Imperative Polymorphism by Store-Based Types as Abstract Interpretations

Casper Bach Poulsen, Peter D. Mosses, Paolo Torrini

Department of Computer Science
Swansea University
Swansea, UK

{cscbp,p.d.mosses,p.torrini}@swansea.ac.uk

13 January 2015
PEPM, Mumbai, India

Type safety

"Well-typed programs can't go wrong"

—Robin Milner

Type safety

"Well-typed programs can't go wrong"

—Robin Milner

Million-rupee question

How do we construct a type system?

POPL'97

Types as Abstract Interpretations

(invited paper)

Patrick Cousot
LIENS, École Normale Supérieure
45, rue d'Ulm
75230 Paris cedex 05 (France)
cousot@dmi.ens.fr, http://www.ens.fr/~cousot

Abstract

Starting from a denotational semantics of the eager untyped lambda-calculus with explicit runtime errors, the standard collecting semantics is defined as specifying the strongest program properties. By a first abstraction, a new sound type collecting semantics is derived in compositional fixpoint form. Then by successive (semi-dual) Galois connection based abstractions, type systems and/or type inference algorithms are designed as abstract semantics or abstract interpreters approximating the type collecting semantics. This leads to a hierarchy of type systems, which is part of the lattice of abstract interpretations of the untyped lambda-calculus. This hierarchy includes two new à la Church/Curry polytype systems. Abstractions of this polytype semantics lead to classical Milner/Mycroft and Damas/-Milner polymorphic type schemes, Church/Curry monotypes and Hindley principal typing algorithm. This shows that types are abstract interpretations.

1 Introduction

The leading idea of abstract interpretation [6, 7, 9, 12] is that program semantics, proof and static analysis methods have common structures which can be exhibited by abstraction of the structure of run-time computations. This leads to an organization of the more or less approximate or refined semantics into a lattice of abstract interpretations. This uni-

checking algorithms can then be developed as a separate concern, and their correctness can be verified with respect to a given type system; this process guarantees that type checkers satisfy the language definition." [2]. Abstract interpretation allows viewing all these different aspects in the more unifying framework of semantic approximation. Formalization of program analysis and type systems within the same abstract interpretation framework should lead to a better understanding of the relationship between these seemingly different approaches to program correctness and optimization.

2 Syntax

The syntax of the untyped eager lambda calculus is:

$$\mathtt{x}, \mathtt{f}, \ldots \in \mathbb{X} \quad : \quad \text{program variables}$$

$$e \in \mathbb{E} \quad : \quad \text{program expressions}$$

$$e ::= \mathtt{x} \mid \boldsymbol{\lambda}\mathtt{x} \cdot e \mid e_1(e_2) \mid \boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e \mid \mathbf{1} \mid e_1 - e_2 \mid (e_1 \mathbin{?} e_2 : e_3)$$

$\boldsymbol{\lambda}\mathtt{x} \cdot e$ is the lambda abstraction and $e_1(e_2)$ the application. In $\boldsymbol{\mu}\mathtt{f} \cdot \boldsymbol{\lambda}\mathtt{x} \cdot e$, the function $\mathtt{f}$ with formal parameter $\mathtt{x}$ is defined recursively. $(e_1 \mathbin{?} e_2 : e_3)$ is the test for zero.

3 Denotational Semantics

Imperative Polymorphism by Store-Based Types as Abstract Interpretations

The vision:

big-step SOS for $\lambda_{\texttt{ref}}$

$\rightsquigarrow$ abstraction

store-based polymorphic type system

Imperative Polymorphism by Store-Based Types as Abstract Interpretations

In the paper:

big-step SOS for λ

$\rightsquigarrow$ abstraction

polymorphic type system

... and a store-based polymorphic type system

Imperative Polymorphism by Store-Based Types as Abstract Interpretations

In the paper:

big-step SOS for λ

⇝ abstraction

polymorphic type system

... and a store-based polymorphic type system

The problem with divergence in big-step SOS

$$x \in \mathit{Var} \qquad i \in \mathbb{Z} \qquad \rho \in \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}$$

$$\mathit{Expr} \ni e ::= \lambda x.e \mid e\, e \mid x \mid i \qquad \mathit{Val} \ni v ::= \langle x, e, \rho \rangle \mid i$$

