

CPS Transformation with Affine Types for Implicit Polymorphism

Taro Sekiyama

National Institute of Informatics

Will be presented at ICFP 2021

CPS transformation

- Exposing control flow via explicit access to continuations

$$\llbracket \lambda f. 42 + (f\ 0) \rrbracket = \lambda f. \lambda k. f\ 0\ (\lambda x. k\ (42 + x))$$

- Applications

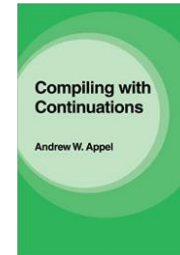
Semantics of control operators

$$\llbracket \mathcal{C}\ \lambda x. e \rrbracket = \lambda k. \llbracket \lambda x. e \rrbracket (\lambda y. \lambda k'. k\ y)\ (\lambda z. z)$$

$$\llbracket \text{shift}\ \lambda x. e \rrbracket = \lambda k. \llbracket \lambda x. e \rrbracket (\lambda y. \lambda k'. k'\ (k\ y))\ (\lambda z. z)$$

$$\llbracket \text{reset}\ e \rrbracket = \lambda k. k\ (\llbracket e \rrbracket (\lambda x. x))$$

Compiler IRs



The Essence of Compiling with Continuations

Cormac Flanagan* Amr Sabry* Bruce F. Duba Matthias Felleisen*

Compiling with Continuations, Continued

Andrew Kennedy
Microsoft Research Cambridge
akenn@microsoft.com

CPS transformation with **type preservation**

□ Exposing control flow via explicit access to continuations

$$\llbracket \lambda f. 42 + (f\ 0) : \tau \rrbracket = \lambda f. \lambda k. f\ 0\ (\lambda x. k\ (42 + x)) : \llbracket \tau \rrbracket$$

□ Applications

Semantics of control operators

$$\llbracket \mathcal{C}\ \lambda x. e \rrbracket = \lambda k. \llbracket \lambda x. e \rrbracket (\lambda y. \lambda k'. k\ y) (\lambda z. z)$$

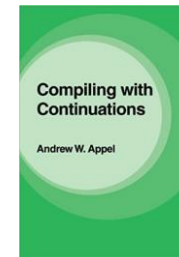
$$\llbracket \text{shift}\ \lambda x. e \rrbracket = \lambda k. \llbracket \lambda x. e \rrbracket (\lambda y. \lambda k'. k' (k\ y)) (\lambda z. z)$$

$$\llbracket \text{reset}\ e \rrbracket = \lambda k. k\ (\llbracket e \rrbracket (\lambda x. x))$$

Fine-grained typing of control operators

$$\frac{\Gamma; \alpha \vdash e : \tau; \beta \quad \Gamma, x: \tau \rightarrow \perp \vdash e : \perp}{\Gamma \vdash \mathcal{C}\ \lambda x. e : \tau}$$

Compiler IRs



Typing IRs



CPS transformation for **polymorphism**

Explicit polymorphism ($\Lambda\alpha. e$ is a value)

Explicit Polymorphism and CPS Conversion

Robert Harper Mark Lillibridge

October, 1992

CMU-CS-92-210

Abstract

We study the typing properties of CPS conversion for an extension of F_ω with control operators. Two classes of evaluation strategies are considered, each with call-by-name and call-by-value variants. Under the “standard” strategies, constructor abstractions are values, and constructor applications can lead to non-trivial control effects. In contrast, the “ML-like” strategies evaluate beneath constructor abstractions, reflecting the usual interpretation of programs in languages based on implicit polymorphism. Three continuation passing style sub-languages are considered.

CPS transformation for **polymorph**

$$\frac{e_1 \mapsto e_2}{\Lambda\alpha. e_1 \mapsto \Lambda\alpha. e_2}$$

Implicit polymorphism (the body of $\Lambda\alpha. e$ can be evaluated)

Polymorphic Type Assignment and CPS Conversion*

ROBERT HARPER[†]

(rwh@cs.cmu.edu)

MARK LILLIBRIDGE[‡]

(mdl@cs.cmu.edu)

*School of Computer Science
Carnegie Mellon University*

5000 Forbes Avenue

“

We obtain CPS transforms for the call-by-value interpretation,
provided that the polymorphic let is restricted to values.

”

transform. This typing property may be extended to Scheme-like continuation-passing primitives, from which the soundness of these extensions follows. We study the extension of these results to the Damas-Milner polymorphic type assignment system under both the call-by-value and call-by-name interpretations. We obtain CPS transforms for the call-by-value interpretation, provided that the polymorphic let is restricted to values, and for the call-by-name interpretation with no restrictions. We prove that there is no call-by-value CPS transform for the full Damas-Milner language that validates the Meyer-Wand typing property and is equivalent to the standard call-by-value transform up to operational equivalence.

