Shifting the Blame

A blame calculus with delimited control

Taro Sekiyama* Atsushi Igarashi Soichiro Ueda

Kyoto University

Shifting the Blame

A blame calculus with delimited control

Taro Sekiyama* Atsushi Igarashi Soichiro Ueda

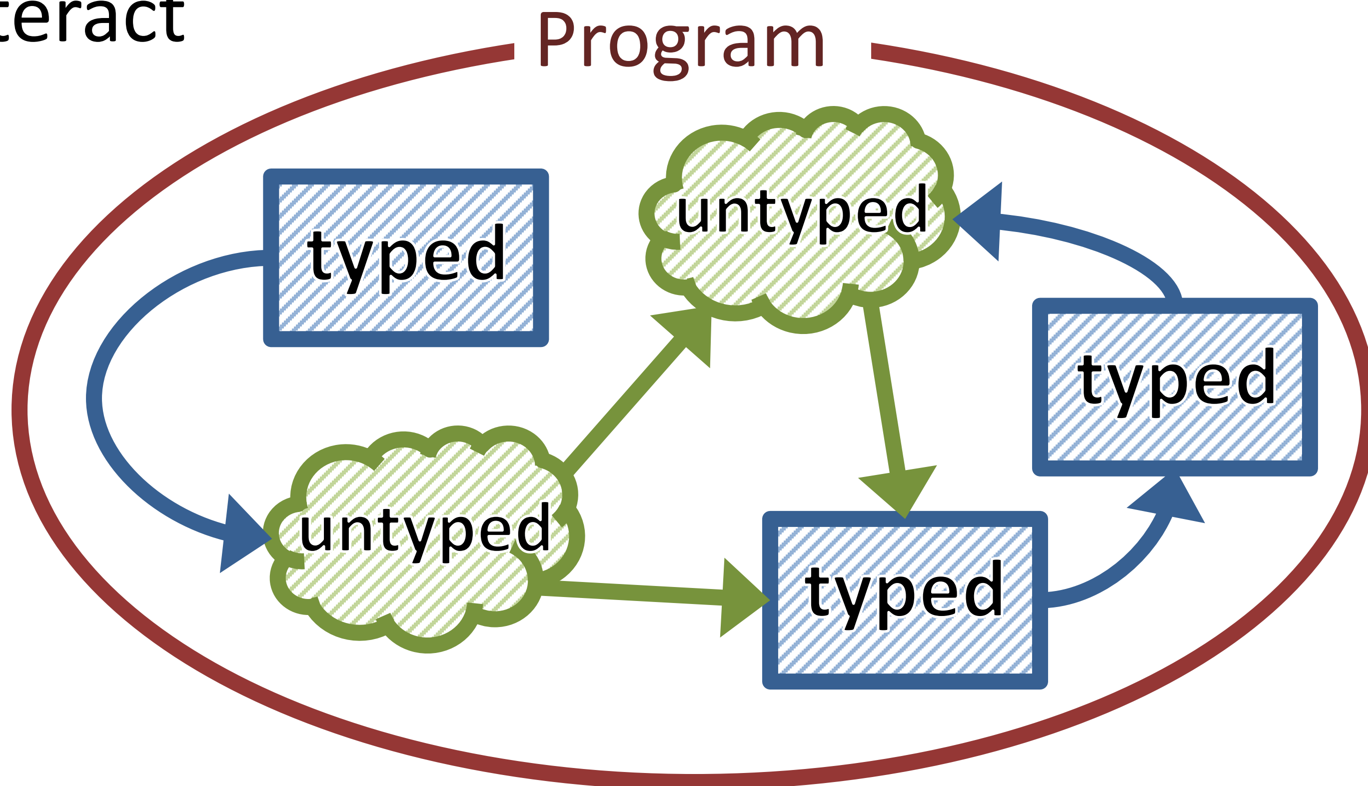
Kyoto University

Gradual typing

[Tobin-Hochstadt&Felleisen'06,Siek&Taha'07]

Integration of static and dynamic typing

- Typed and untyped code can coexist and interact



Blame calculus

[Tobin-Hochstadt&Felleisen'06,Wadler&Findler'09]

A typed lambda calculus to model intermediate languages for gradual typing

- The Dynamic type (Dyn for short)
  - The type for untyped code
- Casts (type coercions) $s : S \Longrightarrow T$
  - coerce term s of type S to type T
  - are used to monitor value flows between typed and untyped code

Example

Gradually typed lang.

```
let x =
 if … then succ
       else true

let y : string =
 if x then
  “true”
 else  “false”
```

Example

Gradually typed lang.

```
let x =
 if … then succ
       else true

let y : string =
 if x then
  “true”
 else  “false”
```

Example

Gradually typed lang.

```
let x =
 if … then succ
       else true

let y : string =
 if x then
   "true"
 else  "false"
```

Example

Gradually typed lang.

```
let x =
 if ... then succ
         else true
let y : string =
 if x then
   "true"
 else  "false"
```

Example

Gradually typed lang.

```
let x =
 if … then succ
        else true
let y : string =
 if x then
   "true"
 else  "false"
```

Example

Gradually typed lang.

```
let x =
  if ... then succ
         else true
let y : string =
  if x then
    "true"
  else  "false"
```

Blame calculus

let x : **Dyn** =

if...then succ : **int→int** $\Longrightarrow$ **Dyn**

else true : **bool** $\Longrightarrow$ **Dyn**

let y : string =

if (x : **Dyn** $\Longrightarrow$ **bool**) then

"true"

else "false"

Example

Untyped code is given Dyn

Gradually typed lang.

```
let x =
  if … then succ
        else true
let y : string =
  if x then
   "true"
  else  "false"
```

Blame calculus

```
let x : Dyn =
if…then succ : int→int ⟹ Dyn
   else true : bool ⟹ Dyn
let y : string =
 if (x : Dyn ⟹ bool) then
  "true"
 else "false"
```

Example

Untyped code is given Dyn

Injection of typed values into untyped code

Gradually typed lang.

```
let x =
 if … then succ
       else true
let y : string =
 if x then
  "true"
 else  "false"
```

```
let x : Dyn =
if…then succ : int→int ⟹ Dyn
    else true : bool ⟹ Dyn
let y : string =
 if (x : Dyn ⟹ bool) then
  "true"
 else "false"
```

Example

Untyped code is given Dyn

Injection of typed values into untyped code

Gradually typed lang.