$$\boxed{\rho \vdash e \Rightarrow v}$$

$$\frac{}{\rho \vdash \lambda x.e \Rightarrow \langle x, e, \rho \rangle} \qquad \frac{\rho(x) = v}{\rho \vdash x \Rightarrow v} \qquad \frac{}{\rho \vdash i \Rightarrow i}$$

$$\frac{\rho \vdash e_1 \Rightarrow \langle x, e, \rho' \rangle \quad \rho \vdash e_2 \Rightarrow v_2 \quad \rho'[x \mapsto v_2] \vdash e \Rightarrow v}{\rho \vdash e_1\ e_2 \Rightarrow v}$$

A solution for divergence in big-step SOS

$$x \in \mathit{Var} \qquad i \in \mathbb{Z} \qquad \rho \in \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}$$

$$\mathit{Expr} \ni e ::= \lambda x.e \mid e\,e \mid x \mid i \qquad \mathit{Val} \ni v ::= \langle x, e, \rho \rangle \mid i$$

$$\mathit{Div} \ni \delta ::= \downarrow \mid \uparrow$$

$$\frac{}{\rho \vdash e_{/\uparrow} \Rightarrow v_{/\uparrow}}\,(\text{Div}) \qquad \boxed{\rho \vdash e_{/\delta} \Rightarrow v_{/\delta'}}$$

$$\frac{}{\rho \vdash \lambda x.e_{/\downarrow} \Rightarrow \langle x, e, \rho \rangle_{/\downarrow}} \qquad \frac{\rho(x) = v}{\rho \vdash x_{/\downarrow} \Rightarrow v_{/\downarrow}} \qquad \frac{}{\rho \vdash i_{/\downarrow} \Rightarrow i_{/\downarrow}}$$

$$\frac{\rho \vdash e_{1/\downarrow} \Rightarrow \langle x, e, \rho' \rangle_{/\delta} \quad \rho \vdash e_{2/\delta} \Rightarrow v_{2/\delta'} \quad \rho'[x \mapsto v_2] \vdash e_{/\delta'} \Rightarrow v_{/\delta''}}{\rho \vdash e_1\; e_{2/\downarrow} \Rightarrow v_{/\delta''}}$$

A solution for divergence in big-step SOS

$$x \in \mathit{Var} \qquad i \in \mathbb{Z} \qquad \rho \in \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}$$

$$\mathit{Expr} \ni e ::= \lambda x.e \mid e\, e \mid x \mid i \qquad \mathit{Val} \ni v ::= \langle x, e, \rho \rangle \mid i$$

Coinductive interpretation

The coinductive interpretation of $\Rightarrow$ contains all computations expressible in the untyped λ-calculus.

$$\frac{}{\rho \vdash \lambda x.e_{/\downarrow} \Rightarrow \langle x, e, \rho \rangle_{/\downarrow}} \qquad \frac{\rho(x) = v}{\rho \vdash x_{/\downarrow} \Rightarrow v_{/\downarrow}} \qquad \frac{}{\rho \vdash i_{/\downarrow} \Rightarrow i_{/\downarrow}}$$

$$\frac{\rho \vdash e_{1/\downarrow} \Rightarrow \langle x, e, \rho' \rangle_{/\delta} \quad \rho \vdash e_{2/\delta} \Rightarrow v_{2/\delta'} \quad \rho'[x \mapsto v_2] \vdash e_{/\delta'} \Rightarrow v_{/\delta''}}{\rho \vdash e_1\, e_{2/\downarrow} \Rightarrow v_{/\delta''}}$$

Semantic function

$$\mathbf{S}[\![\bullet]\!] \in \mathbb{S} \triangleq \mathit{Env} \to \wp(\mathit{Val} \times \mathit{Div})$$

$$\mathbf{S}[\![e]\!] = \Lambda\rho.\{\langle v, \delta\rangle \mid \rho \vdash e_{/\downarrow} \Rightarrow v_{/\delta}\}$$

Semantic function

$$\mathbf{S}[\![\bullet]\!] \in \mathbb{S} \triangleq \mathit{Env} \to \wp(\mathit{Val} \times \mathit{Div})$$

$$\mathbf{S}[\![e]\!] = \Lambda\rho.\{\langle v, \delta\rangle \mid \rho \vdash e_{/\downarrow} \Rightarrow v_{/\delta}\}$$

