CPS Transformation with Affine Types for Implicit Polymorphism

Taro Sekiyama
National Institute of Informatics

Takeshi Tsukada
Chiba University

ICFP 2021

CPS transformation

- Transforming programs into continuation-passing style (CPS)

$$[\![\lambda f.\, 42 + (f\ 0)]\!] \quad = \lambda f.\, \lambda k.\, f\ 0\ \big(\lambda x.\, k\ (42 + x)\big)$$

CPS transformation

- Transforming programs into continuation-passing style (CPS)

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

- Applications

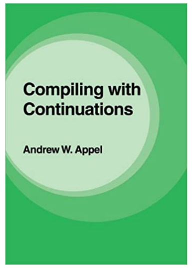
Semantics of control operators

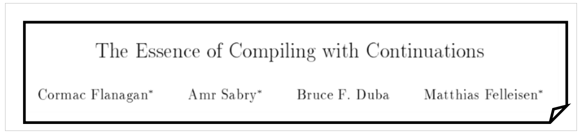
$$\llbracket \mathcal{C}\ \lambda x. e \rrbracket = \lambda k. \llbracket \lambda x. e \rrbracket (\lambda y. \lambda k'. k\ y)\ (\lambda z. z)$$

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

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

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

- Transforming programs into continuation-passing style (CPS)

$$[\![\lambda f.\, 42 + (f\ 0) : \tau]\!] = \lambda f.\, \lambda k.\, f\ 0\ \big(\lambda x.\, k\ (42 + x)\big) : [\![\tau]\!]$$

- Applications

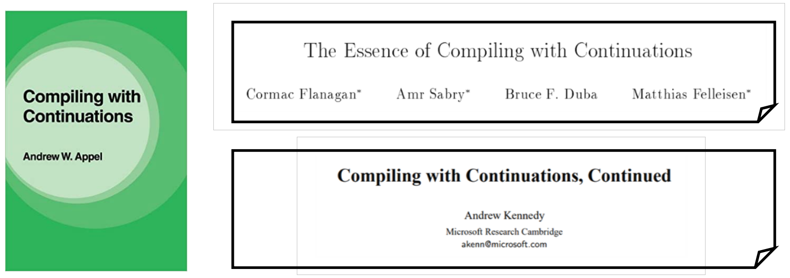
Semantics of control operators

$$[\![\mathcal{C}\ \lambda x.\, e]\!] = \lambda k.\, [\![\lambda x.\, e]\!](\lambda y.\, \lambda k'.\, k\ y)\ (\lambda z.\, z)$$
$$[\![\mathbf{shift}\ \lambda x.\, e]\!] = \lambda k.\ [\![\lambda x.\, e]\!]\ \big(\lambda y.\, \lambda k'.\, k'(k\ y)\big)\ (\lambda z.\, z)$$
$$[\![\mathbf{reset}\ e]\!] = \lambda k.\, k\ ([\![e]\!]\ (\lambda x.\, x))$$

Fine-grained type systems for control operators

$$\Gamma; \alpha \vdash e : \tau; \beta \qquad \frac{\Gamma, \mathrm{x}: \tau \to \bot \vdash e : \bot}{\Gamma \vdash \mathcal{C}\ \lambda x.\, e : \tau}$$