Run-time test to check that the dynamic value is a Boolean

```
le…
le…
 if x then
   "true"
 else  "false"
```

```
let x : Dyn =
if…then succ : int→int ⟹ Dyn
    else true : bool ⟹ Dyn

let y : string =
 if (x : Dyn ⟹ bool) then
  "true"
 else "false"
```

What blame calculus should guarantee

Type Soundness

If something wrong happens,
it is detected as cast failure

Blame Theorem

Statically typed terms are never sources of cast failure

What blame calc

E.g., an integer is called as a function

Type Soundness

If something wrong happens, it is detected as cast failure

Blame Theorem

Statically typed terms are never sources of cast failure

This work

- Extends the blame calculus with delimited-control operators ***shift/reset***
  - A new form of cast to monitor capturing and calling continuations
- Defines continuation passing style (CPS) transformation for the extended calculus
- Investigates three properties
  - Type soundness
  - Blame Theorem
  - Soundness of the CPS transformation

This

can implement various
control effects

- Extends the blame calculus delimited-control operators ***shift/reset***
  - A new form of cast to monitor capturing and calling continuations
- Defines continuation passing style (CPS) transformation for the extended calculus
- Investigates three properties
  - Type soundness
  - Blame Theorem
  - Soundness of the CPS transformation

Challenge for Type Soundness

All value flows between typed and untyped parts have to be monitored by casts

Challenge for Ty

All value flows between typed and untyped parts have to be monitored by casts

Challenge for Ty

All value flows between typed and untyped parts have to be monitored by casts

It isn't trivial to satisfy this property

Challenge for Ty

All value flows between typed and untyped parts have to be monitored by casts

It isn't trivial to satisfy this property

Because terms with control operators can take more actions

- To return a value
- To capture a continuation
- To call a continuation

Challenge for Ty

Paraphrase of
Type Soundness

All value flows between typed and untyped parts have to be monitored by casts

It isn’t trivial to satisfy this property

Because terms with control operators can take more actions

- To return a value
- To capture a continuation
- To call a continuation

monitored by the early
blame calculus

Challenge for Ty

Paraphrase of
Type Soundness

All value flows between typed and
be monitored by casts
sfy this property

We design a system to
monitor these

Becau rms with control operators can take
more ac ns

- To retur a value
- To capture a continuation
- To call a continuation

monitored by the early
blame calculus

Outline

1. Introduction
2. Background: blame calculus (without shift/reset)
3. Problem with control operators
4. Our extension of the blame calculus with shift/reset

Blame calculus [Wadler&Findler'09]

$\lambda^{\rightarrow}$ with Dyn and casts

Types S, T ::= int | … | S → T | Dyn

Terms s, t ::= 1 | + | … | λx.s | s t |

$s : S \Longrightarrow T$

Blame calculus [Wadler&Findler'09]

$\lambda^{\rightarrow}$ with Dyn and casts

Types S, T ::= int | … | S → T | Dyn

Terms s, t ::= 1 | + | … | λx.s | s t |

s : S ⟹ T

the type for untyped code

monitors that the value of s at S can behave as T

Notation

typed code

untyped code

cast

Blame calculus

let x : **Dyn** =

if…then succ : **int→int** ⇒ **Dyn**

else true : **bool** ⇒ **Dyn**

let y : string =

if (x : **Dyn** ⇒ **bool**) then

"true"

else "false"

Notation

typed code

untyped code

cast

Blame calculus

let x : Dyn =
if…then succ : int→int ⇒ Dyn
else true : bool ⇒ Dyn
let y : string =
if (x : Dyn ⇒ bool) then
"true"
else "false"

Notation

typed code

untyped code

cast

Blame calculus

```
let x : Dyn =
if...then succ : int→int ⇒ Dyn
     else true : bool ⇒ Dyn
let y : string =
 if (x : Dyn ⇒ bool) then
  "true"
 else "false"
```

Notation

typed code

untyped code

cast

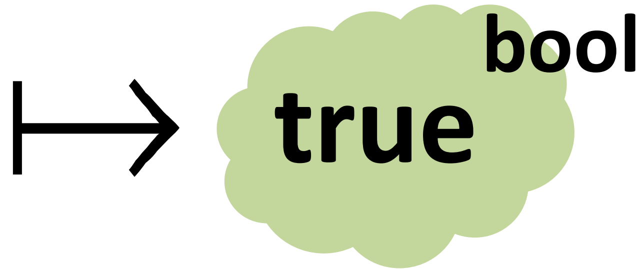
Blame calculus

let x : **Dyn** =
if…then succ : **int→int** ⟹ **Dyn**
else true : **bool** ⟹ **Dyn**

let y : string =
if (x : **Dyn** ⟹ **bool**) then
"true"
else "false"

Cast semantics

true : bool $\Rightarrow$ **Dyn**



- Casts from base types to Dyn always succeed
- Result values have the target value (**true**) and its type (**bool**)

Cast semantics

$$\text{true}^{\text{bool}} : \text{Dyn} \Rightarrow \text{bool} \;\longmapsto\; \text{true}$$

- Casts from Dyn succeed if the tagged type matches with the target type

Cast semantics

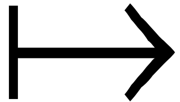
$\text{true}^{\text{bool}} : \text{Dyn} \Rightarrow \text{bool} \mapsto$ **Cast failure**

bool ≠ int

- Casts from Dyn fail if the tagged type ***doesn't*** match with the target type

Cast semantics

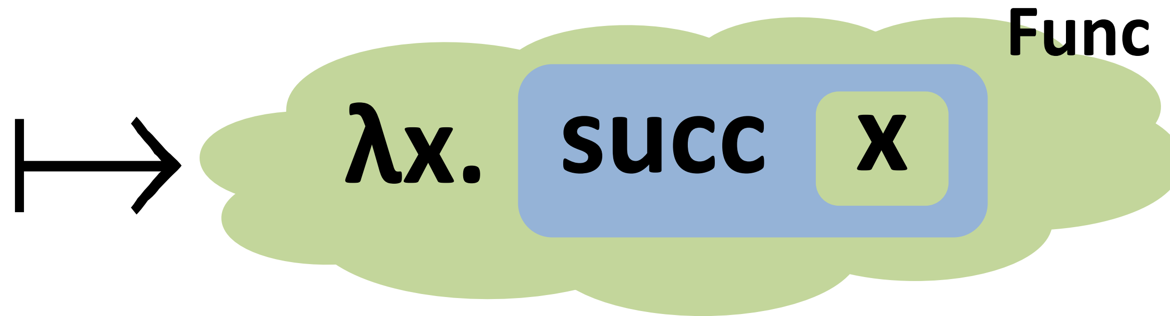
succ : (int→int) ⇒ Dyn



- Casts from function types to Dyn generate wrappers of the target function
- **All value flows between typed and untyped parts are monitored by casts**