For example:

$$\mathbf{S}[\![(\lambda x.\, x)]\!] = \Lambda\rho.\{\langle\langle x, x, \rho\rangle, \downarrow\rangle\}$$

Semantic function

$$\mathbf{S}[\![\bullet]\!] \in \mathbb{S} \triangleq \mathit{Env} \to \wp(\mathit{Val} \times \mathit{Div})$$

$$\mathbf{S}[\![e]\!] = \Lambda\rho.\{\langle v, \delta\rangle \mid \rho \vdash e_{/\downarrow} \Rightarrow v_{/\delta}\}$$

For example:

$$\mathbf{S}[\![(\lambda x.\, x)]\!] = \Lambda\rho.\{\langle\langle x, x, \rho\rangle, \downarrow\rangle\}$$
$$\mathbf{S}[\![(\lambda x.\, x\,x)(\lambda x.\, x\,x)]\!] = \Lambda\rho.\{\langle v, \delta\rangle \mid v \in \mathit{Val},\ \delta \in \mathit{Div}\}$$

Overview

✓big-step SOS for λ

$\rightsquigarrow$ abstraction

polymorphic type system

Type safety

"Well-typed programs can't go wrong"

Type safety

"Well-typed programs always go right"

Type safety

"Well-typed programs always go right"

$\Gamma \vdash e : \tau$

Type safety

"Well-typed programs always go right"

$$\Gamma \vdash e : \tau \implies \rho : \Gamma \implies$$
$$\exists v, \delta.\ \ \rho \vdash e_{/\downarrow} \Rightarrow v_{/\delta} \ \wedge\ \Gamma \vdash v : \tau$$

Abstraction

1. Define abstract domain
2. Type safety as abstraction function and Galois connection
3. Construct typing relation

1. Base-type domain definition

$$\boxed{\mathbb{S} \triangleq \mathit{Env} \to \wp(\mathit{Val} \times \mathit{Div})}$$

Concrete values:

$$\mathit{Val} \ni v ::= \langle x, e, \rho \rangle \mid i \qquad \rho \in \mathit{Env} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{Val}$$

1. Base-type domain definition

$$\boxed{\mathbb{S} \triangleq Env \rightarrow \wp(Val \times Div)}$$

$$\boxed{\mathbb{B} \triangleq \wp(BEnv \times BType)}$$

Concrete values:

$$Val \ni v ::= \langle x, e, \rho \rangle \mid i \qquad \rho \in Env \triangleq Var \xrightarrow{\text{fin}} Val$$

Base-types:

$$BType \ni b ::= \langle x, e, \rho^b \rangle \mid \texttt{int} \qquad \rho^b \in BEnv \triangleq Var \xrightarrow{\text{fin}} BType$$

1. Base-type domain definition

$\mathbb{S} \triangleq Env \to \wp(Val \times Div)$

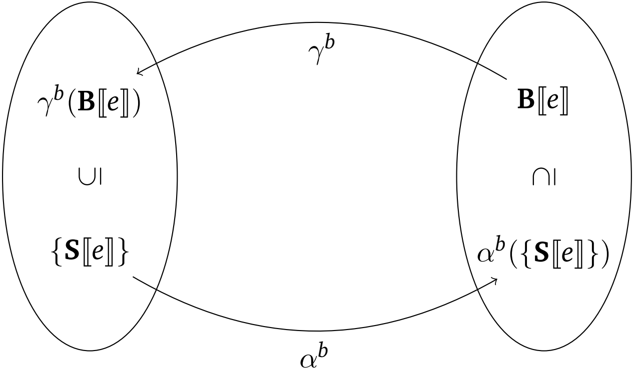
$\mathbb{B} \triangleq \wp(BEnv \times BType)$

Abstraction

$\alpha^b \in \wp(\mathbb{S}) \to \mathbb{B}$	property abstraction
$\alpha^b_v \in Val \to BType$	value abstraction
$\alpha^b_\rho \in Env \to BEnv$	environment abstraction
$\alpha^b_s \in \mathbb{S} \to \mathbb{B}$	semantic function abstraction