Goal of this work

Long-term goal

Obtaining type-preserving CPS transformation for
implicit polymorphism without value restriction

Short-term goal

Obtaining type-preserving CPS transformation for
the implicit version of System F

Note: support for effects with other restriction (e.g. relaxed value restriction) is left open

Review: CPS transformation

$$\llbracket \lambda x. e \rrbracket = \lambda k. k \ \lambda x. \llbracket e \rrbracket$$

$$\llbracket x \rrbracket = \lambda k. k \ x$$

$$\llbracket e_1 \ e_2 \rrbracket = \lambda k. \llbracket e_1 \rrbracket \ (\lambda x. \llbracket e_2 \rrbracket \ (\lambda y. x \ y \ k))$$

Factorizing CPS transformation [Danvy'92]

1. Naming intermediate results of computation

$$e_1 e_2 \Rightarrow \text{let } x = e_1 \text{ in } x e_2$$

2. Sequencing computation by lifting redexes

$$x (\text{let } y = e_1 \text{ in } e_2) \Rightarrow \text{let } y = e_1 \text{ in } x e_2$$

3. Making continuations explicit

Factorizing CPS transformation [Danvy'92]

1. Naming intermediate results of computation

$$e_1 e_2 \Rightarrow \text{let } x = e_1 \text{ in } x e_2$$



Sequencing computation by lifting redexes

$$x (\text{let } y = e_1 \text{ in } e_2) \Rightarrow \text{let } y = e_1 \text{ in } x e_2$$

3. Making continuations explicit

Redex lifting as source-level reduction [Sabry+'92]

$$E[(\lambda x: \tau. e_1) e_2] \mapsto (\lambda x: \tau. E[e_1]) e_2$$

(if $x \notin fv(E) \wedge E \neq \square$)

This rule conflicts with implicit polymorphism
due to the existence of evaluation contexts like $\Lambda\alpha. \square$

Redex lifting as source-level reduction [Sabry+'92]

$$E[(\lambda x: \tau. e_1) e_2] \mapsto (\lambda x: \tau. E[e_1]) e_2$$

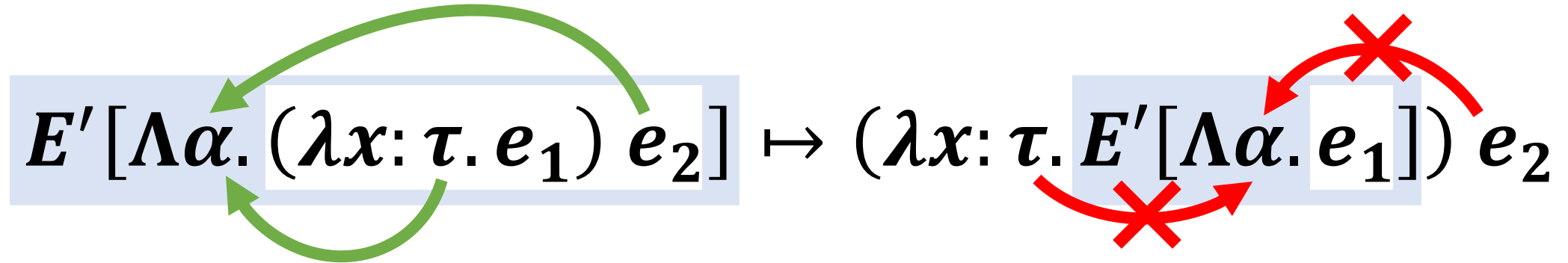
(if $x \notin fv(E) \wedge E \neq \square$)

Replacing by $E'[\Lambda\alpha. \square]$

The diagram illustrates the replacement of the evaluation context E in the redex lifting rule. A blue box at the bottom contains the text "Replacing by $E'[\Lambda\alpha. \square]$ ". Two blue arrows originate from this box: one points to the E in the left-hand side of the rule, and the other points to the E in the right-hand side of the rule. The E and $E[e_1]$ terms in the rule are highlighted with light blue rectangular backgrounds.