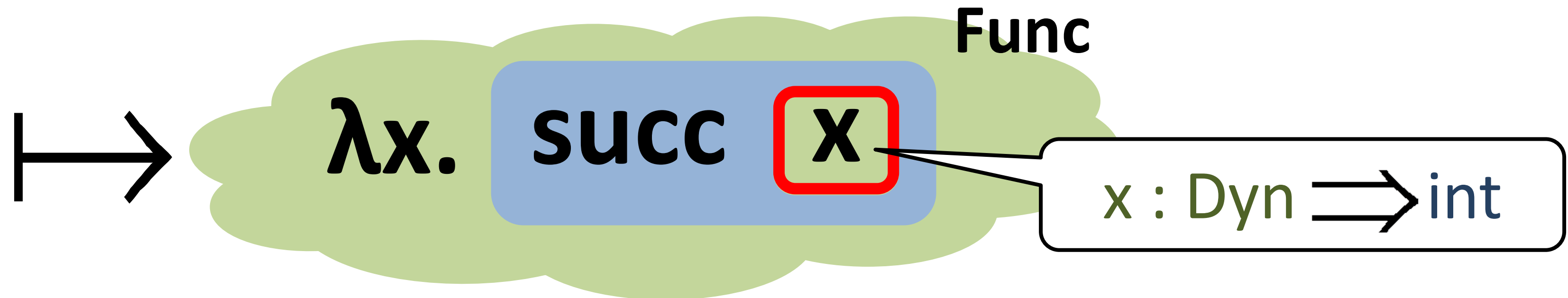
Cast semantics

$\mathsf{succ} : (\mathsf{int} \rightarrow \mathsf{int}) \Rightarrow \mathsf{Dyn}$



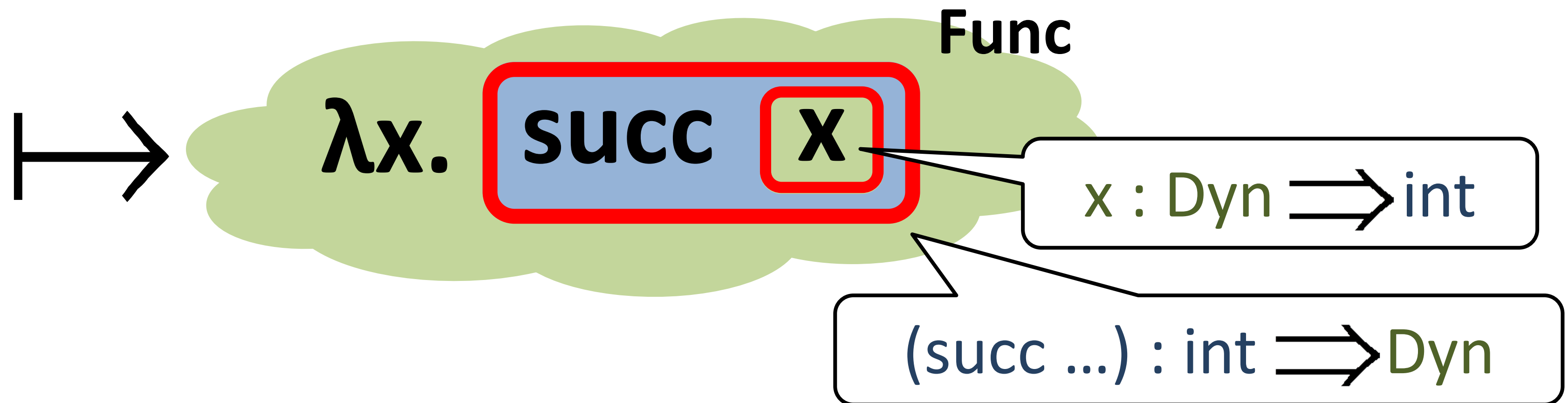
- Casts from function types to Dyn generate wrappers of the target function
- **All value flows between typed and untyped parts are monitored by casts**

Cast semantics



- Casts from function types to Dyn generate wrappers of the target function
- **All value flows between typed and untyped parts are monitored by casts**

Cast semantics



- Casts from function types to Dyn generate wrappers of the target function
- **All value flows between typed and untyped parts are monitored by casts**

Outline

1. Introduction
2. Background: blame calculus (without shift/reset)
3. Problem with control operators
4. Our extension of the blame calculus with shift/reset

Problem with control operators

NOT all value flows between typed and untyped code are monitored

because the standard cast semantics cannot monitor capturing and calling continuations

Shift and reset

CPS-based operators to manipulate “delimited” continuations

- Reset $\langle s \rangle$ delimits continuations in s
- Shift ($\mathcal{S}$ k. s) captures continuations up to the closest reset as k

$$\langle E[\mathcal{S}\ k.\ s] \rangle \longmapsto \langle s[k := \lambda x. \langle E[x] \rangle] \rangle$$

where E is an evaluation context without reset

Shift and reset

used to impl. exceptions, backtracking, monads, etc.

CPS-based operators to manipula continuations

- Reset $\langle s \rangle$ delimits continuations in s
- Shift ($\mathbb{S}$ k. s) captures continuations up to the closest reset as k

$$\langle E[\mathbb{S}\ k.\ s] \rangle \longmapsto \langle s[k{:=}\lambda x.\langle E[x] \rangle] \rangle$$

where E is an evaluation context without reset

Reduction example

$$\langle 3 + (\mathcal{S}\ k.\ (k\ 1) == 4)\rangle$$

$$= \langle E[\mathcal{S}\ k.\ (k\ 1) == 4]\rangle \text{ where } \mathbf{E} = \mathbf{3 + []}$$

$$\longmapsto \langle (k\ \ 1) == 4\rangle$$

Reduction example

$\langle 3 + (\mathcal{S}\ k.\ (k\ 1) == 4)\rangle$

$= \langle E[\mathcal{S}\ k.\ (k\ 1) == 4]\rangle$ where $E = 3 + [\,]$

$\mapsto \langle (k\ \ 1) == 4\rangle$

Reduction example

$$\langle 3 + (\mathbf{S}\ k.\ (k\ 1) == 4) \rangle$$

$$= \langle E[\mathbf{S}\ k.\ (k\ 1) == 4] \rangle \text{ where } \mathbf{E} = \mathbf{3 + []}$$

$$\longmapsto \langle ((\lambda x.\langle 3 + x \rangle)\ 1) == 4 \rangle$$

Reduction example

$\langle 3 + (\mathcal{S}\ k.\ (k\ 1) == 4) \rangle$

$= \langle E[\mathcal{S}\ k.\ (k\ 1) == 4] \rangle$ where $E = 3 + []$

$\longmapsto \langle ((\lambda x. \langle 3 + x \rangle)\ 1) == 4 \rangle$

captured continuation

$\longmapsto \langle \langle 3+1 \rangle == 4 \rangle$

$\longmapsto^*$ **true**

Problem of the standard cast semantics in the presence of shift/reset

v = λx:int. 3 + (S k. (k 1) == x)

⟨f (v 4)⟩

Problem of the standard cast semantics in the presence of shift/reset

v = λx:int. 3 + (S k. (k 1) == x)

