpha.R$, in which the occurrences of α in R are bound by Λ. Thus, R in the original term must be parametric over type substitutions and relational interpretations to α. This is ensured by allowing interpretations ρ to grow.

However, this world extension—specifically, the growth of interpretations ρ—causes a circularity problem in the construction of a *compositional* logical relation. Compositionality is a key property of logical relations that establish parametricity, stating that the logical relation $\mathcal{R}[\![A[B/\alpha]]\!]$ at type $A[B/\alpha]$ is equivalent to the logical relation $\mathcal{R}[\![A]\!]$ in a world where $\mathcal{R}[\![B]\!]$ is the relational interpretation of the type variable α. The problem is that, while a world needs to involve the logical relation $\mathcal{R}[\![B]\!]$ as a relational interpretation, what results are in $\mathcal{R}[\![B]\!]$ also depends on a world. This indicates the need of solving the following recursive equations with a problem of circularity:

$$\begin{array}{lcl} \text{World} & = & \mathit{TypCtx} \times (\mathit{TyVar} \to \mathit{Type} \times \mathit{Type} \times \text{Rel}) \\ \text{Rel} & = & \text{World} \to \mathcal{P}(\mathit{Res} \times \mathit{Res}) \end{array}$$

where *TypCtx*, *TyVar*, *Type*, and *Res* are the sets of all typing contexts, type variables, types, and results, respectively. Worlds must involve relational interpretations (which are in *Rel*) because they may grow over time (see the above example). Note that logical relations for polymorphic languages only with closed type abstractions, as System F, can avoid this circularity by restricting running terms to be closed. Under this restriction, interpretations ρ need not grow while computation proceeds, and relational interpretations can be separated from worlds. By contrast, we need to address this problem due to the existence of open type abstractions.

A well-known approach to addressing circularity in Kripke logical relations is to stratify circular definitions with natural numbers called *step indices* [Ahmed 2006; Appel and McAllester 2001]. Intuitively, a step index is the number of the computation steps for which related terms must behave similarly. *Step-indexed* Kripke logical relations involve steps in worlds and decrease them as computation proceeds. The present work constructs a step-indexed Kripke logical relation to avoid the circularity between worlds and relational interpretations.

6.2 Formal Definition

This section provides the formal definition of our logical relation. We first explain auxiliary definitions in Figure 5 and then the logical relation in Figure 6. Hereinafter, we identify typing contexts Δ_1 and Δ_2 up to permutation (i.e., $\Delta, \alpha^{\pi_1}, \beta^{\pi_2}, \Delta'$ is identical with $\Delta, \beta^{\pi_2}, \alpha^{\pi_1}, \Delta'$) for simplifying the technical development. Because Δ contains only type variables, this identification does not change typability of terms. A use π is larger than or equal to π', written $\pi \geq \pi'$, if and only if $\pi = \pi' + \pi_0$ for some π_0. We use the metavariable S to denote sets of type variables and write $S_1 \# S_2$ if and only

$$
\begin{array}{lcl}
\mathrm{Atom}\,[\Delta, A_1, A_2] & \stackrel{\mathrm{def}}{=} & \{(M_1, M_2) \mid \Delta \vdash M_1 : A_1 \ \wedge\ \Delta \vdash M_2 : A_2\} \\
\mathrm{Atom}^{\mathrm{res}}\,[\Delta, A_1, A_2] & \stackrel{\mathrm{def}}{=} & \{(R_1, R_2) \mid (R_1, R_2) \in \mathrm{Atom}\,[\Delta, A_1, A_2]\} \\
\mathrm{Atom}\,[W, A] & \stackrel{\mathrm{def}}{=} & \mathrm{Atom}\,[W.\Delta, W.\rho_{\mathrm{fst}}(A), W.\rho_{\mathrm{snd}}(A)] \\
\mathrm{World}_n & \stackrel{\mathrm{def}}{=} & \{(m,\ \Delta,\ \rho) \in \mathit{Nat} \times \mathit{TypCtx} \times (\mathit{TyVar} \rightharpoonup \mathit{Type} \times \mathit{Type} \times \mathrm{Rel}_m) \mid \\
 & & \qquad m < n \wedge \vdash (m,\ \Delta,\ \rho)\} \\
\mathrm{Rel}_n & \stackrel{\mathrm{def}}{=} & \bigcup_{A_1, A_2} \mathrm{Rel}_n[A_1, A_2] \\
\mathrm{Rel}_n[A_1, A_2] & \stackrel{\mathrm{def}}{=} & \{r \in (W : \mathrm{World}_n) \rightharpoonup \mathcal{P}(\mathrm{Atom}^{\mathrm{res}}\,[W.\Delta, W.\rho_{\mathrm{fst}}(A_1), W.\rho_{\mathrm{snd}}(A_2)]) \mid \\
 & & \quad \forall\, W_1.\, \forall\, W_2 \sqsupseteq W_1.\, \forall\, (R_1, R_2) \in r(W_1).\ (R_1, R_2)_{W_2} \in r(W_2) \\
 & & \quad \wedge\ \forall\, W, \rho.\ \rho \uplus W \in \mathit{dom}(r) \ \wedge\ \mathit{dom}(\rho) \# \mathit{ftv}(A_1) \ \wedge\ \mathit{dom}(\rho) \# \mathit{ftv}(A_2) \\
 & & \qquad\qquad \Longrightarrow r(\rho \uplus W) \subseteq r(W) \\
 & & \quad \wedge\ \forall\, W, \alpha.\ \{\alpha\} \# \mathit{ftv}(A_1) \ \wedge\ \{\alpha\} \# \mathit{ftv}(A_2) \ \wedge \vdash W \\
 & & \qquad\qquad \Longrightarrow \forall\, (R_1, R_2) \in r(W@\alpha).\ (\nu\alpha.\, R_1, \nu\alpha.\, R_2) \in r(W) \\
 & & \quad \wedge\ \forall\, W, \alpha.\ \{\alpha\} \# W \\
 & & \quad \Longrightarrow \forall\, (R_1, R_2) \in r(W).\ (\nu\alpha.\, R_1, R_2) \in r(W) \ \wedge\ (R_1, \nu\alpha.\, R_2) \in r(W)\ \}
\end{array}
$$