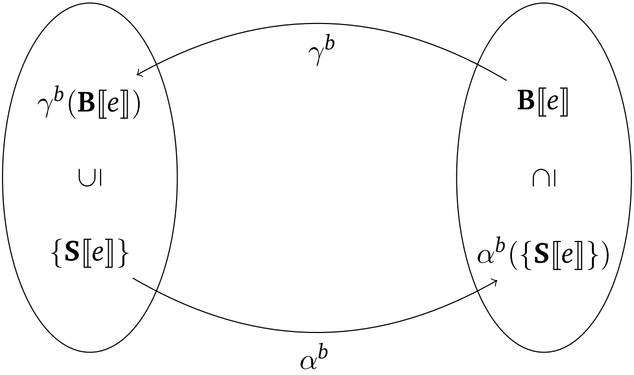
2. Base-type safety as Galois connection

$\mathbb{C} \triangleq \wp(\mathbb{S})$ $\mathbb{B} \triangleq \wp(BEnv \times BType)$



2. Base-type safety as Galois connection

$\mathbb{C} \triangleq \wp(\mathbb{S})$ $\mathbb{B} \triangleq \wp(BEnv \times BType)$



$$\langle \mathbb{C}, \subseteq \rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \mathbb{B}, \supseteq \rangle$$

See paper for details

3. Constructing typing function $\mathbf{B}[\![\bullet]\!]$

Define:

$$\mathbf{B}[\![e]\!] \triangleq \{\langle \rho^b, b \rangle \mid \rho^b \vdash e \Rightarrow^b b\}$$

3. Constructing typing function $\mathbf{B}[\![\bullet]\!]$

Define:

$$\mathbf{B}[\![e]\!] \triangleq \{\langle \rho^b, b \rangle \mid \rho^b \vdash e \Rightarrow^b b\}$$

Type safety as guiding principle:

$$\mathbf{B}[\![e]\!] \subseteq \alpha_s^b(\mathbf{S}[\![e]\!])$$

3. Constructing typing function $\mathbf{B}[\![\bullet]\!]$

Define:

$$\mathbf{B}[\![e]\!] \triangleq \{\langle \rho^b, b \rangle \mid \rho^b \vdash e \Rightarrow^b b\}$$

Type safety as guiding principle:

$$\mathbf{B}[\![e]\!] \subseteq \alpha_s^b(\mathbf{S}[\![e]\!])$$

Define α_s^b:

$$\alpha_s^b(S) \triangleq \{\langle \rho^b, b \rangle \mid \forall \rho.\ \rho^b = \alpha_\rho^b(\rho) \Longrightarrow \exists v.\ b = \alpha_v^b(v) \wedge \langle v, \downarrow \rangle \in S(\rho)\}$$

3. Constructing typing function $\mathbf{B}[\![\bullet]\!]$

Define:

Unfolding the guiding principle

$$\rho^b \vdash e \Rightarrow^b b \implies \rho^b = \alpha^b_\rho(\rho) \implies$$
$$\exists v.\ \rho \vdash e_{/\downarrow} \Rightarrow v_{/\downarrow} \ \wedge\ \alpha^b_v(v) = b$$

Deriving the structure of $\Rightarrow^b$:
rule induction on typing relation, using knowledge of $\Rightarrow$ and α^b_v.

$$\exists v.\ b = \alpha^b_v(v) \wedge \langle v, \downarrow\rangle \in S(\rho)\}$$

Overview

✓big-step SOS for λ

$\rightsquigarrow$ ✓abstraction

polymorphic type system

Polymorphic type system

$$\boxed{\mathbb{B} \triangleq \wp(\mathit{BEnv} \times \mathit{BType})} \qquad \boxed{\mathbb{P} \triangleq \wp(\mathit{PEnv} \times \mathit{MType})}$$

Base-types:

$$\mathit{BType} \ni b ::= \langle x, e, \rho^b \rangle \mid \mathtt{int} \qquad \rho^b \in \mathit{BEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{BType}$$

Monotypes with polytype environments:

$$\mathit{MType} \ni m ::= m \to m \mid \mathtt{int} \qquad \rho^p \in \mathit{PEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \wp(\mathit{MType})$$

Polymorphic type system

$$\boxed{\mathbb{B} \triangleq \wp(\mathit{BEnv} \times \mathit{BType})} \qquad \boxed{\mathbb{P} \triangleq \wp(\mathit{PEnv} \times \mathit{MType})}$$