⟨f (v 4)⟩ ⟼ ⟨f ((λy. v y) 4)⟩

Problem of the standard cast semantics in the presence of shift/reset

$v = \lambda x{:}int.\ 3 + (S\ k.\ (k\ 1) == x)$

$\langle f\ (v\ 4)\rangle \longmapsto \langle f\ ((\lambda y.\ v\ y)\ 4)\rangle$

$\longmapsto \langle f\ (v\ 4)\rangle$

Problem of the standard cast semantics in the presence of shift/reset

$v = \lambda x{:}int.\ 3 + (S\ k.\ (k\ 1) == x)$

$\langle f\ (v\ 4)\rangle \longmapsto \langle f\ ((\lambda y.\ v\ y)\ 4)\rangle$

$\longmapsto \langle f\ (v\ 4)\rangle$

$4 : Dyn \Rightarrow int$

Problem of the standard cast semantics in the presence of shift/reset

v = λx:int. 3 + (S k. (k 1) == x)

⟨f (v 4)⟩ ⟼ ⟨f ((λy. v y) 4)⟩

⟼ ⟨f (v 4)⟩ ⟼ ⟨f (v 4)⟩

Problem of the standard cast semantics in the presence of shift/reset

v = λx:int. 3 + (S k. (k 1) == x)

⟨f (v 4)⟩ ↦ ⟨f ((λy. v y) 4)⟩

↦ ⟨f (v 4)⟩ ↦ ⟨f (v 4)⟩

↦ ⟨f (3 + (S k. (k 1) == 4))⟩

Problem of the standard cast semantics in the presence of shift/reset

…

$\longmapsto$ ⟨f (3 + (S k. (k 1) == 4))⟩

Problem of the standard cast semantics in the presence of shift/reset

...

$\mapsto \langle f\ (3 + (\mathcal{S}\ k.\ (k\ 1) == 4)) \rangle$

$\mapsto \langle (k \quad 1) == 4 \rangle$

Problem of the standard cast semantics in the presence of shift/reset

...

$\mapsto$ ⟨f (3 + (S k. (k 1) == 4))⟩

$\mapsto$ ⟨ (k 1) == 4 ⟩

captured continuation

Problem of the standard cast semantics in the presence of shift/reset

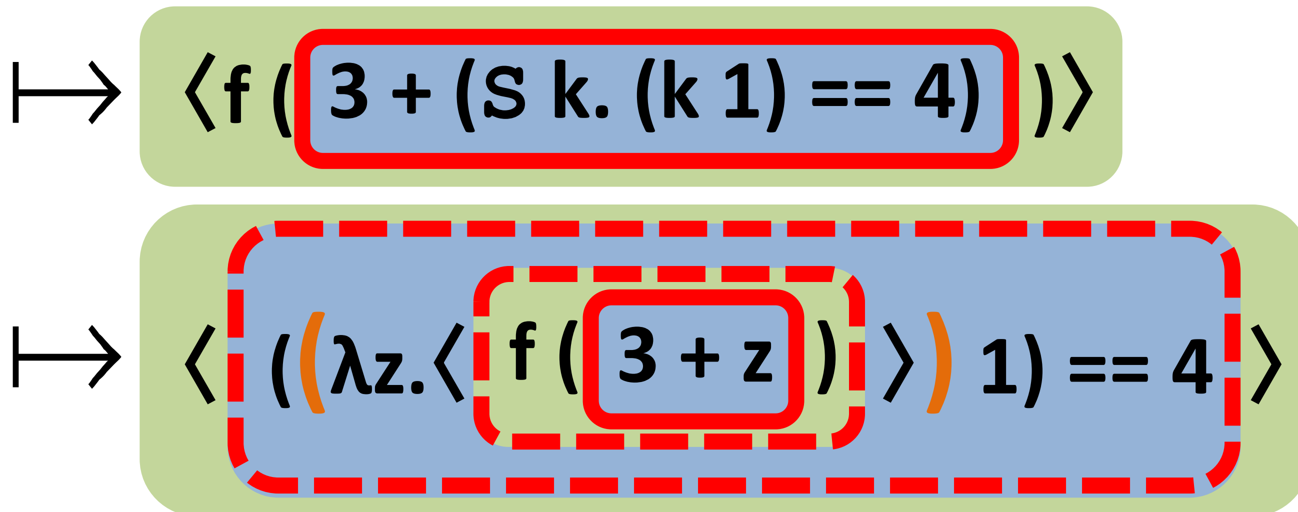
…

$\longmapsto$ ⟨f (3 + (S k. (k 1) == 4))⟩

$\longmapsto$ ⟨ ((λz.⟨ f (3 + z) ⟩) 1) == 4 ⟩

captured continuation

Problem of the standard cast semantics in the presence of shift/reset



NOT all value flows between typed and untyped parts are monitored

Outline

1. Introduction
2. Background: blame calculus (without shift/reset)
3. Problem with control operators
4. Our extension of the blame calculus with shift/reset

How does the problem happen?

An untyped continuation is captured and sent to typed code

v = **λx:int. 3 + (S k. (k 1) == x)**

...

$\longmapsto$ **⟨f (3 + (S k. (k 1) == 4))⟩**

How does the problem happen?

An untyped continuation is captured and sent to typed code

Captured continuations are expected to be typed

v = **λx:int. 3 + (S k. (k 1) == x)**

...

$\mapsto$ **⟨f (3 + (S k. (k 1) == 4))⟩**

How does the problem happen?

An untyped continuation is captured and sent to typed code

Captured continuations are expected to be typed

v = **λx:int. 3 + (S k. (k 1) == x)**

The captured continuation **f (3 + z)** is untyped

...

$\mapsto$ **⟨f (3 + (S k. (k 1) == 4))⟩**

Our solution

Changing the cast semantics so that:

- ***Typed*** code captures only ***typed*** continuations
- ***Untyped*** code captures only ***untyped*** continuations

Our cast semantics

⟨f (v' 4)⟩

Our cast semantics

$\langle f\ (v'\ 4) \rangle$

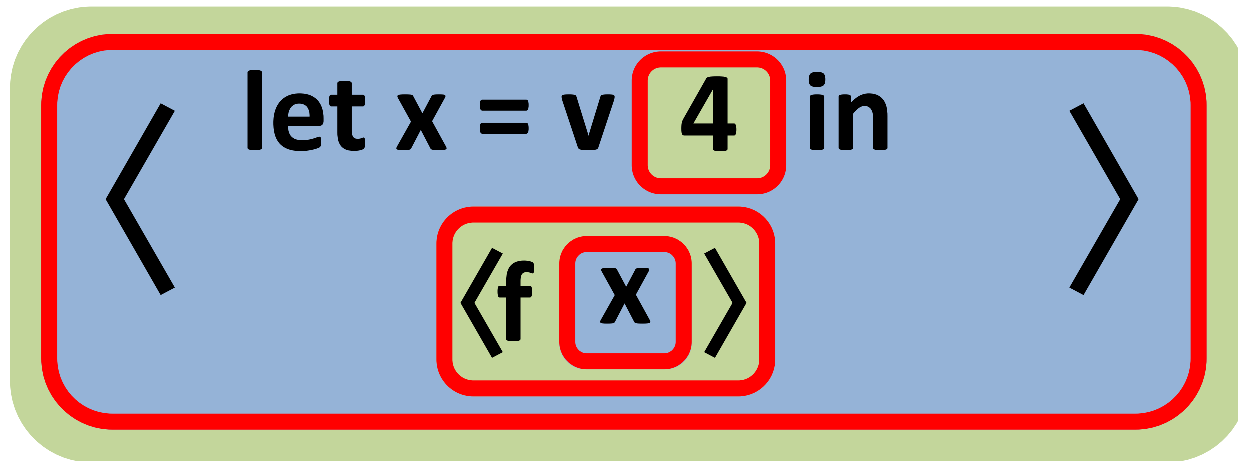
$\leftarrow$ v : int→int $\Longrightarrow$ Dyn