$$
\begin{array}{lcl}
S \# W & \stackrel{\mathrm{def}}{=} & S \# \mathit{dom}(W.\Delta) \ \wedge\ S \# \mathit{dom}(W.\rho) \\
\rho \uplus W & \stackrel{\mathrm{def}}{=} & (W.n,\ W.\Delta,\ \rho \uplus W.\rho) \quad (\text{if } \mathit{dom}(\rho) \# W) \\
\omega\, W & \stackrel{\mathrm{def}}{=} & (W.n,\ \omega(W.\Delta),\ W.\rho) \\
W@\alpha & \stackrel{\mathrm{def}}{=} & (W.n,\ (W.\Delta, \alpha^{\mathbf{1}}),\ W.\rho) \quad (\text{if } \{\alpha\} \# W) \\
(n + m,\ \Delta,\ \rho) - m & \stackrel{\mathrm{def}}{=} & (n,\ \Delta,\ \rho) \\
\blacktriangleright W & \stackrel{\mathrm{def}}{=} & W - 1 \\
\vdash W & \stackrel{\mathrm{def}}{=} & \mathit{dom}(W.\rho) \# \mathit{dom}(W.\Delta) \ \wedge\ \forall\, \alpha \in \mathit{dom}(W.\rho).\ W \vdash W.\rho(\alpha) \\
W \vdash (A_1, A_2, r) & \stackrel{\mathrm{def}}{=} & W.\Delta \vdash A_1 \ \wedge\ W.\Delta \vdash A_2 \ \wedge\ r \in \mathrm{Rel}_{W.n}[A_1, A_2] \\
W_1 \sqsupseteq W_2 & \stackrel{\mathrm{def}}{=} & \vdash W_1 \ \wedge \vdash W_2 \ \wedge\ W_1.n \le W_2.n \ \wedge \\
 & & \quad \exists\, \rho.\ (W_1.\Delta, \dagger(\rho)) \gg W_2.\Delta \ \wedge\ W_1.\rho = \rho \circ W_2.\rho \ \wedge\ W_2.\Delta \succ \rho \\
\Delta_1 \gg \Delta_2 & \stackrel{\mathrm{def}}{=} & \exists\, \Delta, \Delta_0.\ \Delta_1 = (\Delta_2 + \Delta), \Delta_0 \\
\dagger(\rho) & \stackrel{\mathrm{def}}{=} & \omega\Delta \text{ such that } \mathit{dom}(\Delta) = \mathit{dom}(\rho) \\
\rho_2 \circ \rho_1 & \stackrel{\mathrm{def}}{=} & \rho_2 \uplus \{\alpha \Longmapsto (\rho_{2\mathrm{fst}}(\rho_{1\mathrm{fst}}(\alpha)), \rho_{2\mathrm{snd}}(\rho_{1\mathrm{snd}}(\alpha)), \rho_1[\alpha]) \mid \alpha \in \mathit{dom}(\rho_1)\} \\
\Gamma \succ \rho & \stackrel{\mathrm{def}}{=} & \forall\, \alpha \in \mathit{dom}(\rho) \cap \mathit{dom}(\Gamma). \\
 & & \quad \forall\, \beta \in (\mathit{ftv}(\rho_{\mathrm{fst}}(\alpha)) \cup \mathit{ftv}(\rho_{\mathrm{snd}}(\alpha))) \cap \mathit{dom}(\Gamma).\ \beta^{\mathbf{0}} \in \Gamma \\
(R_1, R_2)_W & \stackrel{\mathrm{def}}{=} & (W.\rho_{\mathrm{fst}}(R_1),\ W.\rho_{\mathrm{snd}}(R_2)) \\
(W_1, W_2) \ni W_3 & \stackrel{\mathrm{def}}{=} & W_1.n = W_2.n = W_3.n \ \wedge\ W_1.\Delta + W_2.\Delta = W_3.\Delta \ \wedge\ W_1.\rho = W_2.\rho = W_3.\rho
\end{array}
$$

Fig. 5. Auxiliary Definitions for the Logical Relation.

if the sets S_1 and S_2 are disjoint. Further, we write $\mathcal{P}(X)$ for the power set of a set X and $X \rightharpoonup Y$ for the set of partial functions from X to Y.

6.2.1 Auxiliary Definitions. We define various relations as subsets of $\mathrm{Atom}[\Delta, A_1, A_2]$, which consists of pairs of terms that are of the types A_1 and A_2 under the typing context Δ, respectively. We denote the set of pairs of well-typed results by $\mathrm{Atom}^{\mathrm{res}}$.

As discussed in Section 6.1, worlds are triples of the form (n, Δ, ρ): n is a step index; Δ is a typing context under which related terms are typechecked; and ρ is a map from type variables α to triples (A_1, A_2, r) that consist of types A_1 and A_2 substituted for α (on the left- and right-hand sides, respectively) and relational interpretation r of α. For ease of access, we employ the dot notation: $W.n$, $W.\Delta$, and $W.\rho$ denote the step index, typing context, and interpretation of W, respectively. We also write ρ_{fst} and ρ_{snd} for capture-avoiding type substitutions that map a type variable α in $dom(\rho)$ to A_1 and A_2 when $\rho(\alpha) = (A_1, A_2, r)$, respectively. The bottom half of Figure 5 presents operations to manipulate worlds. $\rho \uplus W$ returns the same world as W except that the interpretation is $\rho \uplus W.\rho$ (we write $\rho_1 \uplus \rho_2$ for the concatenation of mappings ρ_1 and ρ_2 with disjoint domains). ωW applies the operation ω and $W@\alpha$ adds $\alpha^{\mathbf{1}}$ to $W.\Delta$. $W - n$ decreases $W.n$ by number n. We write $\blacktriangleright W$ for the one-step later world of W.

World_n is a set of well-formed worlds indexed by natural numbers smaller than n. A world W is well formed, written $\vdash W$, if and only if the domains of $W.\Delta$ and $W.\rho$ are disjoint and, for any (A, A_2, r) in the codomain of $W.\rho$, the types A_1 and A_2 are well formed under $W.\Delta$ and r is a relational interpretation indexed by $W.n$.

$\mathrm{Rel}_n[A_1, A_2]$ is a set of relational interpretations, which, given a current world W, return a set of pairs of results related at a certain type variable in W. The types of the results depend on $W.\rho_{\mathrm{fst}}$ and $W.\rho_{\mathrm{snd}}$ because type substitutions may grow over time. Relational interpretations in $\mathrm{Rel}_n[A_1, A_2]$ require that the current world W be in World_n and worlds in World_n involve relational interpretations in Rel_m for some $m < n$. Thus, the definitions of worlds and relational interpretations are stratified and can avoid circularity.

A relational interpretation r must satisfy four properties. The first is monotonicity under the world extension $\sqsupseteq$. This is a common property required by Kripke logical relations and states that results R_1 and R_2 related in a world W_1 must be related in any future world W_2 of W_1. Notice that free type variables in the related results R_1 and R_2 may be substituted in the future world. Thus, the definition in Figure 5 applies $W_2.\rho_{\mathrm{fst}}$ and $W_2.\rho_{\mathrm{snd}}$ to R_1 and R_2, respectively. This is expressed by the notation $(R_1, R_2)_{W_2}$ that is defined in the bottom of Figure 5. We follow this convention throughout the definitions for the logical relation. The second property is what we call *irrelevance*,[5] which states that which results the relational interpretation r contains is independent of interpretations of type variables irrelevant to the types A_1 and A_2. This is required to prove the logical relation compositional. Note that ordinary logical relations should be irrelevant because they should reference only interpretations of type variables that occur in indexed types; see Neis et al. [2011] for example. The last two properties require r to be closed under the ν constructor. The third is needed to prove the logical relation compatible with the ν constructor. The fourth is for enabling the flexible reasoning with the logical relation. For example, it allows reasoning about equivalence of polymorphic functions such that one returns a given argument wrapped with redundant ν-binders but the other returns the argument itself. These two requirements might correspond to partial bijections in Kripke logical relations for name generation [Neis et al. 2011; New et al. 2020; Pitts and Stark 1993]. Partial bijections represent one-to-one correspondences between visible names. In our setting, visible are type variables with the use $\mathbf{1}$ in a world, that is, those that can be bounded by ν-constructors.