This rule conflicts with implicit polymorphism due to the existence of evaluation contexts like $\Lambda\alpha. \square$

Redex lifting in implicit polymorphism



Problem: the reduction “intrudes” the scope of α , invalidating the references to α in τ and e_2

$\Lambda\alpha$ must be *lowered*
to generalize α in e_1

VERSUS

$\Lambda\alpha$ must be *lifted*
to bind α in τ and e_2

Key idea of our solution

Decomposing $\Lambda\alpha$ into more atomic constructors

Restrictions $\nu\alpha.e$

only bind α (not generalize)

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \nu\alpha.e : \tau} \quad a \notin ftv(\tau)$$

Open type abstractions $\Lambda^\circ\langle\alpha, e\rangle$

only generalize α (not bind)

$$\frac{\Gamma \vdash e : \tau \quad \alpha \in \Gamma}{\Gamma \vdash \Lambda^\circ\langle\alpha, e\rangle : \forall\alpha.\tau}$$

Relationship to type abstraction: $\Lambda\alpha.e \equiv \nu\alpha.\Lambda^\circ\langle\alpha, e\rangle$

Remark: These typing rules don't imply type safety and need refinement as shown later

Examples

Restrictions $\nu\alpha. e$

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \nu\alpha. e : \tau}$$

**Open type
abstractions** $\Lambda^\circ \langle \alpha, e \rangle$

$$\frac{\Gamma \vdash e : \tau \quad \alpha \in \Gamma}{\Gamma \vdash \Lambda^\circ \langle \alpha, e \rangle : \forall \alpha. \tau}$$

$$\vdash \nu\alpha. \Lambda^\circ \langle \alpha, \lambda x: \alpha. x \rangle : \forall \alpha. \alpha \rightarrow \alpha$$

$$\not\vdash \Lambda^\circ \langle \alpha, \lambda x: \alpha. x \rangle : \forall \alpha. \alpha \rightarrow \alpha$$

$$\alpha, x: \alpha \rightarrow \alpha \vdash \Lambda^\circ \langle \alpha, x \rangle : \forall \alpha. \alpha \rightarrow \alpha$$


Redex lifting with decomposed type abstraction

$E'[\Lambda\alpha. (\lambda x: \tau. e_1) e_2] \quad \mapsto$


$$\Lambda\alpha. \square \equiv \textcolor{brown}{v}\alpha. \textcolor{green}{\Lambda}^\circ \langle \alpha, \square \rangle$$

Redex lifting with decomposed type abstraction

$E'[\textcolor{brown}{v}\alpha. \textcolor{green}{\Lambda}^\circ\langle\alpha, (\lambda x:\tau. e_1) e_2\rangle] \mapsto$



$\Lambda\alpha. \Box \equiv \textcolor{brown}{v}\alpha. \textcolor{green}{\Lambda}^\circ\langle\alpha, \Box\rangle$

Redex lifting with decomposed type abstraction

$$E'[\mathbf{v}\alpha. \Lambda^\circ\langle\alpha, (\lambda x:\tau. e_1) e_2\rangle] \mapsto \mathbf{v}\alpha. E'[\Lambda^\circ\langle\alpha, (\lambda x:\tau. e_1) e_2\rangle]$$

($\mathbf{v}\alpha$ is lifted)

Redex lifting with decomposed type abstraction

$$\begin{aligned} E'[\mathbf{v}\alpha. \Lambda^\circ\langle\alpha, (\lambda x:\tau. e_1) e_2\rangle] &\mapsto \mathbf{v}\alpha. E'[\Lambda^\circ\langle\alpha, (\lambda x:\tau. e_1) e_2\rangle] \\ &\quad (\mathbf{v}\alpha \text{ is lifted}) \\ &\mapsto \mathbf{v}\alpha. (\lambda x:\tau. E'[\Lambda^\circ\langle\alpha, e_1\rangle]) e_2 \\ &\quad (\text{the redex is lifted}) \end{aligned}$$

Redex lifting with decomposed type abstraction

$$E'[\mathbf{v}\alpha. \Lambda^\circ\langle\alpha, (\lambda x:\tau. e_1) e_2\rangle] \mapsto \mathbf{v}\alpha. E'[\Lambda^\circ\langle\alpha, (\lambda x:\tau. e_1) e_2\rangle]$$