Our cast semantics

⟨f (v' 4)⟩

v : int→int ⇒ Dyn

$\mapsto$

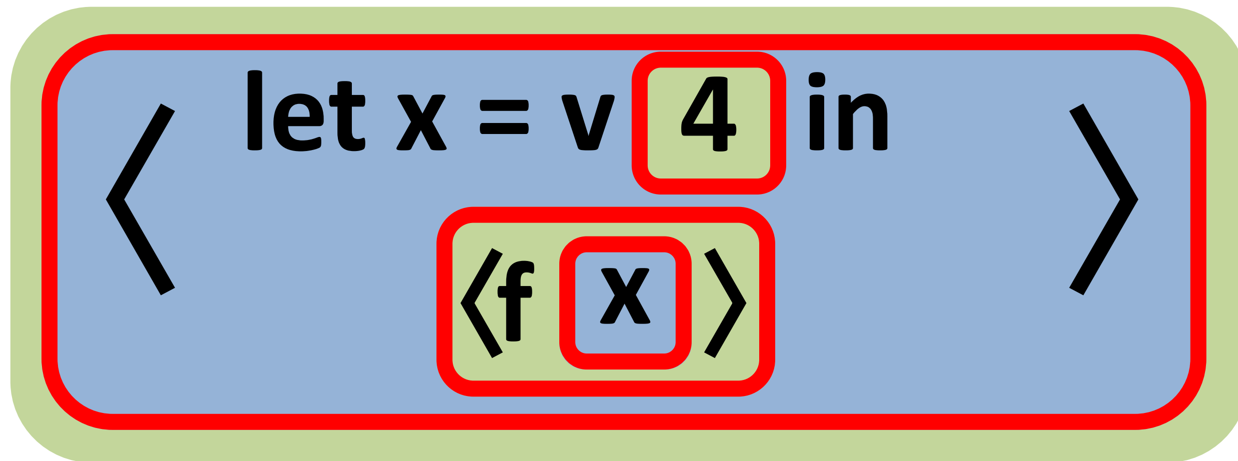


Our cast semantics

⟨f (v' 4)⟩

v : int→int ⟹ Dyn

↦



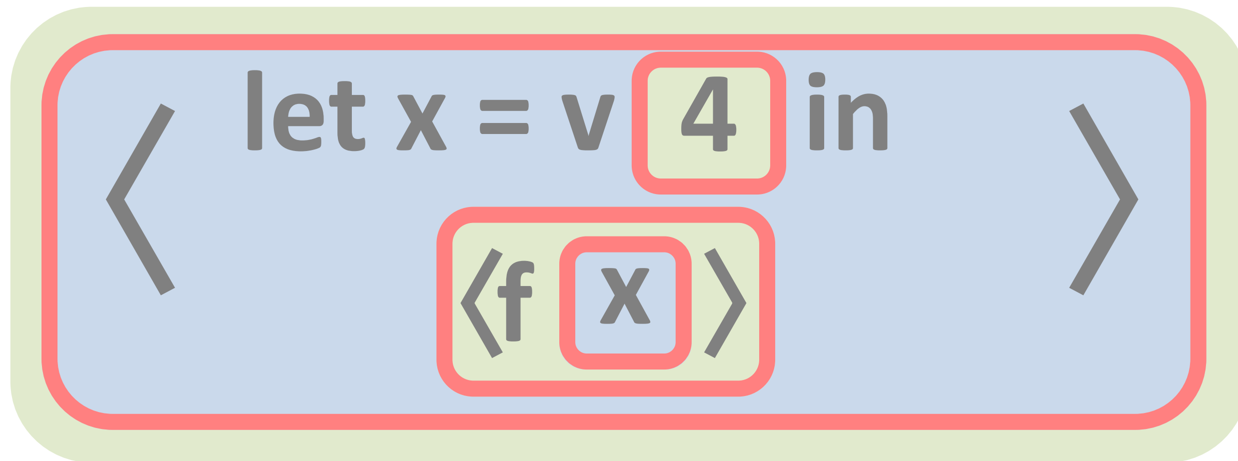
≈ ⟨f (v 4)⟩ (if all casts are ignored)

Our cast semantics

⟨f (v' 4)⟩

v : int→int ⟹ Dyn

$\mapsto$



≈ ⟨f (v 4)⟩ (if all casts are ignored)

Our cast semantics

The target function v is called with the argument to v’

⟨f (v’ …

v : int … ⟹ Dyn

$\mapsto$ ⟨ let x = v 4 in ⟨f x⟩ ⟩

≈ ⟨f (v 4)⟩ (if all casts are ignored)

Our cast semantics

⟨f (v'

The cast for the argument type

v : Int ⇒ Dyn

$\mapsto$ ⟨ let x = v 4 in ⟨f x⟩ ⟩

≈ ⟨f (v 4)⟩ (if all casts are ignored)