A world W_2 can be extended to a world W_1 ($W_1 \sqsupseteq W_2$) if and only if the number of computation steps left in W_1 is not more than that in W_2 and, for any type variable α^{π} in $W_2.\Delta$, either $W_1.\Delta$

[5] The name comes from Neis et al. [2011].

$$
\begin{array}{lll}
\mathcal{R}[\![\iota]\!]\ W & \stackrel{\text{def}}{=} & \{(v\overline{\alpha_1}.\ c, v\overline{\alpha_2}.\ c)\ \in\ \mathrm{Atom}\,[\,W, \iota\,]\} \\
\mathcal{R}[\![\alpha]\!]\ W & \stackrel{\text{def}}{=} & W.\rho[\alpha](\blacktriangleright W) \\
\mathcal{R}[\![A \multimap B]\!]\ W & \stackrel{\text{def}}{=} & \{(R_1, R_2)\ \in\ \mathrm{Atom}\,[\,W, A \multimap B\,] \mid \forall\, W' \sqsupseteq W.\ \forall\,(W_1,\ W_2) \ni W'. \\
 & & \quad W_1 \sqsupseteq W \Longrightarrow \forall\,(R_1',\ R_2')\ \in\ \mathcal{R}[\![A]\!]\ W_2.\ (R_1\,R_1',\ R_2\,R_2')_{W'}\ \in\ \mathcal{E}[\![B]\!]\ W'\} \\
\mathcal{R}[\![\forall\alpha.A]\!]\ W & \stackrel{\text{def}}{=} & \{(R_1, R_2)\ \in\ \mathrm{Atom}\,[\,W, \forall\alpha.A\,] \mid \forall\, W' \sqsupseteq W.\ \forall\, B_1, B_2, r. \\
 & & \quad \omega\, W' \vdash (B_1, B_2, r)\ \wedge\ \{\alpha\}\#\omega\, W' \Longrightarrow \\
 & & \quad\quad (R_1\,B_1, R_2\,B_2)_{\omega W'}\ \in\ \mathcal{E}[\![A]\!]\ \{\alpha \Rightarrow (B_1, B_2, r)\} \uplus \omega\, W'\} \\
\mathcal{R}[\![!A]\!]\ W & \stackrel{\text{def}}{=} & \{(R_1, R_2)\ \in\ \mathrm{Atom}\,[\,W, !A\,] \mid (\mathsf{let}\,!x = R_1\,\mathsf{in}\,x, \mathsf{let}\,!x = R_2\,\mathsf{in}\,x)\ \in\ \mathcal{E}[\![A]\!]\ \omega W\} \\
\mathcal{E}[\![A]\!]\ W & \stackrel{\text{def}}{=} & \{(M_1, M_2)\ \in\ \mathrm{Atom}\,[\,W, A\,] \mid \forall\, W' \sqsupseteq W.\ \forall\, n\ <\ W'.n. \\
 & & \quad \forall\, R_1.\ W'.\rho_{\mathrm{fst}}(M_1)\ \longrightarrow^n\ R_1 \Longrightarrow \\
 & & \quad\quad \exists\, R_2.\ W'.\rho_{\mathrm{snd}}(M_2)\ \longrightarrow^*\ R_2\ \wedge\ (R_1, R_2)\ \in\ \mathcal{R}[\![A]\!]\ (W' - n)\} \\
\mathcal{G}[\![\Gamma]\!] & \stackrel{\text{def}}{=} & \{(W, \varsigma) \mid \exists\,\Delta.\ \exists\ \textstyle\prod_{x\,\in\,dom_{=1}(\Gamma)}\Delta_x. \\
 & & \quad \vdash W\ \wedge\ \Gamma \succ W.\rho\ \wedge\ W.\Delta\ =\ \Delta + \textstyle\sum_{x\in dom_{=1}(\Gamma)}\Delta_x \\
 & & \quad \wedge\,\forall\,\alpha^{\pi}\ \in\ \Gamma.\ (\exists\,\pi' \geq \pi.\ \alpha^{\pi'}\ \in\ \Delta)\ \vee\ (\pi\ =\ \mathbf{0}\ \wedge\ \alpha\ \in\ dom(W.\rho)) \\
 & & \quad \wedge\,\forall\, x :^{\mathbf{1}} A\ \in\ \Gamma.\ (\varsigma_{\mathrm{fst}}(x), \varsigma_{\mathrm{snd}}(x))\ \in\ \mathcal{R}[\![A]\!]\ (W.n,\ \Delta_x,\ W.\rho) \\
 & & \quad \wedge\,\forall\, x :^{\omega} A\ \in\ \Gamma.\ (\varsigma_{\mathrm{fst}}(x), \varsigma_{\mathrm{snd}}(x))\ \in\ \mathcal{R}[\![A]\!]\ \omega W\}
\end{array}
$$

$$
\begin{array}{lll}
\Gamma \vdash M_1 \preceq M_2 : A & \stackrel{\text{def}}{=} & \Gamma \vdash M_1 : A\ \wedge\ \Gamma \vdash M_2 : A\ \wedge \\
 & & \quad \forall\,(W, \varsigma)\ \in\ \mathcal{G}[\![\Gamma]\!].\ (\varsigma_{\mathrm{fst}}(M_1), \varsigma_{\mathrm{snd}}(M_2))_W\ \in\ \mathcal{E}[\![A]\!]\ W \\
\Gamma \vdash M_1 \approx M_2 : A & \stackrel{\text{def}}{=} & \Gamma \vdash M_1 \preceq M_2 : A\ \wedge\ \Gamma \vdash M_2 \preceq M_1 : A
\end{array}
$$

Fig. 6. Logical relation.

contains $\alpha^{\pi'}$ for some $\pi' \geq \pi$, or $W_1.\rho$ provides α with an interpretation if $\pi = \mathbf{0}$ (i.e., α may be bound by Λ and thus may be replaced). To express this condition, the definition of $\sqsupseteq$ uses three auxiliary definitions: $\Delta_1 \gg \Delta_2$ states that Δ_2 may evolve into Δ_1, that is, Δ_1 may contain more type variables and larger uses than Δ_2; $\dagger(\rho)$ is a typing context that consists of the type variables mapped by ρ and assigns only the use $\mathbf{0}$; and $\rho_2 \circ \rho_1$ is an interpretation composed of ρ_2 and ρ_1. Then, the definition of $\sqsupseteq$ formally expresses the condition on type variables in $W_2.\Delta$. For any $\alpha^{\pi} \in W_2.\Delta$, the formula $W_1.\Delta, \dagger(\rho) \gg W_2.\Delta$ implies either $\alpha^{\pi'} \in W_1.\Delta$ for some $\pi' \geq \pi$, or $\pi = \mathbf{0} \wedge \alpha \in dom(\rho)$. For the latter case, $W_1.\rho = \rho \circ W_2.\rho$ implies that $W_1.\rho$ gives an interpretation of α. The substitution composition in this formula ensures that α referenced in W_2 is replaced with types given by ρ in W_1. The last condition $W_2.\Delta \succ \rho$ states that ρ provides only type substitutions that preserve typing of terms typechecked under $W_2.\Delta$. For type substitutions in ρ to preserve the typing, the type substitution lemma (Lemma 2 in Section 4.4) allows substituted types to reference type variables in $W_2.\Delta$ only when $W_2.\Delta$ assigns use $\mathbf{0}$ to them. Thus, if ρ substitutes types $\rho_{\mathrm{fst}}(\alpha)$ and $\rho_{\mathrm{snd}}(\alpha)$ for a type variable α in $W_2.\Delta$, and if $\rho_{\mathrm{fst}}(\alpha)$ or $\rho_{\mathrm{snd}}(\alpha)$ references a type variable β in $W_2.\Delta$, then we require that $W_2.\Delta$ assign the use $\mathbf{0}$ to β. Notice that $\rho_{\mathrm{fst}}(\alpha)$ and $\rho_{\mathrm{snd}}(\alpha)$ can reference type variables in $W_1.\Delta$ but not in $W_2.\Delta$ without restriction on their uses because such type variables must not occur—thus not be generalized—in terms typechecked under $W_2.\Delta$.

We defer the explanation of $(W_1, W_2) \ni W_3$ to the next section.