Base-types:

$$\mathit{BType} \ni b ::= \langle x, e, \rho^b \rangle \mid \texttt{int} \qquad \rho^b \in \mathit{BEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \mathit{BType}$$

Monotypes with polytype environments:

$$\mathit{MType} \ni m ::= m \rightarrow m \mid \texttt{int} \qquad \rho^p \in \mathit{PEnv} \triangleq \mathit{Var} \xrightarrow{\text{fin}} \wp(\mathit{MType})$$

See paper for details

Overview

✓big-step SOS for λ

$\rightsquigarrow$ ✓abstraction

✓polymorphic type system

Overview

✓big-step SOS for λ

$\wr$

And now:

A preliminary store-based type system and some implications for imperative polymorphism.

$\downarrow$

✓polymorphic type system

Imperative let-polymorphism for $\lambda_{\texttt{ref}}$

Imperative polymorphism?

- *polymorphic type inference* in the presence of *imperative features* (e.g., *references*)

Imperative let-polymorphism for $\lambda_{\texttt{ref}}$

Imperative polymorphism?

- *polymorphic type inference* in the presence of *imperative features* (e.g., *references*)

$$\boxed{\rho \vdash e_{/\delta} \Rightarrow v_{/\delta'}}$$

$$Expr \ni e ::= \lambda x.e \mid e\,e \mid x \mid i$$

$$Val \ni v ::= \langle x, e, \rho \rangle \mid i$$

Imperative let-polymorphism for $\lambda_{\mathtt{ref}}$

Imperative polymorphism?

- *polymorphic type inference* in the presence of *imperative features* (e.g., *references*)

$$\boxed{\rho \vdash e_{/\delta \mid \sigma} \Rightarrow v_{/\delta' \mid \sigma'}}$$

$$\begin{aligned} \mathit{Expr} \ni e ::= {} & \lambda x.e \mid e\,e \mid x \mid i \mid \mathtt{ref}\ e \mid \mathtt{!}e \mid e\ \mathtt{:=}\ e \\ \mid {} & \mathtt{let}\ x\ \mathtt{=}\ e\ \mathtt{in}\ e \mid e; e \\ \mathit{Val} \ni v ::= {} & \langle x, e, \rho \rangle \mid i \mid l \end{aligned}$$

$$\sigma \in \mathit{Loc} \xrightarrow{\text{fin}} \mathit{Val}$$

An example – value restriction

An error-free program:

$$\texttt{let}\ \mathit{mkref}\ \texttt{=}\ (\lambda x.\, \texttt{ref}\ x)$$
$$\texttt{in}\ \mathit{mkref}\ 1\texttt{;}\ \ \mathit{mkref}\ \texttt{true}$$

- $\mathit{mkref}\ \texttt{=}\ (\lambda x.\texttt{ref}\ x) : \forall\alpha.\ \alpha \to (\alpha\ \texttt{ref})$

An example – value restriction

An error-free program:

$$\texttt{let } \mathit{mkref} \texttt{ = } (\lambda x.\,\texttt{ref } x)$$
$$\texttt{in } \mathit{mkref}\,1\texttt{; } \mathit{mkref}\,\texttt{true}$$

- $\mathit{mkref} \texttt{ = } (\lambda x.\texttt{ref } x) : \forall\alpha.\, \alpha \to (\alpha\ \texttt{ref})$
- $(\mathit{mkref}\,1) : \texttt{int ref}$

An example – value restriction

An error-free program:

$$\texttt{let}\ \mathit{mkref}\ \texttt{=}\ (\lambda x.\ \texttt{ref}\ x)$$
$$\texttt{in}\ \mathit{mkref}\ 1\texttt{;}\ \ \mathit{mkref}\ \texttt{true}$$

- $\mathit{mkref}\ \texttt{=}\ (\lambda x.\texttt{ref}\ x) : \forall\alpha.\ \alpha \to (\alpha\ \texttt{ref})$
- $(\mathit{mkref}\ 1) : \texttt{int ref}$
- $(\mathit{mkref}\ \texttt{true}) : \texttt{bool ref}$