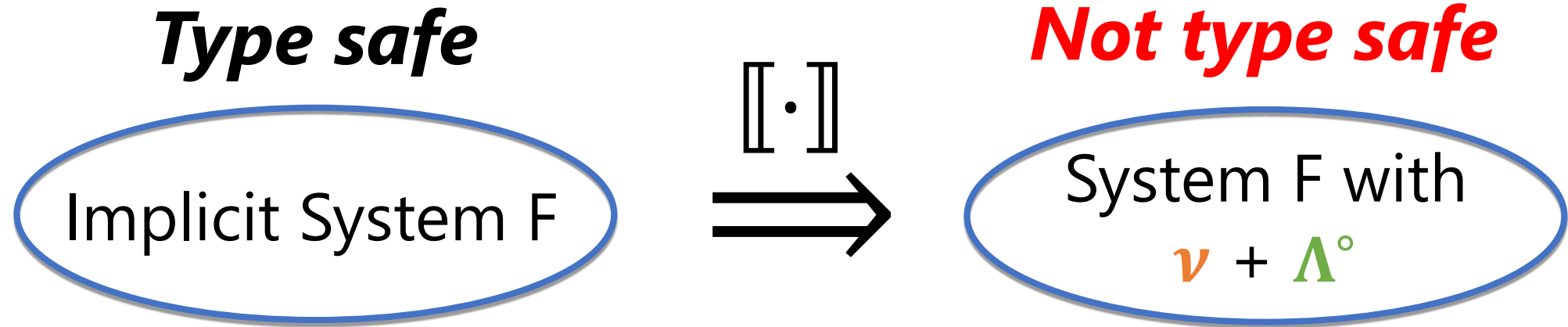
($\mathbf{v}\alpha$ is lifted)

$$\mapsto \mathbf{v}\alpha. (\lambda x:\tau. E'[\Lambda^\circ\langle\alpha, e_1\rangle]) e_2)$$

(the redex is lifted)

Requirements for typing	Generalize α in e_1	Bind α in τ and e_2
How solved?	By lowering $\Lambda^\circ\langle\alpha, \square\rangle$	By lifting $\mathbf{v}\alpha$

What we have got



Unsafety by re-generalization

Let $M \equiv \textcolor{brown}{\nu}\alpha. \textcolor{green}{\Lambda}^\circ\langle\alpha, \lambda x: \alpha. \textcolor{green}{\Lambda}^\circ\langle\alpha, \lambda y: \alpha. x\rangle\rangle$

$\vdash M : \forall\alpha. \alpha \rightarrow \forall\alpha. \alpha \rightarrow \alpha$

So $\vdash (M \text{ bool true}) \text{ int } 0 : \text{int}$

But $(M \text{ bool true}) \text{ int } 0 \mapsto^* \text{true}$

Restrictions $\textcolor{brown}{\nu}\alpha. e$

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \textcolor{brown}{\nu}\alpha. e : \tau}$$

**Open type
abstractions** $\textcolor{green}{\Lambda}^\circ\langle\alpha, e\rangle$

$$\frac{\Gamma \vdash e : \tau \quad \alpha \in \Gamma}{\Gamma \vdash \textcolor{green}{\Lambda}^\circ\langle\alpha. e\rangle : \forall\alpha. \tau}$$

Unsafety by re-generalization

Let $M \equiv \nu\alpha. \Lambda^\circ\langle\alpha, \lambda x: \alpha. \Lambda^\circ\langle\alpha, \lambda y: \alpha. x\rangle\rangle$

Cause: The same type variable may be generalized multiple times

Solution: Using linear / affine typing

Restrictions $\nu\alpha. e$

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \nu\alpha. e : \tau}$$

**Open type
abstractions** $\Lambda^\circ\langle\alpha, e\rangle$

$$\frac{\Gamma \vdash e : \tau \quad \alpha \in \Gamma}{\Gamma \vdash \Lambda^\circ\langle\alpha. e \rangle : \forall\alpha. \tau}$$

Type-safe CPS target language Λ^{open}

Polymorphic affine lambda calculus
+ restrictions + open type abstractions

Syntax

Types $A, B ::= \alpha \mid \forall \alpha. A \mid A \multimap B \mid !A \mid \text{int} \mid \dots$

Terms $M ::= x \mid c \mid \lambda x. M \mid M_1 M_2 \mid \Lambda \alpha. M \mid M A \mid$
 $!M \mid \text{let } !x = M_1 \text{ in } M_2 \mid \nu \alpha. M \mid \Lambda^\circ \langle \alpha, M \rangle$

Semantics (excerpt)

$$\frac{M_1 \mapsto M_2}{\Lambda^\circ \langle \alpha, M_1 \rangle \mapsto \Lambda^\circ \langle \alpha, M_2 \rangle}$$

$$\Lambda^\circ \langle \alpha, V \rangle \mapsto \Lambda \alpha. V$$

Type system

Uses $\pi ::= 0 \mid 1 \mid \omega$

Typing contexts $\Gamma ::= \emptyset \mid \Gamma, x :^\pi A \mid \Gamma, \alpha^\pi$