6.2.2 The Logical Relation. Figure 6 presents the definition of the logical relation, which consists of relations $\mathcal{R}[\![A]\!]\ W$ over pairs of results and relations $\mathcal{E}[\![A]\!]\ W$ over pairs of terms. These relations are subsets of Atom $[W, A]$ that contains terms typechecked under typing context $W.\Delta$ against type A in which type variables are replaced with the corresponding type substitutions $W.\rho_{\mathrm{fst}}$ and $W.\rho_{\mathrm{snd}}$; the top of Figure 5 presents the definition.

The result relations $\mathcal{R}[\![A]\!]\ W$ for base and of-course types are straightforward. Related results of a base type must share the same constant in their underlying parts. The result relations $\mathcal{R}[\![!A]\!]\ W$ for of-course types require related results to be constituted by unrestricted results related at A in ωW where apparent type variables may be referenced only in an unrestricted manner.

Results are related at a type variable α if and only if they are contained in the relational interpretation of α. When $\rho(\alpha) = (A_1, A_2, r)$, we write $\rho[\alpha]$ for the relational interpretation r. As seen in the definitions of World_n and Rel_n, relational interpretations in a world W require argument worlds having step indices smaller than $W.n$. Thus, the result relation $\mathcal{R}[\![\alpha]\!]\ W$ passes the later world $\blacktriangleright W$ of the current world W to the relational interpretation $W.\rho[\alpha]$. However, this definition causes an issue with the proof of compositionality; we will discuss it in Section 6.4 in detail.

The relations $\mathcal{R}[\![A \multimap B]\!]\ W$ for function types are defined as in the previous work on Kripke logical relations [Ahmed et al. 2009; Ahmed 2006; Appel and McAllester 2001; Neis et al. 2011; Pitts and Stark 1993]. Intuitively, results R_1 and R_2 related at a function type $A \multimap B$ in a world W map arguments R_1' and R_2' related at the argument type A in any future world W' of W to terms related at the return type B in W'. However, as discussed in Section 6.1, the arguments R_1' and R_2' to test the functions R_1 and R_2 are restricted not only by the type index A but also by the typing context $W'.\Delta$ of the world W' in which the applications run. Specifically, we need to prevent the case that both the functions and arguments involve open type abstractions with the same type variables because function applications composed of them violate the affine discipline in Λ^{open}. To implement this idea, we use an operation $(W_1, W_2) \Subset W$ that splits a world W into two worlds W_1 and W_2, which are the same as W except that their typing contexts are obtained by splitting the uses in $W.\Delta$. See the bottom of Figure 5 for the formal definition. The result relation $\mathcal{R}[\![A \multimap B]\!]\ W$ splits a future world W' into W_1 and W_2 and use W_2 to construct arguments R_1' and R_2'. To ensure that the functions R_1 and R_2 are still typechecked in the other world W_1, the result relation requires W_1 to be an extension of W. Notice that this requirement is not implied only by $W' \sqsupseteq W$ and $(W_1, W_2) \Subset W'$. For example, let W and W' be worlds that assign the use $\mathbf{1}$ to a type variable α. Then, $W_1.\Delta \gg W.\Delta$ holds only if $W_1.\Delta$ assigns $\mathbf{1}$ to α. However, because $(W_1, W_2) \Subset W'$ ensures only $W_1.\Delta + W_2.\Delta = W'.\Delta$ for their typing contexts, it is possible that $W_1.\Delta$ assigns the use $\mathbf{0}$ (and $W_2.\Delta$ assigns $\mathbf{1}$) to α and, as a result, $W_1.\Delta \gg W.\Delta$ does not hold. It is also notable that, as in the previous work, the result relation allows constructing arguments R_1' and R_2' and testing the applications $R_1\ R_1'$ and $R_2\ R_2'$ in *any* future world. This is crucial for ensuring monotonicity of the result relations under world extension, a key property of Kripke logical relations.

The relation $\mathcal{R}[\![\forall\alpha.A]\!]\ W$ for polymorphic type $\forall\alpha.A$ in world W relates results R_1 and R_2 such that, for any future world W' of W, given a well-formed interpretation triple (B_1, B_2, r) under $\omega W'$, the type applications $R_1\ B_1$ and $R_2\ B_2$ are related at the type A in the extended world $\{\alpha \Rightarrow (B_1, B_2, r)\} \uplus \omega W'$. The world $\omega W'$, not W', is used because the underlying values of R_1 and R_2 are closed type abstractions, and their bodies are typechecked under $\omega(W'.\Delta)$.

The term relation $\mathcal{E}[\![A]\!]\ W$ defines terms related at a type A in a world W. We write $M \longrightarrow^n M'$ when term M evaluates to term M' by n steps. Then, terms M_1 and M_2 are related by $\mathcal{E}[\![A]\!]\ W$ if and only if, for any $W' \sqsupseteq W$, when the term M_1 evaluates to a result R_1 by n steps for some $n < W'.n$, the term M_2 also evaluates to a result R_2 and the results R_1 and R_2 are related at A in the world $(W' - n)$ where the $W'.n - n$ steps are left to run. This definition only says that the term M_2 mimics the behavior of M_1 up to $W.n$ steps. As seen shortly, the logical relation relates

only terms that mimic each other up to any number of steps, thus indicating that the related terms behave equivalently.

We have defined relations for terms wherein only type variables may occur free. For extending to terms with both free type and term variables, we define $\mathcal{G}[\![\Gamma]\!]$, which determines pairs of a relational result substitution, ranged over by ς, and a world consistent with typing context Γ. A relational substitution ς maps variables to pairs of results, and substitutions ς_{fst} and ς_{snd} replace a variable x with results R_1 and R_2 when $\varsigma(x) = (R_1, R_2)$, respectively.

Let $(W, \varsigma) \in \mathcal{G}[\![\Gamma]\!]$. For unrestricted variables in the typing context Γ, ς assigns unrestricted results; thus, they must be related in the world ωW. For affine variables, ς assigns affine results. Because the results in ς are put into a single term, the typing contexts to typecheck the results in ς must be able to merge with each other; otherwise, applying ς may produce a term that violates the affine discipline. Thus, $\mathcal{G}$ assigns a tying context Δ_x for every affine variable x in Γ, requires results assigned for x to be related under Δ_x, and requires that all the typing contexts Δ_x can merge with each other. To formalize this idea, we introduce the following notation: $\sum_{x\in I} \Delta_x$ is the typing context $\Delta_{x_1} + \cdots + \Delta_{x_n}$ given a family of typing contexts $\Delta_{x_1}, \cdots, \Delta_{x_n}$ with a finite index set of variables $I = \{x_1, \cdots, x_n\}$, and $dom_{=1}(\Gamma)$ is the finite set of variables that are affine in Γ. We also write $\exists \prod_{x\in I} \Delta_x$ to existentially quantify $\Delta_{x_1}, \cdots, \Delta_{x_n}$. Then, $\mathcal{G}$ requires $W.\Delta$ to be represented by $\Delta + \sum_{x\in dom_{=1}(\Gamma)} \Delta_x$ for some typing context Δ that maintains or enlarges the uses in Γ. Every type variable in Γ is contained either in Δ or in the interpretation $W.\rho$ if its use is $\mathbf{0}$. The condition $\Gamma \succ W.\rho$ ensures that type substitutions in $W.\rho$ preserve typing of terms typechecked under Γ.

The logical approximation relation $\Gamma \vdash M_1 \preceq M_2 : A$ states that, for any world W and relational result substitution ς that respect typing context Γ, $\rho_{\mathrm{snd}}(M_2)$ mimics the behavior of term $\rho_{\mathrm{fst}}(M_1)$ in the world W. The logical (equivalence) relation $\Gamma \vdash M_1 \approx M_2 : A$ states that M_1 logically approximates M_2 and vice versa.