Compiler IRs

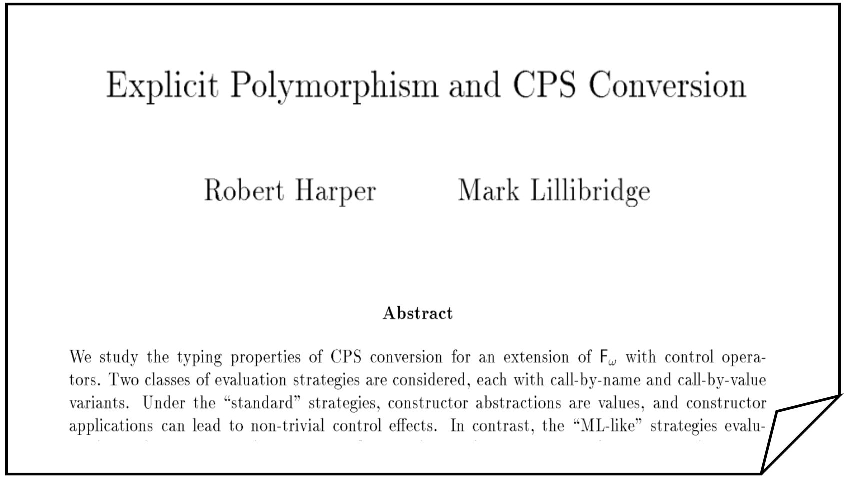


Compiler verification by typing IRs

Typed code → Typed CPS IR → Typed ASM

CPS transformation for polymorphism

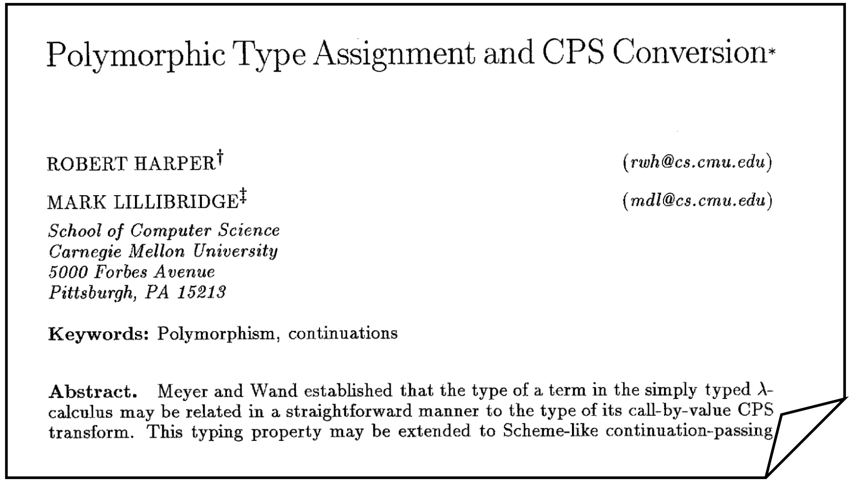
Established under *value restriction* [Harper&Lilibridge '94]
(polymorphic expressions in a source language must be values)


Explicit Polymorphism and CPS Conversion

Robert Harper   Mark Lillibridge

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 evalu-


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
Pittsburgh, PA 15213

Keywords: Polymorphism, continuations

Abstract. Meyer and Wand established that the type of a term in the simply typed λ-calculus may be related in a straightforward manner to the type of its call-by-value CPS transform. This typing property may be extended to Scheme-like continuation-passing

Problem: Not applicable to implicitly polymorphic languages w/o value restriction (like OCaml)

This work

Research question (long-term)

Is it possible to define type-preserving CPS transformation for implicit polymorphism without value restriction?

Contribution (short-term)

Showing it is possible for the implicit version of System F

Equivalent to allowing the reduction

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

Review: CPS transformation

$$[\![\lambda x.\, e]\!] = \lambda k.\, k\ \lambda x.\, [\![e]\!]$$

$$[\![x]\!] = \lambda k.\, k\ x$$

$$[\![e_1\ e_2]\!] = \lambda k.\, [\![e_1]\!]\ (\lambda x.\, [\![e_2]\!]\ (\lambda y.\, x\ y\ k))$$

Factorizing CPS transformation [Danvy'92]

1. Naming intermediate results of computation

$$\boldsymbol{e_1\ e_2 \ \Longrightarrow\ \mathbf{let}\ x = e_1\ e_2\ \mathbf{in}\ x}$$

2. Sequencing computation by lifting redexes

$$\boldsymbol{x\ (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2)\ \Longrightarrow\ \mathbf{let}\ y = e_1\ \mathbf{in}\ x\ e_2}$$

3. Making continuations explicit

Factorizing CPS transformation [Danvy'92]