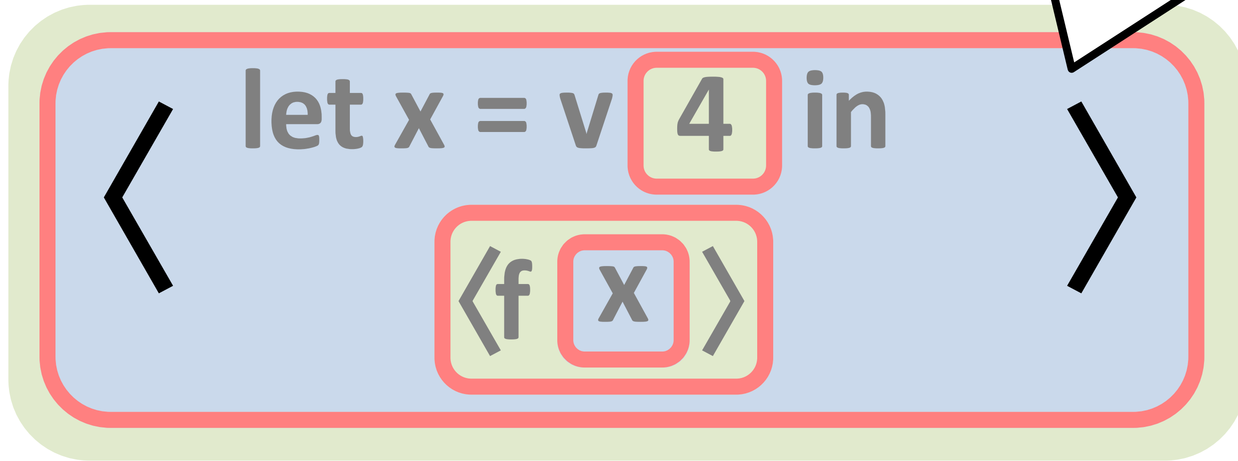
Our ca

guarantees captured continuations in typed function v are typed

⟨f (v' 4)⟩

v : int→int ⟹ Dyn

↦

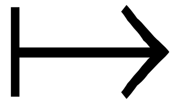


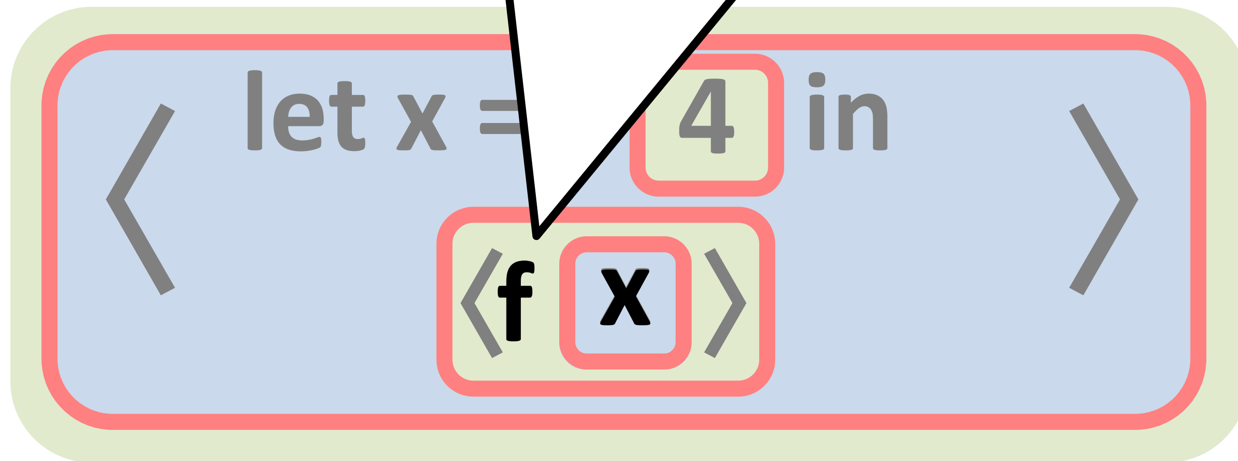
≈ ⟨f (v 4)⟩ (if all casts are ignored)

Our cast semantics

⟨f (v' 4)⟩

v : int→ Dyn

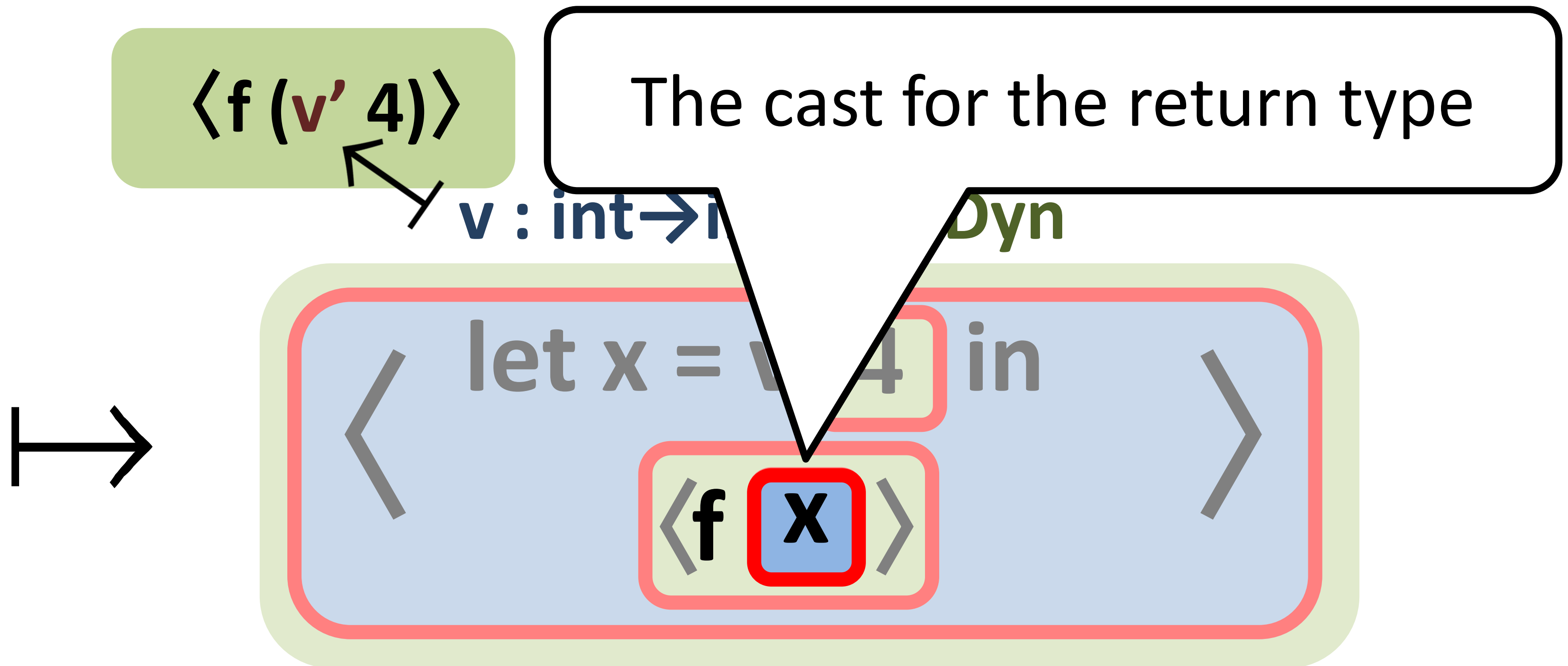




The result x of v is passed to f

≈ ⟨f (v 4)⟩ (if all casts are ignored)

Our cast semantics



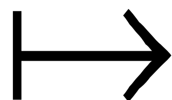
≈ ⟨f (v 4)⟩ (if all casts are ignored)