6.3 Properties

We show parametricity of Λ^{open} and soundness of the logical relation with respect to contextual equivalence. A key property to prove them is the compatibility lemmas, which say that the logical approximation relation is closed under the term constructors. This paper presents high-level proof sketches of the compatibility lemmas for restrictions and open type abstractions; the statements and detailed proofs of all the compatibility lemmas are in the supplementary material.

Lemma 7 (Compatibility: Restrictions). *If* $\Gamma, \alpha^{\mathbf{1}} \vdash M_1 \preceq M_2 : A$ *and* $\Gamma \vdash A$, *then* $\Gamma \vdash \nu\alpha.\, M_1 \preceq \nu\alpha.\, M_2 : A$.

Proof. For simplicity, in this sketch, suppose M_1 and M_2 to be results R_1 and R_2, respectively. Notice that it is easy to address non-result terms; see the supplementary material.

Let $(W, \varsigma) \in \mathcal{G}[\![\Gamma]\!]$. Then, it suffices to show $(\nu\alpha.\, R_1, \nu\alpha.\, R_2)_W \in \mathcal{R}[\![A]\!]\, W$.

$(W, \varsigma) \in \mathcal{G}[\![\Gamma]\!]$ implies $(W@\alpha, \varsigma) \in \mathcal{G}[\![\Gamma, \alpha^{\mathbf{1}}]\!]$. Because $\Gamma, \alpha^{\mathbf{1}} \vdash R_1 \preceq R_2 : A$, we have $(R_1, R_2)_{W@\alpha} \in \mathcal{R}[\![A]\!]\, W@\alpha$. We then obtain the conclusion by applying the following lemma:

$$\forall\, \alpha, A, W, R_1, R_2.\ \{\alpha\} \# ftv(A) \wedge \vdash W \wedge (R_1, R_2) \in \mathcal{R}[\![A]\!]\, W@\alpha \Longrightarrow (\nu\alpha.\, R_1, \nu\alpha.\, R_2) \in \mathcal{R}[\![A]\!]\, W\,.$$

This lemma can be proven by induction on A; the proof depends on the third property of relational interpretations, specifically in the case that A is a type variable. □

Lemma 8 (Compatibility: Open Type Abstractions). *If* $\Gamma_1, \alpha^{\mathbf{0}}, \Gamma_2 \vdash M_1 \preceq M_2 : !A$, *then* $\Gamma_1, \alpha^{\mathbf{1}}, \Gamma_2 \vdash \Lambda^{\circ}\langle\, \alpha, M_1 \,\rangle \preceq \Lambda^{\circ}\langle\, \alpha, M_2 \,\rangle : !\forall\alpha.A$.

Proof. This sketch supposes M_1 and M_2 to be values for simplicity. Further, it is easy to find that the values take the forms $!R_1$ and $!R_2$ for some R_1 and R_2, respectively.

Let $(W, \varsigma) \in \mathcal{G}[\![\Gamma_1, \alpha^{\mathbf{1}}, \Gamma_2]\!]$. Then, we need to show

$$(\Lambda^\circ\langle\, \alpha, !R_1\,\rangle, \Lambda^\circ\langle\, \alpha, !R_2\,\rangle)_W \in \mathcal{E}[\![!\forall\alpha.A]\!]\ W\ .$$

Because, for $i \in \{1,2\}$, $\Lambda^\circ\langle\, \alpha, !R_i\,\rangle \longrightarrow\ !\Lambda\alpha.R_i$ and $\mathsf{let}\, !x = !\Lambda\alpha.R_i\, \mathsf{in}\, x \longrightarrow \Lambda\alpha.R_i$, the proof boils down to showing that, for any $W_1 \sqsupseteq \omega W - 2$ and for any B_1, B_2, r such that $\omega W_1 \vdash (B_1, B_2, r)$,

$$((\Lambda\alpha.R_1)\, B_1, (\Lambda\alpha.R_2)\, B_2)_{\omega W_1} \in \mathcal{E}[\![A[\beta/\alpha]]\!]\ \{\beta \Rightarrow (B_1, B_2, r)\} \uplus \omega W_1 \tag{1}$$

for some fresh β; the type variable α in A is renamed to β because $\alpha \in dom(W_1.\Delta) \cup dom(W_1.\rho)$.

Let W' be a world obtained by replacing $\alpha^{\mathbf{1}} \in W.\Delta$ with $\alpha^{\mathbf{0}}$. Then, as $(W', \varsigma) \in \mathcal{G}[\![\Gamma_1, \alpha^{\mathbf{0}}, \Gamma_2]\!]$ and $\Gamma_1, \alpha^{\mathbf{0}}, \Gamma_2 \vdash\ !R_1 \preceq\ !R_2 :\ !A$, we can have $(!R_1, !R_2)_{W'} \in \mathcal{R}[\![!A]\!]\ W'$. This further implies

$$(R_1, R_2)_{W'} \in \mathcal{R}[\![A]\!]\ (\omega W' - 1)\ .$$

Here, we rename type variable α in this formula to β (this renaming is justified using $(W, \varsigma) \in \mathcal{G}[\![\Gamma_1, \alpha^{\mathbf{1}}, \Gamma_2]\!]$) and then add $\alpha^{\mathbf{0}}$ to the typing context of the renamed world. As a result, we have

$$(R_1[\beta/\alpha], R_2[\beta/\alpha])_{W'} \in \mathcal{R}[\![A[\beta/\alpha]]\!]\ (\omega(W'@\beta) - 1)\ .$$

As $\omega(W_1@\beta) \sqsupseteq \omega(W'@\beta) - 1$ (this is proven by $W_1 \sqsupseteq \omega W - 2 = \omega W' - 2$) and the world extension relation allows giving interpretation (B_1, B_2, r) to $\beta^{\mathbf{0}}$, monotonicity of the result relation implies

$$(R_1[B_1/\alpha], R_2[B_2/\alpha])_{\omega W_1} \in \mathcal{R}[\![A[\beta/\alpha]]\!]\ \{\beta \Rightarrow (B_1, B_2, r)\} \uplus \omega W_1\ .$$

This implies the formula (1) that we must prove. □

Theorem 5 (Fundamental Property). *If* $\Gamma \vdash M : A$, *then* $\Gamma \vdash M \approx M : A$.

Proof. By induction on the derivation of $\Gamma \vdash M : A$ with the compatibility lemmas. □

To define contextual equivalence, we introduce the notion of contexts, which is defined as follows.

$$\begin{array}{rcl} \mathbb{C} & ::= & [\,] \mid \lambda x.\mathbb{C} \mid \mathbb{C}\, M_2 \mid M_1\, \mathbb{C} \mid\ !\mathbb{C} \mid \mathsf{let}\, !x = \mathbb{C}\, \mathsf{in}\, M_2 \mid \mathsf{let}\, !x = M_1\, \mathsf{in}\, \mathbb{C} \mid \\ & & \nu\alpha.\, \mathbb{C} \mid \Lambda^\circ\langle\, \alpha, \mathbb{C}\,\rangle \mid \Lambda\alpha.\mathbb{C} \mid \mathbb{C}\, A \end{array}$$

A context typing judgment $\mathbb{C} : (\Gamma \vdash A) \rightsquigarrow (\Gamma' \vdash A')$ states that, given a term M such that $\Gamma \vdash M : A$, the typing judgment $\Gamma' \vdash \mathbb{C}[M] : A'$ is derivable; see the supplementary material for the formal definition of the inference rules of context typing.