π must be 1 or 0, expressing if α can be generalized or not

$$\frac{\Gamma, \alpha^1 \vdash M : A \quad \alpha \notin \text{ftv}(A)}{\Gamma \vdash \mathbf{v}\alpha.M : A}$$

$$\frac{\Gamma_1, \alpha^0, \Gamma_2 \vdash M : !A}{\Gamma_1, \alpha^1, \Gamma_2 \vdash \Lambda^\circ \langle \alpha.M \rangle : !\forall \alpha. A}$$

CPS transformation $\llbracket \cdot \rrbracket$, a bit formally

Mapping from typing derivations in implicit System F to Λ^{open}

$$\llbracket \frac{\Theta, \alpha \vdash e : \tau}{\Theta \vdash e : \forall \alpha. \tau} \rrbracket = \lambda k : \llbracket \forall \alpha. \tau \rrbracket. \textcolor{brown}{v}\alpha. \llbracket \Theta, \alpha \vdash e : \tau \rrbracket (\lambda x : \llbracket \tau \rrbracket. k \textcolor{green}{\Lambda}^\circ \langle \alpha, x \rangle)$$

$$\begin{array}{c} \frac{\vdash \Theta \quad x : \tau \in \Theta}{\llbracket \Theta \vdash x : \tau \rrbracket \Rightarrow \Lambda \alpha. \lambda k. k ! x} \text{C_VAR} \quad \frac{\vdash \Theta}{\llbracket \Theta \vdash ty \rightarrow (c) \rrbracket \Rightarrow \Lambda \alpha. \lambda k. k \llbracket c : ty \rightarrow (c) \rrbracket} \text{C_CONST} \\ \\ \frac{\llbracket \Theta, x : \tau_1 \vdash e : \tau_2 \rrbracket \Rightarrow R \quad y \text{ is fresh}}{\llbracket \Theta \vdash \lambda x. e : \tau_1 \rightarrow \tau_2 \rrbracket \Rightarrow \Lambda \alpha. \lambda k. k ! (\lambda y. \text{let } !x = y \text{ in } R)} \text{C_ABS} \\ \\ \frac{\llbracket \Theta \vdash e_1 : \tau_1 \rightarrow \tau_2 \rrbracket \Rightarrow R_1 \quad \llbracket \Theta \vdash e_2 : \tau_1 \rrbracket \Rightarrow R_2 \quad x \text{ is fresh}}{\llbracket \Theta \vdash e_1 e_2 : \tau_2 \rrbracket \Rightarrow \Lambda \alpha. \lambda k. R_1 \alpha (\lambda x. R_2 \alpha (\lambda y. \text{let } !z = x \text{ in } z y \alpha k))} \text{C_APP} \\ \\ \frac{\llbracket \Theta, \beta \vdash e : \tau \rrbracket \Rightarrow R}{\llbracket \Theta \vdash e : \forall \beta. \tau \rrbracket \Rightarrow \Lambda \alpha. \lambda k. v\beta. R \alpha (\lambda x. k \textcolor{green}{\Lambda}^\circ \langle \beta, x \rangle)} \text{C_TABS} \\ \\ \frac{\llbracket \Theta \vdash e : \forall \beta. \tau_2 \rrbracket \Rightarrow R \quad \Theta \vdash \tau_1}{\llbracket \Theta \vdash e : \tau_2 [\tau_1 / \beta] \rrbracket \Rightarrow \Lambda \alpha. \lambda k. R \alpha (\lambda x. \text{let } !y = x \text{ in } k ! (y \llbracket \tau_1 \rrbracket_v))} \text{C_TAPP} \end{array}$$

Type preservation

Given a derivation D of $\Theta \vdash e : \tau$ in implicit System F,
 $\llbracket \Theta \rrbracket \vdash \llbracket D \rrbracket : \llbracket \tau \rrbracket$ is derivable in Λ^{open}

Other topics covered in the paper

- ❑ Meaning preservation of the CPS transformation
 - via Plotkin's CPS transformation
- ❑ Parametricity of Λ^{open}
 - by a step-indexed Kripke logical relation

Future directions

❑ Addressing control operators (w/ and w/o value restriction)

- Sketched for deep effect handlers in row effect typing by [Hillerström et al., FSCD'17]
- What about:
 - Other forms of effect handlers (e.g., shallow and lexically scoped handlers)?
 - Other effect typing (e.g., contextual polymorphism)?

❑ Extending to other binding constructs under which evaluation proceeds

- E.g., staged computation

Conclusion

Type-preserving CPS transformation is challenging for implicit polymorphism without the value restriction

- ❑ Addressed implicit System F

- by a new CPS target language with restrictions, open type abstractions, and affine types

- ❑ What about effectful languages like OCaml?