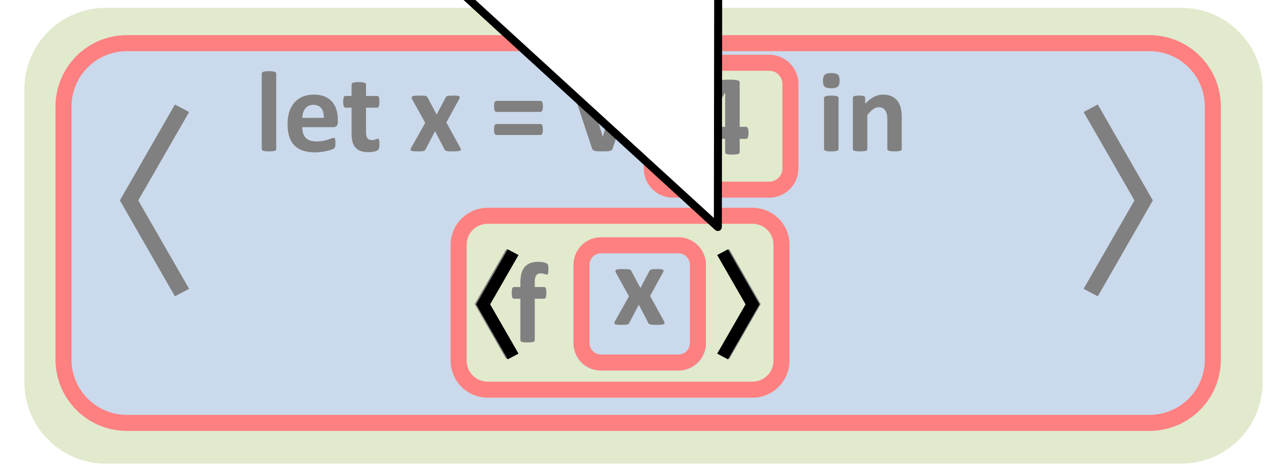
Our cast semantics

guarantees captured continuations in untyped function f are untyped

⟨f (v’ 4 ...

v : in... ⇒ Dyn

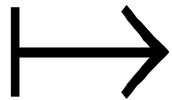


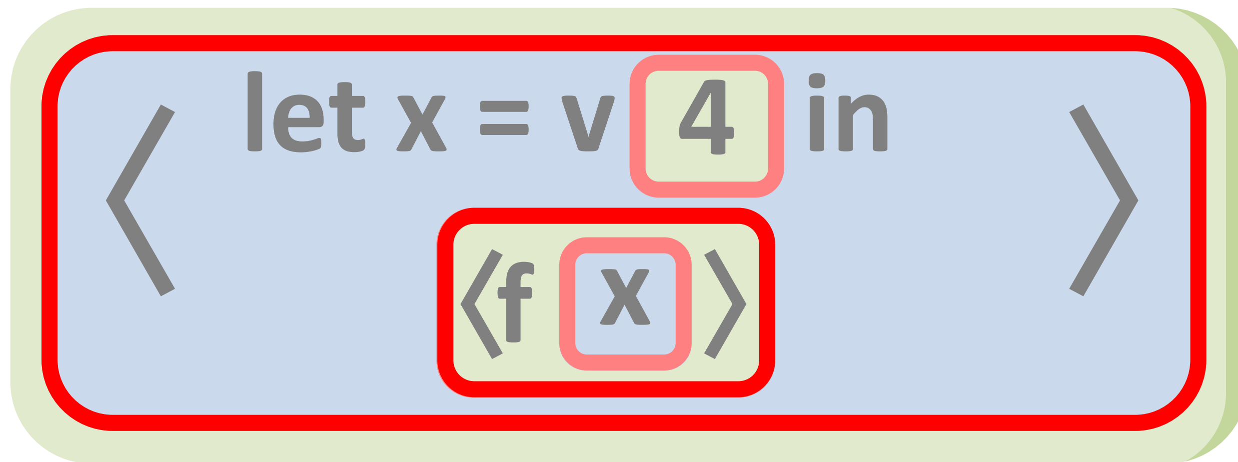


≈ ⟨**f (v 4)**⟩ (if all casts are ignored)

Our cast semantics

Additional casts are constructed by using type information given in Danvy&Filinski's type system





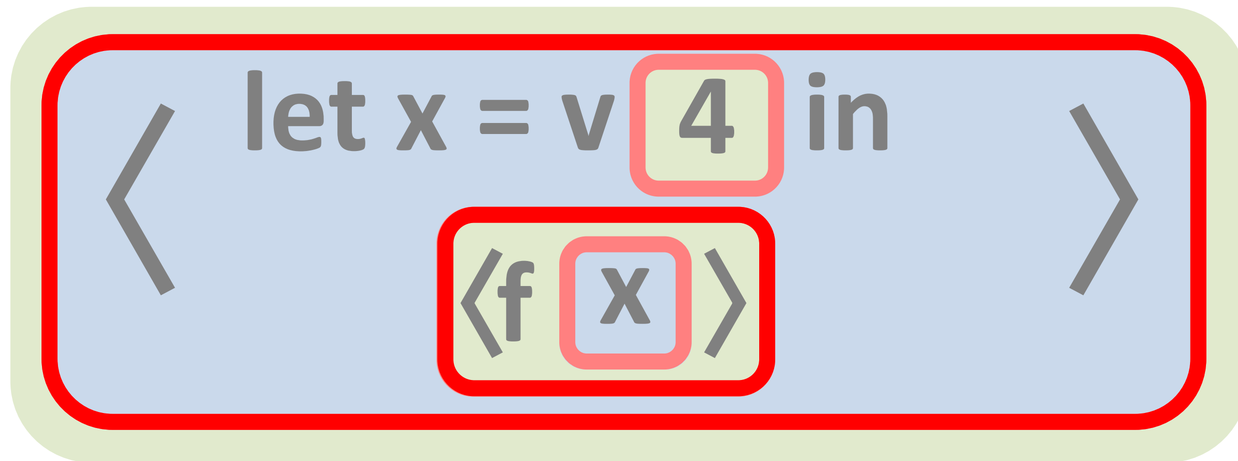
≈ **⟨f (v 4)⟩** (if all casts are ignored)

Our cast semantics

⟨f (v' 4)⟩

v : int→int ⟹ Dyn

↦



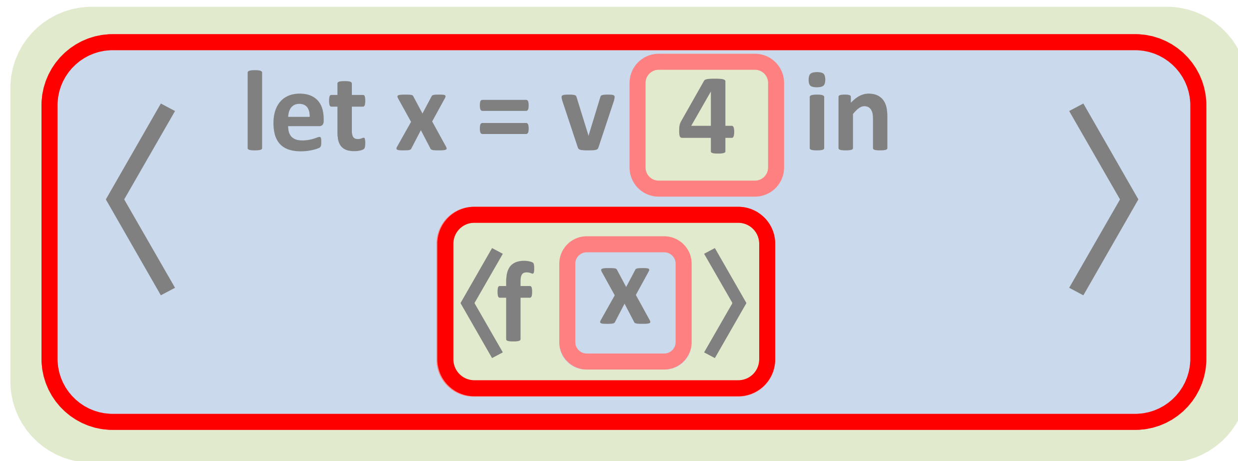
≈ ⟨f (v 4)⟩ (if all casts are ignored)