Definition 4 (Contextual Equivalence). *Contextual equivalence* $\Gamma \vdash M_1 \approx_{ctx} M_2 : A$ *states that (1)* $\Gamma \vdash M_1 : A$, *(2)* $\Gamma \vdash M_2 : A$, *and (3) for any base type* ι, *constant* c *of* ι, *program context* $\mathbb{C}$ *such that* $\mathbb{C} : (\Gamma \vdash A) \rightsquigarrow (\emptyset \vdash \iota)$, $\mathbb{C}[M_1] \longrightarrow^*\ \nu\overline{\alpha_1}.\, c$ *for some* $\overline{\alpha_1}$ *if and only if* $\mathbb{C}[M_2] \longrightarrow^*\ \nu\overline{\alpha_2}.\, c$ *for some* $\overline{\alpha_2}$.

Theorem 6 (Soundness w.r.t. Contextual Equivalence). *If* $\Gamma \vdash M_1 \approx M_2 : A$, *then* $\Gamma \vdash M_1 \approx_{ctx} M_2 : A$.

Proof. Let $\mathbb{C}$ be a context such that $\mathbb{C} : (\Gamma \vdash A) \rightsquigarrow (\emptyset \vdash \iota)$. By the compatibility lemmas, $\emptyset \vdash \mathbb{C}[M_1] \approx \mathbb{C}[M_2] : \iota$ holds. Because the logical relation satisfies adequacy (i.e., given two terms related by the logical relation, the termination of a term on one side implies the termination of the term on the other side) and results related at a base type must have the same constant, we obtain that: $\mathbb{C}[M_1] \longrightarrow^*\ \nu\overline{\alpha_1}.\, c$ for some $\overline{\alpha_1}$ if and only if $\mathbb{C}[M_2] \longrightarrow^*\ \nu\overline{\alpha_2}.\, c$ for some $\overline{\alpha_2}$. □

Finally, although we believe that every well-typed term in Λ^{open} terminates, our logical relation does not imply it, which disables full reasoning about free theorems [Wadler 1989]. For example, in System F, a function of type $\forall\alpha.\alpha \to \alpha$ must be equivalent to the polymorphic identity function. However, the development in this work cannot ensure that a function f of $\forall\alpha.\alpha \multimap \alpha$ is total, and therefore we cannot prove the function f equivalent to the identity function. Nevertheless, our

logical relation can address *partial* free theorems wherein the termination property is assumed. Interested readers are referred to the supplementary material, which discusses partial free theorems for the empty type and the polymorphic identity type.

6.4 An Issue with the Result Relations at Type Variables

The result relations at type variables are defined using one-step later worlds. This gives rise to a gap between program steps and step indices because referring to results involves no computation.

Specifically, it cause a technical issue with compositionality. Initially, we have formulated compositionality for result relations as

$$\mathcal{R}[\![A[B/\alpha]]\!]\ W\ =\ \mathcal{R}[\![A]\!]\ W'$$

where $W' = \{\alpha \Rrightarrow (W.\rho_{\mathrm{fst}}(B), W.\rho_{\mathrm{snd}}(B), r)\} \uplus W$ and relational interpretation r maps a given world W to $\mathcal{R}[\![B]\!]\ W$. However, this equation does not hold in the case of $A = \alpha$. In this case, we need to prove

$$\mathcal{R}[\![B]\!]\ W\ =\ \mathcal{R}[\![\alpha]\!]\ W'\ .$$

On the right-hand side, $\mathcal{R}[\![\alpha]\!]\ W' = r(\blacktriangleright W') = \mathcal{R}[\![B]\!]\ \blacktriangleright W' = \mathcal{R}[\![B]\!]\ \blacktriangleright W$ (the last equation can be proven using the irrelevance assumption on relational interpretations because we can suppose that B is irrelevant to α). Unfortunately, in general, $\mathcal{R}[\![B]\!]\ \blacktriangleright W$ is not equal to $\mathcal{R}[\![B]\!]\ W$ which appears on the left-hand side of the equation.

To resolve this issue, instead of above r, our proof of compositionality uses an interpretation r_0 that increases the step index of a given world, that is, r_0 maps a given world W to $\mathcal{R}[\![B]\!]\ (W+1)$ (where $W+1 = (W.n+1,\ W.\Delta,\ W.\rho)$). By taking this r_0, we can have $r_0(\blacktriangleright W') = \mathcal{R}[\![B]\!]\ (\blacktriangleright W'+1) = \mathcal{R}[\![B]\!]\ W' = \mathcal{R}[\![B]\!]\ W$ (again, the last equation comes from the irrelevance of B to α). Then, we can prove the relations on the left- and right-hand sides in compositionality equivalent.

Unfortunately, this trick is ad-hoc and might not work well on languages with more complex features, such as higher-order store. A promising approach to scaling to other features is to use the technique developed by Ahmed et al. [2017], who provided a logical relation where the interpretations of type variables are only defined one step later, like ours. Ahmed et al. aligned the logical relation with program steps by computationally deferring the reference to values of type arguments substituted for type variables via *conversion*. We expect that applying their idea to our setting enables defining a scalable logical relation, but it is left for future work.

7 RELATED WORK

7.1 Type-Preserving CPS Transformation

Type preservation of (CBV) CPS transformation for the simply typed λ-calculus was discovered by Meyer and Wand [1985]. Harper and Lillibridge attempted to extend it to polymorphism [Harper and Lillibridge 1993a,b], but they discovered that extending to CBV implicit polymorphism needs some restriction even if it is pure. Specifically, Harper and Lillibridge [1993b] proved that there exists no meaning- and type-preserving CPS transformation where both the source and target languages are $\lambda_v^\forall$. The use of $\lambda_v^\forall$ as the target enables the contamination of the CPS image with "exotic" terms corresponding to programming facilities, such as `call/cc`, that are inexpressive in the source language and cannot safely cooperate with unrestricted implicit polymorphism. These unsafe exotic terms cause the evaluation of certain CPS terms to go "wrong." As $\lambda_v^\forall$ is type safe, such CPS terms cannot be well typed. We address this problem by restricting the use of continuations in the CPS image to be affine, which excludes the unsafe exotic terms. This restriction might allow the existence of exotic terms using continuations at most once, but our result indicates that they safely cooperate with CPS terms.

7.2 CPS Transformation with Linearity

Berdine et al. [2001, 2002] formalized the idea that pure terms use continuations only linearly via CBV CPS transformation to the linear λ-calculus of Barber and Plotkin [1996]. They also showed that certain control constructs can be expressed by linearly used continuations. Berdine [2004] extended this idea to the affine setting. Hasegawa [2002] generalized the notion of linearly used continuations to that of *linearly used effects* and provided the monadic transformation from the computational λ-calculus [Moggi 1989] to the linear λ-calculus. Thielecke [2003] found that the invocation of `call/cc` can be characterized by two type-preserving CPS transformations: one with answer type polymorphism and the other with linear types. Thielecke equipped the source language with implicit polymorphism but imposed the value restriction. Thielecke [2004] provided only CBN CPS transformation with answer type polymorphism for a language supporting both of `call/cc` and unrestricted implicit polymorphism. Target languages with linear/affine types can characterize source terms more precisely [Hasegawa 2002], but, to the best of our knowledge, no prior work has examined linear/affine typing to achieve type-preserving CPS transformation for unrestricted implicit polymorphism. We utilize the linear use of continuations in source terms for that aim.

7.3 Logical Relations and Parametricity

Logical relations are well-known techniques to reason about the properties of programs. Since the initial development by Tait [1967] and the seminal work by Plotkin [1973], logical relations have been extended to a variety of programming facilities and applications. Here, we discuss only the previous work closely related to ours. Reynolds [1983] established relational parametricity, which ensures that polymorphic terms behave equivalently no matter how they are instantiated. This is a simple but powerful reasoning principle; for example, it can prove theorems about polymorphic functions [Wadler 1989]. As mentioned in Section 1.2, parametricity can also enhance CPS transformation.