An example – value restriction

An error-free program:

$$\texttt{let}\ \mathit{mkref} = (\lambda y.\, y)\, (\lambda x.\, \texttt{ref}\ x)$$
$$\texttt{in}\ \mathit{mkref}\, 1;\ \mathit{mkref}\, \texttt{true}$$

- $\mathit{mkref} = (\lambda y.\, y)\, (\lambda x.\, \texttt{ref}\ x) : \alpha \to (\alpha\ \texttt{ref})$

An example – value restriction

An error-free program:

$$\texttt{let}\ \mathit{mkref}\ \texttt{=}\ (\lambda y.\, y)\,(\lambda x.\, \texttt{ref}\ x)$$
$$\texttt{in}\ \mathit{mkref}\, 1\texttt{;}\ \ \mathit{mkref}\, \texttt{true}$$

- $\mathit{mkref}\ \texttt{=}\ (\lambda y.\, y)\,(\lambda x.\, \texttt{ref}\ x) : \texttt{int} \to (\texttt{int ref})$
- $(\mathit{mkref}\, 1) : \texttt{int ref}$

An example – value restriction

An error-free program:

$$\begin{array}{l}\texttt{let}\ \ \mathit{mkref}\ \texttt{=}\ (\lambda y.\, y)\,(\lambda x.\, \texttt{ref}\ \ x) \\ \texttt{in}\ \ \mathit{mkref}\,1\,;\ \ \mathit{mkref}\,\texttt{true}\end{array}$$

- $\mathit{mkref}\ \texttt{=}\ (\lambda y.\, y)\,(\lambda x.\, \texttt{ref}\ \ x) : \texttt{int} \to (\texttt{int}\,\texttt{ref})$
- $(\mathit{mkref}\,1) : \texttt{int}\,\texttt{ref}$
- $(\mathit{mkref}\,\texttt{true})$: inhibited by value restriction ☹

Let-polymorphism for λ

$$\boxed{\rho^p \vdash e \Rightarrow^p m}$$

$$MType \ni m ::= m \rightarrow m \mid \texttt{int}$$

$$\rho^p \in PEnv \triangleq Var \xrightarrow{\text{fin}} \wp(MType)$$

Store-based let-polymorphism for $\lambda_{\texttt{ref}}$

$$\boxed{\Gamma^P \vdash e_{/\varsigma} \Rightarrow^S M_{/\varsigma'}}$$

$$\begin{aligned} MType^{\varsigma} \ni M &::= \langle M, \varsigma \rangle \to \langle M, \varsigma \rangle \mid \texttt{int} \mid l \\ \Gamma^P &\in PEnv \triangleq Var \xrightarrow{\text{fin}} \wp(MType^{\varsigma}) \\ \varsigma &\in Loc \xrightarrow{\text{fin}} MType^{\varsigma} \end{aligned}$$

Store-based let-polymorphism for $\lambda_{\texttt{ref}}$

Store-based function types

$$\frac{\Gamma^P[x \mapsto \{M_2\}] \vdash e_{/\varsigma_0} \Rightarrow^S M_{1/\varsigma_\Delta}}{\Gamma^P \vdash \lambda x.e_{/\varsigma} \Rightarrow^S \langle M_2, \varsigma_0 \rangle \to \langle M_1, \varsigma_\Delta \rangle_{/\varsigma}}$$

$$\frac{\begin{array}{c}\Gamma^P \vdash e_{1/\varsigma} \Rightarrow^S \langle M_2, \varsigma_0 \rangle \to \langle M_1, \varsigma_\Delta \rangle_{/\varsigma'} \\ \Gamma^P \vdash e_{2/\varsigma'} \Rightarrow^S M_{2/\varsigma''} \qquad \varsigma_0 \preceq \varsigma''\end{array}}{\Gamma^P \vdash e_1\ e_{2/\varsigma} \Rightarrow^S M_{1/\varsigma'' \oslash \varsigma_\Delta}}$$