Our cast semantics

⟨f (v' 4)⟩

v : int/int→int/bool ⇒ Dyn

↦



≈ ⟨f (v 4)⟩ (if all casts are ignored)

requires that the closest reset's body at call point result in an integer

⟨f (v' 4)⟩

$v : int/\mathbf{int} \rightarrow int/\mathbf{bool} \Longrightarrow Dyn$

$\mapsto$ ⟨ let x = v 4 in ⟨f x⟩ ⟩

⟨…⟩ : Dyn ⟹ int

to check the outer reset's body returns an integer

guarantees that the closest reset
at call point returns a Boolean

$\langle f\ (v'\ 4)\rangle$

$v : int/int \rightarrow int/bool \Longrightarrow Dyn$

$\mapsto \langle\ \text{let } x = v\ 4 \text{ in } \langle f\ x\rangle\ \rangle$

$\approx \langle f\ (v$

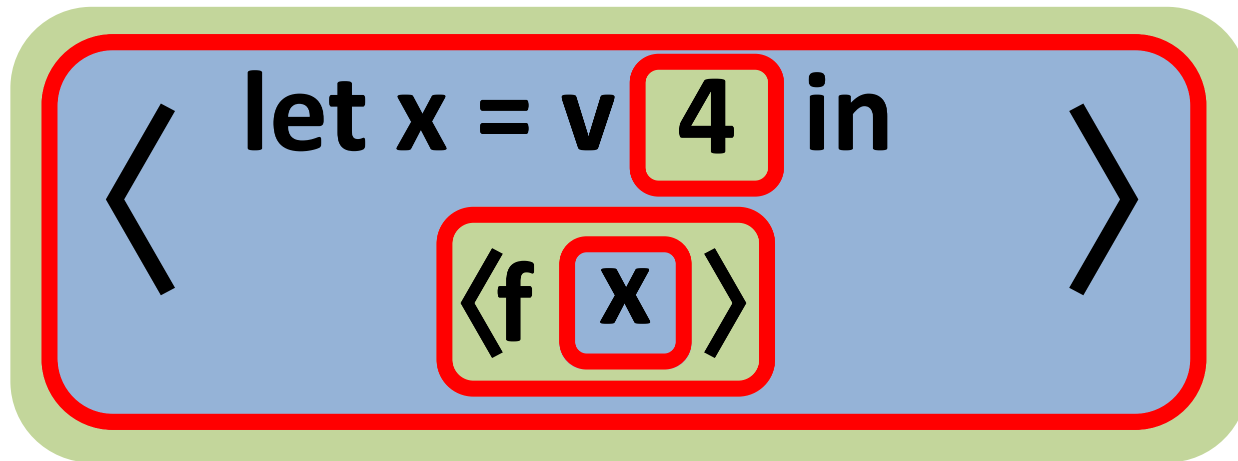
$\langle \ldots \rangle : bool \Longrightarrow Dyn$
to inject Boolean values to untyped code

Our cast semantics

⟨f (v′ 4)⟩

v : int/int→int/bool ⇒ Dyn

↦



≈ ⟨f (v 4)⟩ (if all casts are ignored)

Our calculus

Program syntax:

Types $S, T, \alpha, \beta ::= \text{int} \mid \ldots \mid \text{Dyn} \mid S/\alpha \rightarrow T/\beta$

Terms $s, t \quad ::= 1 \mid + \mid \ldots \mid \lambda x.s \mid s\ t \mid$

$s : S \Longrightarrow T \mid \mathbb{S}\, k.\, s \mid \langle s \rangle$

Type system:

$$\Gamma; \alpha \vdash s : T; \beta$$

Semantics:

shift/reset + our cast semantics

Type soundness

If $\langle s \rangle$ is a well typed, closed term, then:

- $\langle s \rangle$ diverges;
- $\langle s \rangle \longmapsto^* v$ for some v; or
- some cast in $\langle s \rangle$ fails

via Progress and Preservation

If something wrong happens,
it is detected as cast failure

In the paper…

- A formal system including run-time terms
- Support for “blame”

- Blame Theorem
  - Statically typed terms are never responsible for cast failure
- CPS transformation
- Sounenss of the CPS transformation
  - Preservation of Type
  - Preservation of Equality

Our CPS transformation $[[\cdot]]$

- $[[\cdot]]$ transforms terms/types in our calculus to ones in the simply typed blame calculus
- The definition is standard except for casts

$$[[s : S \Rightarrow T]] := \lambda k.\ [[s]]\ (\lambda x.\ x : [[S]] \Rightarrow [[T]])$$

$$[[\mathrm{Dyn}]] := \mathrm{Dyn}$$

$$[[S/\alpha \to T/\beta]] := [[S]] \to ([[T]] \to [[\alpha]]) \to [[\beta]]$$

Soundness of the CPS transformation

Preservation of Type

If $\Gamma;\alpha \vdash s : T; \beta$,
then $[[\Gamma]] \vdash [[s]] : ([[T]] \to [[\alpha]]) \to [[\beta]]$

Preservation of Equality

If $s \longmapsto t$, then $[[s]] \sim [[t]]$

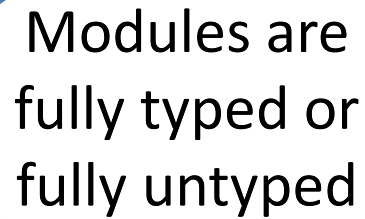
where $\sim$ is an equational system with usual call-by-value axioms and a few additional axiomas

Related work

Constraining delimited control with contracts
[Takikawa et al., ESOP'13]

studies "macro" gradual typing with:

- Control operators not based on CPS
  - powerful enough to express shift/reset
- Contract system
  - allows refined type information to be represented



Related work

Constraining delimited control with contracts
[Takikawa et al., ESOP'13]

studies "macro" gradual typing with:

- Control operators not based on CPS
  - powerful enough to express shift/reset
- Contract system
  - allows refined type information to be represented

Conclusion

An extension of blame calculus with shift/reset

- Cast semantic to monitor capturing and calling continuations
- Three properties investigated
  - Type soundness
  - Blame Theorem
  - Soundness of the CPS transformation