We proved parametricity of Λ^{open} by developing a step-indexed Kripke logical relation. Kripke logical relations have been proven powerful enough to deal with programming facilities with some circularity, and an established approach to circularity is to involve step indices in possible worlds [Ahmed et al. 2009; Ahmed 2006; Appel and McAllester 2001; Neis et al. 2011]. We also employed this approach to avoid circularity between worlds and relational interpretations.

Zhao et al. [2010] provided a logical relation for a polymorphic, linearly typed language and proved parametricity and soundness of the logical relation with respect to contextual equivalence. As in the present work, their logical relation may also relate open terms—in particular, it can relate terms containing free term and type variables—to exploit linearity of their language. Their logical relation is indexed by typing contexts and allows their augmentation, as ours is indexed by possible worlds that contain typing contexts. However, unlike our work, they did not consider substitution for free (type) variables in running terms. Thus, their handling of free (type) variables is similar to that of dynamic name creation in the work by Pitts and Stark [1993].

7.4 Decomposition of Type Abstraction

A key idea of our CPS target language Λ^{open} is the decomposition of the type abstraction mechanism. Montagu and Rémy [2009] took a similar approach to decomposing the module unpacking construct with existential types, aiming at the simplification of a module language. As we decompose type abstraction into restrictions and open type abstractions, their decomposition derives two more atomic constructs for unpacking: one for binding existential type variables and the other for linking the existential type variables with the witness types of unpacked modules. They managed

existential type variables as *linear* resources to avoid linking the same type variable with different witness types. In their work, all functions are unrestricted. To ensure the linearity of existential type variables in such a situation, their language prevents functions from using the existential type variables bound outside the functions. This is more restrictive than the use of linear types (with which linear resources can occur within linear functions), but it enables avoiding some circularity and easily translating their language to System F. However, their approach is inadequate for our setting because we need to generalize type variables within continuation functions outside which the type variables are bound. The use of affine types allows more flexible use of type variables.

8 CONCLUSION

This work studied type-preserving CBV CPS transformation for a pure language with unrestricted implicit polymorphism. We identified the challenge of scope intrusion, which happens by lifting terms under type variable binders to the top of a program, and addressed it by defining a new type-safe CPS target language Λ^{open} with restrictions, open type abstractions, and affine types. Restrictions and open type abstractions can defer binding of type variables in closed type abstraction, and affine types enforce the type-safe use of open type abstractions. We then provided a CPS transformation from Curry-style CBV System F to Λ^{open} and proved the CPS transformation type and meaning preserving. Aiming at establishing parametricity of Λ^{open}, we also constructed a parametric, step-indexed Kripke logical relation for terms in which free type variables may occur and may be replaced later via open type abstractions. We captured this characteristic behavior of Λ^{open} with worlds and the world extension relation and proved the Fundamental Property of the logical relation and its soundness with respect to contextual equivalence. We believe that this study has opened up the possibility for more languages to gain the benefits of type-preserving CPS transformation.

Our work depends on the linearity of continuations, but it is violated in the presence of some effects. For example, general control operators allow invoking continuations multiple times, and, even if captured continuations are restricted to be invoked only once, multi-shot continuations can be implemented with higher-order store [Friedman and Haynes 1985]. Because naively introducing these effects to Curry-style System F results in being unsound [Gordon et al. 1979; Harper and Lillibridge 1991], we need to restrict, e.g., polymorphism or effects. It is a crucial future work to provide type-preserving CPS transformation for effectful languages with restrictions other than the value restriction.

The following are other future directions. An application of our work is to design typed IRs for implicitly polymorphic CBV languages that do not adopt the value restriction. Extending the development to other IR forms is also attractive. In particular, A-normal form translation [Flanagan et al. 1993] has a problem similar to CPS transformation because it also lifts redexes to the top of a program. Applying our approach to A-normal form translation is promising to solve the problem. Our CPS transformation also suggests type safety of a polymorphic language where continuations are used only once. Therefore, we conjecture that a polymorphic language only with a *one-shot* control operator (and without higher-order store) is safe. Finally, we are curious about extending our work to dependent typing and to programming facilities that allow evaluation beneath binding constructs, such as staged computation.

ACKNOWLEDGMENTS

We would like to thank Atsushi Igarashi for advice and the anonymous reviewers at POPL 2021 and ICFP 2021 for their close reading and valuable comments. This work was supported in part by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST and JSPS KAKENHI Grant Numbers JP19K20247 (Sekiyama) and JP19K20211 (Tsukada).

REFERENCES

Amal Ahmed and Matthias Blume. 2011. An equivalence-preserving CPS translation via multi-language semantics. In *ACM SIGPLAN international conference on Functional Programming, ICFP 2011*. 431–444. https://doi.org/10.1145/2034773.2034830

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent representation independence. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*. 340–353. https://doi.org/10.1145/1480881.1480925

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for free for free: parametricity, with and without types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 39:1–39:28. https://doi.org/10.1145/3110283

Amal J. Ahmed. 2006. Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*. Springer, 69–83. https://doi.org/10.1007/11693024_6

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press.

Andrew W. Appel and David B. MacQueen. 1991. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming, 3rd International Symposium, PLILP 1991, Proceedings*. 1–13. https://doi.org/10.1007/3-540-54444-5_83

Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* 23, 5 (2001), 657–683. https://doi.org/10.1145/504709.504712

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018*. 56–65. https://doi.org/10.1145/3209108.3209189

Andrew Barber and Gordon D. Plotkin. 1996. *Dual Intuitionistic Linear Logic*. Technical Report. ECS-LFCS-96-347.

Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, and Hayo Thielecke. 2001. Linearly Used Continuations. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)*. 47–54. https://www.microsoft.com/en-us/research/publication/linearly-used-continuations/

Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, and Hayo Thielecke. 2002. Linear Continuation-Passing. *High. Order Symb. Comput.* 15, 2-3 (2002), 181–208. https://doi.org/10.1023/A:1020891112409

Joshua James Berdine. 2004. *Linear and Affine Typing of Continuation-Passing Style*. Technical Report. RR-04-04.

Dariusz Biernacki, Olivier Danvy, and Kevin Millikin. 2015. A Dynamic Continuation-Passing Style for Dynamic Delimited Continuations. *ACM Trans. Program. Lang. Syst.* 38, 1 (2015), 2:1–2:25. https://doi.org/10.1145/2794078

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2018. Type-preserving CPS translation of Σ and Π types is not not possible. *Proc. ACM Program. Lang.* 2, POPL (2018), 22:1–22:33. https://doi.org/10.1145/3158110

Iliano Cervesato and Frank Pfenning. 1996. A Linear Logical Framework. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, LICS 1996*. 264–275. https://doi.org/10.1109/LICS.1996.561339

Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with continuations, or without? whatever. *Proc. ACM Program. Lang.* 3, ICFP (2019), 79:1–79:28. https://doi.org/10.1145/3341643

Olivier Danvy. 1992. *Three Steps for the CPS Transformation*. Technical Report. CIS-92-2.

Olivier Danvy and Andrzej Filinski. 1989. *A functional abstraction of typed contexts*. Technical Report.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. 151–160. https://doi.org/10.1145/91556.91622

Olivier Danvy and John Hatcliff. 1992. Thunks (Continued). In *Actes WSA'92 Workshop on Static Analysis (Bordeaux, France), September 1992, Laboratoire Bordelais de Recherche en Informatique (LaBRI), Proceedings*. 3–11.

Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce F. Duba. 1986. Reasoning with Continuations. In *Proceedings of the Symposium on Logic in Computer Science (LICS '86)*. 131–141.

Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theorical Computer Science* 103, 2 (1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

Michael J. Fischer. 1972. Lambda Calculus Schemata. In *Proceedings of ACM Conference on Proving Assertions about Programs*. 104–109. https://doi.org/10.1145/800235.807077

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*. 237–247. https://doi.org/10.1145/155090.155113

Matthew Fluet and Stephen Weeks. 2001. Contification Using Dominators. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01)*. 2–13. https://doi.org/10.1145/507635.507639

Daniel P. Friedman and Christopher T. Haynes. 1985. Constraining Control. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 1985*, Mary S. Van Deusen, Zvi Galil, and Brian K. Reid (Eds.). ACM Press, 245–254. https://doi.org/10.1145/318593.318654

Jacques Garrigue. 2004. Relaxing the Value Restriction. In *Functional and Logic Programming, 7th International Symposium, FLOPS 2004, Proceedings*. 196–213. https://doi.org/10.1007/978-3-540-24754-8_15

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

J. Y. Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de Doctorat d'État. Université Paris 7.

Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF.* Lecture Notes in Computer Science, Vol. 78. Springer. https://doi.org/10.1007/3-540-09724-4

Robert Harper and Mark Lillibridge. 1991. ML with callcc is unsound. Announcement on the types electronic forum. https://www.cis.upenn.edu/~bcpierce/types/archives/1991/msg00034.html

Robert Harper and Mark Lillibridge. 1993a. Explicit Polymorphism and CPS Conversion. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 206–219. https://doi.org/10.1145/158511.158630

Robert Harper and Mark Lillibridge. 1993b. Polymorphic Type Assignment and CPS Conversion. *Lisp and Symbolic Computation* 6, 3-4 (1993), 361–380.

Masahito Hasegawa. 2002. Linearly Used Effects: Monadic and CPS Transformations into the Linear Lambda Calculus. In *Functional and Logic Programming, 6th International Symposium, FLOPS 2002.* 167–182. https://doi.org/10.1007/3-540-45788-7_10

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017.* 18:1–18:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.18

My Hoang, John C. Mitchell, and Ramesh Viswanathan. 1993. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93).* 15–25. https://doi.org/10.1109/LICS.1993.287604

Atsushi Igarashi and Naoki Kobayashi. 1997. Type-Based Analysis of Communication for Concurrent Programming Languages. In *Static Analysis, 4th International Symposium, SAS '97.* 187–201. https://doi.org/10.1007/BFb0032742

Yukiyoshi Kameyama and Takuo Yonezawa. 2008. Typed Dynamic Control Operators for Delimited Continuations. In *Functional and Logic Programming, 9th International Symposium, FLOPS 2008.* 239–254. https://doi.org/10.1007/978-3-540-78969-7_18

Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming* 27 (2017), e7. https://doi.org/10.1017/S0956796816000320

Andrew Kennedy. 2007. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007.* 177–190. https://doi.org/10.1145/1291151.1291179

Naoki Kobayashi. 2013. Model Checking Higher-Order Programs. *J. ACM* 60, 3 (2013), 20:1–20:62. https://doi.org/10.1145/2487241.2487246

Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A graph-based higher-order intermediate representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015.* 202–212. https://doi.org/10.1109/CGO.2015.7054200

Daniel Leivant. 1983. Polymorphic Type Inference. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages, POPL 1983.* 88–98. https://doi.org/10.1145/567067.567077

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. 2020. The OCaml system release 4.10: Documentation and user's manua. https://caml.inria.fr/pub/docs/manual-ocaml/

Xavier Leroy and Pierre Weis. 1991. Polymorphic Type Inference and Assignment. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages.* 291–302. https://doi.org/10.1145/99583.99622

John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 1995. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. In *Eleventh Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 1995.* 370–392. https://doi.org/10.1016/S1571-0661(04)00022-2

Albert R. Meyer and Mitchell Wand. 1985. Continuation Semantics in Typed Lambda-Calculi (Summary). In *Logics of Programs.* 219–224. https://doi.org/10.1007/3-540-15648-8_17

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989.* 14–23. https://doi.org/10.1109/LICS.1989.39155

Benoît Montagu and Didier Rémy. 2009. Modeling abstract types in modules with open existential types. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009.* 354–365. https://doi.org/10.1145/1480881.1480926

James H. Morris. 1969. *Lambda-calculus models of programming languages.* Ph.D. Dissertation. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/64850

J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (1999), 527–568. https://doi.org/10.1145/319301.319345

Georg Neis, Derek Dreyer, and Andreas Rossberg. 2011. Non-parametric parametricity. *J. Funct. Program.* 21, 4-5 (2011), 497–562. https://doi.org/10.1017/S0956796811000165

Max S. New, Dustin Jamner, and Amal Ahmed. 2020. Graduality and parametricity: together again for the first time. *Proc. ACM Program. Lang.* 4, POPL (2020), 46:1–46:32. https://doi.org/10.1145/3371114

Simon L. Peyton Jones and Philip Wadler. 1993. Imperative Functional Programming. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 71–84. https://doi.org/10.1145/158511.158524

Andrew M. Pitts and Ian David Bede Stark. 1993. Observable Properties of Higher Order Functions that Dynamically Create Local Names, or What's new?. In *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS 1993*. 122–141. https://doi.org/10.1007/3-540-57182-5_8

Gordon D. Plotkin. 1973. *Lambda-definability and logical relations.* Technical Report.

Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2 (ACM '72)*. 717–740. https://doi.org/10.1145/800194.805852

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*. 408–423. https://doi.org/10.1007/3-540-06859-7_148

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*. 513–523.

John C. Reynolds. 1993. The Discoveries of Continuations. *Lisp and Symbolic Computation* 6, 3-4 (1993), 233–248.

Amr Sabry and Matthias Felleisen. 1992. Reasoning About Programs in Continuation-Passing Style. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992*. 288–298. https://doi.org/10.1145/141471.141563

Taro Sekiyama and Atsushi Igarashi. 2019. Handling Polymorphic Algebraic Effects. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings*. 353–380. https://doi.org/10.1007/978-3-030-17184-1_13

Taro Sekiyama, Takeshi Tsukada, and Atsushi Igarashi. 2020. Signature restriction for polymorphic algebraic effects. *Proc. ACM Program. Lang.* 4, ICFP (2020), 117:1–117:30. https://doi.org/10.1145/3408999

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658

Hayo Thielecke. 2003. From control effects to typed continuation passing. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 139–149. https://doi.org/10.1145/640128.604144

Hayo Thielecke. 2004. Answer Type Polymorphism in Call-by-Name Continuation Passing. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004*. 279–293. https://doi.org/10.1007/978-3-540-24725-8_20

Mads Tofte. 1990. Type Inference for Polymorphic References. *Inf. Comput.* 89, 1 (1990), 1–34. https://doi.org/10.1016/0890-5401(90)90018-D

David N. Turner, Philip Wadler, and Christian Mossin. 1995. Once Upon a Type. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995*. 1–11. https://doi.org/10.1145/224164.224168

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989*. 347–359. https://doi.org/10.1145/99370.99404

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. https://doi.org/10.1006/inco.1994.1093

Jianzhou Zhao, Qi Zhang, and Steve Zdancewic. 2010. Relational Parametricity for a Polymorphic Linear Lambda Calculus. In *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6461)*, Kazunori Ueda (Ed.). Springer, 344–359. https://doi.org/10.1007/978-3-642-17164-2_24