1. Naming intermediate results of computation

$$\boldsymbol{e_1\ e_2 \implies \mathbf{let}\ x = e_1\ e_2\ \mathbf{in}\ x}$$

Sequencing computation by lifting redexes

$$\boldsymbol{x\ (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2) \implies \mathbf{let}\ y = e_1\ \mathbf{in}\ x\ e_2}$$

3. Making continuations explicit

Redex lifting as source-level reduction

[Sabry+'92]

$$\boldsymbol{E}[(\boldsymbol{\lambda x}{:}\,\boldsymbol{\tau}.\,\boldsymbol{e_1})\ \boldsymbol{e_2}] \ \mapsto\ (\boldsymbol{\lambda x}{:}\,\boldsymbol{\tau}.\,\boldsymbol{E}[\boldsymbol{e_1}])\ \boldsymbol{e_2}$$

where $\boldsymbol{E}$ is an evaluation context such that $\boldsymbol{x} \notin \boldsymbol{fv}(\boldsymbol{E})$

This rule conflicts with implicit polymorphism due to evaluation contexts where the hole $\square$ appears under $\boldsymbol{\Lambda}$ (like $\boldsymbol{\Lambda\alpha}.\,\square$)

Redex lifting as source-level reduction

[Sabry+'92]

$$\boldsymbol{E}[(\lambda x{:}\,\tau.\,e_1)\ e_2] \mapsto (\lambda x{:}\,\tau.\,\boldsymbol{E}[e_1])\ e_2$$

where $\boldsymbol{E}$ is an evaluation context such that $x \notin fv(\boldsymbol{E})$

Instantiate by $\boldsymbol{E}'[\Lambda\alpha.\,\square]$

This rule conflicts with implicit polymorphism due to evaluation contexts where the hole $\square$ appears under $\boldsymbol{\Lambda}$ (like $\Lambda\alpha.\,\square$)

Problem: redex lifting in implicit polymorphism

$$\boldsymbol{E'[\Lambda\alpha.\,(\lambda x{:}\,\tau.\,e_1)\;e_2]} \;\mapsto\; \boldsymbol{(\lambda x{:}\,\tau.\,E'[\Lambda\alpha.\,e_1])\;e_2}$$

$\boldsymbol{\tau}$ and $\boldsymbol{e_2}$ can refer to $\boldsymbol{\alpha}$

$\boldsymbol{\tau}$ and $\boldsymbol{e_2}$ *CANNOT* refer to $\boldsymbol{\alpha}$ as they are outside the scope of $\boldsymbol{\alpha}$

Cause: Conflict between generalization and binding by $\boldsymbol{\Lambda}$

Generalizing $\boldsymbol{\alpha}$ in $\boldsymbol{e_1}$ requires lowering $\boldsymbol{\Lambda\alpha}$	**VERSUS**	Binding $\boldsymbol{\alpha}$ in $\boldsymbol{\tau}$ and $\boldsymbol{e_2}$ requires lifting $\boldsymbol{\Lambda\alpha}$

Key idea of our solution

Decomposing $\boldsymbol{\Lambda\alpha}$ into more atomic constructors

Restrictions $\nu\alpha.\, e$

only bind $\boldsymbol{\alpha}$ (not generalize)

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

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

only generalize $\boldsymbol{\alpha}$ (not bind)

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

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

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

Examples

$$\vdash \nu\alpha.\,\Lambda^{\circ}\langle\alpha, \lambda x{:}\,\alpha.\,x\rangle \;:\; \forall\alpha.\,\alpha \to \alpha$$

$$\nvdash \quad \Lambda^{\circ}\langle\alpha, \lambda x{:}\,\alpha.\,x\rangle \;:\; \forall\alpha.\,\alpha \to \alpha$$

$$\alpha, x{:}\,\alpha \to \alpha \vdash \quad \Lambda^{\circ}\langle\alpha, x\rangle \;:\; \forall\alpha.\,\alpha \to \alpha$$

Restrictions $\nu\alpha.\,e$	Open type abstractions $\Lambda^{\circ}\langle\alpha, e\rangle$
$\dfrac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \nu\alpha.\,e : \tau}$	$\dfrac{\Gamma \vdash e : \tau \quad \alpha \in \Gamma}{\Gamma \vdash \Lambda^{\circ}\langle\alpha.\,e\rangle : \forall\alpha.\,\tau}$