Store-based let-polymorphism for $\lambda_{\texttt{ref}}$

Store-based function types

$$\frac{\Gamma^P[x \mapsto \{M_2\}] \vdash e_{/\varsigma_0} \Rightarrow^S M_{1/\varsigma_\Delta}}{\Gamma^P \vdash \lambda x.e_{/\varsigma} \Rightarrow^S \langle M_2, \varsigma_0 \rangle \to \langle M_1, \varsigma_\Delta \rangle_{/\varsigma}}$$

$$\frac{\Gamma^P \vdash e_{1/\varsigma} \Rightarrow^S \langle M_2, \varsigma_0 \rangle \to \langle M_1, \varsigma_\Delta \rangle_{/\varsigma'} \qquad \Gamma^P \vdash e_{2/\varsigma'} \Rightarrow^S M_{2/\varsigma''} \qquad \varsigma_0 \preceq \varsigma''}{\Gamma^P \vdash e_1\ e_{2/\varsigma} \Rightarrow^S M_{1/\varsigma'' \oslash \varsigma_\Delta}}$$

An example – store-based types

$$\texttt{let}\ \mathit{mkref}\ \texttt{=}\ (\lambda y.y)(\lambda x.\texttt{ref}\ x)$$
$$\texttt{in}\ \mathit{mkref}\,1\texttt{;}\ \mathit{mkref}\,\texttt{true}$$

- $\mathit{mkref}\ \texttt{=}\ (\lambda y.\,y)\,(\lambda x.\,\texttt{ref}\ x)_{/.}$:

An example – store-based types

$$\texttt{let } \mathit{mkref} \texttt{ = } (\lambda y.y)(\lambda x.\texttt{ref } x)$$
$$\texttt{in } \mathit{mkref}\, 1\texttt{; } \mathit{mkref}\, \texttt{true}$$

- $\mathit{mkref} \texttt{ = } (\lambda y.\, y)\, (\lambda x.\, \texttt{ref } x)_{/.}$:
  - $(\lambda y.\, y)_{/.}$: $\langle \alpha, \cdot \rangle \rightarrow \langle \alpha, \cdot \rangle_{/.}$

An example – store-based types

$$\texttt{let}\ \mathit{mkref}\ \texttt{=}\ (\lambda y.y)(\lambda x.\texttt{ref}\ x)$$
$$\texttt{in}\ \mathit{mkref}\ 1\texttt{;}\ \mathit{mkref}\ \texttt{true}$$

- $\mathit{mkref}\ \texttt{=}\ (\lambda y.\, y)\,(\lambda x.\, \texttt{ref}\ x)_{/.}$:
  - $(\lambda y.\, y)_{/.} : \langle \alpha, \cdot \rangle \to \langle \alpha, \cdot \rangle_{/.}$
  - $(\lambda x.\, \texttt{ref}\ x)_{/.}$:

An example – store-based types

$\texttt{let}\ \mathit{mkref} = (\lambda y.y)(\lambda x.\texttt{ref}\ x)$
$\texttt{in}\ \mathit{mkref}\ 1\,;\ \mathit{mkref}\ \texttt{true}$

- $\mathit{mkref} = (\lambda y.\, y)\, (\lambda x.\, \texttt{ref}\ x)_{/.} : (\forall \ell.\, \forall \beta.\, \langle \beta, \cdot \rangle \to \langle \ell, (\ell \mapsto \beta) \rangle)_{/.}$
  - $(\lambda y.\, y)_{/.} : \langle \langle \beta, \cdot \rangle \to \langle \ell, (\ell \mapsto \beta) \rangle, \cdot \rangle \to \langle \langle \beta, \cdot \rangle \to \langle \ell, (\ell \mapsto \beta) \rangle, \cdot \rangle_{/.}$
  - $(\lambda x.\, \texttt{ref}\ x)_{/.} : \langle \beta, \cdot \rangle \to \langle \ell, (\ell \mapsto \beta) \rangle_{/.}$

An example – store-based types

$$\texttt{let } \mathit{mkref} \texttt{ = } (\lambda y.y)(\lambda x.\texttt{ref } x)$$
$$\texttt{in } \mathit{mkref}\, 1\texttt{; } \mathit{mkref}\, \texttt{true}$$

- $\mathit{mkref} \texttt{ = } (\lambda y.\, y)\,(\lambda x.\, \texttt{ref } x)_{/.} : (\forall \ell.\, \forall \beta.\, \langle \beta, \cdot \rangle \to \langle \ell, (\ell \mapsto \beta) \rangle)_{/.}$
- $(\mathit{mkref}\, 1)_{/.}$:

An example – store-based types

$$\begin{array}{l}\texttt{let}\ \ mkref\ \texttt{=}\ (\lambda y.y)(\lambda x.\texttt{ref}\ \ x)\\ \texttt{in}\ \ mkref\ 1\,\texttt{;}\ \ mkref\ \texttt{true}\end{array}$$

- $mkref\ \texttt{=}\ (\lambda y.\, y)\,(\lambda x.\ \texttt{ref}\ \ x)_{/\cdot} : (\forall \ell.\, \forall \beta.\, \langle \beta, \cdot \rangle \rightarrow \langle \ell, (\ell \mapsto \beta) \rangle)_{/\cdot}$
- $(mkref\ 1)_{/\cdot} : l_{1/(l_1 \mapsto \texttt{int})}$

An example – store-based types

$$\texttt{let}\ \mathit{mkref}\ \texttt{=}\ (\lambda y.y)(\lambda x.\texttt{ref}\ x)$$
$$\texttt{in}\ \mathit{mkref}\ 1\texttt{;}\ \mathit{mkref}\ \texttt{true}$$

- $\mathit{mkref}\ \texttt{=}\ (\lambda y.\, y)\,(\lambda x.\, \texttt{ref}\ x)_{/\cdot} : (\forall \ell.\, \forall \beta.\, \langle \beta, \cdot \rangle \to \langle \ell, (\ell \mapsto \beta) \rangle)_{/\cdot}$
- $(\mathit{mkref}\ 1)_{/\cdot} : l_{1/(l_1 \mapsto \texttt{int})}$
- $(\mathit{mkref}\ \texttt{true})_{/(l_1 \mapsto \texttt{int})}$:

An example – store-based types

$$\begin{array}{l}\texttt{let}\ \ mkref\ \texttt{=}\ (\lambda y.y)(\lambda x.\texttt{ref}\ \ x)\\ \texttt{in}\ \ mkref\ 1\texttt{;}\ \ mkref\ \texttt{true}\end{array}$$

- $mkref\ \texttt{=}\ (\lambda y.\ y)\,(\lambda x.\ \texttt{ref}\ \ x)_{/\cdot} : (\forall \ell.\,\forall \beta.\,\langle \beta, \cdot\rangle \rightarrow \langle \ell, (\ell \mapsto \beta)\rangle)_{/\cdot}$
- $(mkref\ 1)_{/\cdot} : l_{1/(l_1 \mapsto \texttt{int})}$
- $(mkref\ \texttt{true})_{/(l_1 \mapsto \texttt{int})} : l_{2/(l_1 \mapsto \texttt{int};l_2 \mapsto \texttt{bool})}$

An example – store-based types

$$\texttt{let}\ \mathit{mkref}\ \texttt{=}\ (\lambda y.y)(\lambda x.\texttt{ref}\ x)$$
$$\texttt{in}\ \mathit{mkref}\ 1\texttt{;}\ \mathit{mkref}\ \texttt{true}$$

- $\mathit{mkref}\ \texttt{=}\ (\lambda y.\, y)\,(\lambda x.\, \texttt{ref}\ x)_{/\cdot} : (\forall \ell.\, \forall \beta.\, \langle \beta, \cdot \rangle \to \langle \ell, (\ell \mapsto \beta) \rangle)_{/\cdot}$
- $(\mathit{mkref}\ 1)_{/\cdot} : l_{1/(l_1 \mapsto \texttt{int})}$
- $(\mathit{mkref}\ \texttt{true})_{/(l_1 \mapsto \texttt{int})} : l_{2/(l_1 \mapsto \texttt{int}; l_2 \mapsto \texttt{bool})}$

Passes type checking ☺

Conclusions and future work

Divergence in big-step SOS:

- concise semantics amenable to abstract interpretation

Conclusions and future work

Divergence in big-step SOS:

- concise semantics amenable to abstract interpretation

Abstraction as guiding principle:

- viable for constructing safe type systems

Conclusions and future work

Divergence in big-step SOS:

- concise semantics amenable to abstract interpretation

Abstraction as guiding principle:

- viable for constructing safe type systems
- ongoing work: Coq encoding and proofs

Conclusions and future work

Divergence in big-step SOS:

- concise semantics amenable to abstract interpretation

Abstraction as guiding principle:

- viable for constructing safe type systems
- ongoing work: Coq encoding and proofs

Store-based types:

- derivation remains to be rigorously checked