Solution: redex lifting with ν and Λ°

$$\Lambda\alpha.\, e \equiv \nu\alpha.\, \Lambda^{\circ}\langle \alpha, e\rangle$$

$$\begin{aligned} & E'[\Lambda\alpha.\, (\lambda x{:}\,\tau.\, e_1)\ e_2] \\ = {} & E'[\nu\alpha.\, \Lambda^{\circ}\langle \alpha, (\lambda x{:}\,\tau.\, e_1)\ e_2\rangle] \end{aligned}$$

Solution: redex lifting with ν and Λ°

$$E'[\Lambda\alpha.\,(\lambda x{:}\,\tau.\,e_1)\ e_2]$$

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

$$\mapsto$$

Solution: redex lifting with ν and Λ°

Step 1: lifting ν

$$E'[\Lambda\alpha.\,(\lambda x{:}\,\tau.\,e_1)\;e_2]$$

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

$$\mapsto$$

Solution: redex lifting with ν and Λ°

Step 1: lifting ν

$$E'[\Lambda\alpha.\,(\lambda x{:}\,\tau.\,e_1)\;e_2]$$

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

Step 2: lifting the redex
(i.e., lowering the evaluation context)

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

Requirements for typing	Generalizing α in e_1	Binding α in τ and e_2
How addressed?	By lowering $\Lambda^\circ\langle\alpha,\square\rangle$	By lifting $\nu\alpha$

What could be obtained

Type safe

Implicit System F

CPS trans.

$$\xRightarrow{[\![\cdot]\!]}$$

Not type safe

System F with

$\boldsymbol{\nu} + \boldsymbol{\Lambda}^{\circ}$

Unsafety by re-generalization

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

$$\vdash \boldsymbol{M} : \forall \boldsymbol{\alpha}.\, \boldsymbol{\alpha} \to \forall \boldsymbol{\alpha}.\, \boldsymbol{\alpha} \to \boldsymbol{\alpha}$$

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

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

Restrictions $\nu\alpha.\, e$	**Open type abstractions $\Lambda^\circ\langle\alpha, e\rangle$**
$\dfrac{\boldsymbol{\Gamma}, \boldsymbol{\alpha} \vdash \boldsymbol{e} : \boldsymbol{\tau}}{\boldsymbol{\Gamma} \vdash \boldsymbol{\nu\alpha}.\, \boldsymbol{e} : \boldsymbol{\tau}}$	$\dfrac{\boldsymbol{\Gamma} \vdash \boldsymbol{e} : \boldsymbol{\tau} \quad \boldsymbol{\alpha} \in \boldsymbol{\Gamma}}{\boldsymbol{\Gamma} \vdash \boldsymbol{\Lambda^\circ\langle\alpha.\, e\rangle} : \forall \boldsymbol{\alpha}.\, \boldsymbol{\tau}}$

Unsafety by re-generalization

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

Cause: The same type variable may be generalized multiple times

Solution: Using affine typing to enforce type variables are generalized only once

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

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

What has been obtained

Type safe

Implicit System F

CPS trans.

$[\![\cdot]\!]$

$\Longrightarrow$

Type safe

$\boldsymbol{\Lambda}^{\mathbf{open}}$

(System F w/ $\boldsymbol{\nu}$ + $\boldsymbol{\Lambda}^{\circ}$
+ affine typing)

Other topics covered in the paper

- ❑Details of $\mathbf{\Lambda^{open}}$ and the CPS transformation
- ❑Meaning preservation of the CPS transformation
- ❑Parametricity of $\mathbf{\Lambda^{open}}$

Conclusion

Challenging to obtain type-preserving CPS transformation for implicit polymorphism w/o value restriction

- This work addressed implicit System F as a first step
  - Proposed a new CPS target language with restrictions, open type abstractions, and affine types
- Future work: support for other features
  - Effects
  - Other binding constructs under which evaluation proceeds (e.g., staged computation)