Verifying a CPS Transformation

Martijn Vermaat

mvermaat@cs.vu.nl
http://www.cs.vu.nl/~mvermaat/

Bachelor Project
December 20, 2007

Continuation-Passing Style

Direct style

Function returns the result of its computation.
Example: $\lambda x.\, x - 2$

Continuation-Passing Style

Direct style

Function returns the result of its computation.
Example: $\lambda x.\, x - 2$

Continuation passing style

Function passes result to a *continuation*.
Example: $\lambda x\, k.\, k\,(x - 2)$

Continuation-Passing Style

Direct style

Function returns the result of its computation.
Example: $\lambda x.\, x - 2$

Continuation passing style

Function passes result to a *continuation*.
Example: $\lambda x\, k.\, k\,(x - 2)$

- Order of evaluation is fixed
- Suitable for aggressive optimizations

Transforming to CPS

Programs in direct style can be mechanically transformed to equivalent prorgrams in CPS:

- Plotkin, 1975
- Danvy and Nielsen, 2003
- ...and many more

Plotkin's Original Transformation

Plotkin, 1975:

$$
\begin{aligned}
[\![x]\!] &= \lambda k.\, k\, x \\
[\![\lambda x.\, M]\!] &= \lambda k.\, k\, (\lambda x.\, [\![M]\!]) \\
[\![M\, N]\!] &= \lambda k.\, [\![M]\!]\, (\lambda m.\, [\![N]\!]\, (\lambda n.\, m\, n\, k))
\end{aligned}
$$

Plotkin's Original Transformation

Plotkin, 1975:

$$
\begin{aligned}
[\![x]\!] &= \lambda k.\, k\, x \\
[\![\lambda x.\, M]\!] &= \lambda k.\, k\, (\lambda x.\, [\![M]\!]) \\
[\![M\, N]\!] &= \lambda k.\, [\![M]\!]\, (\lambda m.\, [\![N]\!]\, (\lambda n.\, m\, n\, k))
\end{aligned}
$$

Generates many administrative redexes:

$$
\begin{aligned}
[\![(\lambda x.\, x)\, y]\!] &= \lambda k.\, (\lambda k.\, k\, (\lambda x.\, (\lambda k.\, k\, x)))\, (\lambda m.\, (\lambda k.\, k\, y)\, (\lambda n.\, m\, n\, k)) \\
&\to_\beta \lambda k.\, (\lambda m.\, (\lambda k.\, k\, y)\, (\lambda n.\, m\, n\, k))\, (\lambda x.\, (\lambda k.\, k\, x)) \\
&\to_\beta \lambda k.\, (\lambda k.\, k\, y)\, (\lambda n.\, (\lambda x.\, (\lambda k.\, k\, x))\, n\, k) \\
&\to_\beta \lambda k.\, (\lambda n.\, (\lambda x.\, (\lambda k.\, k\, x))\, n\, k)\, y \\
&\to_\beta \lambda k.\, (\lambda x.\, (\lambda k.\, k\, x))\, y\, k
\end{aligned}
$$

Plotkin Optimized

Slightly optimized version of Plotkin's original:

$$
\begin{aligned}
[\![x]\!] \triangleright k &= k\,x \\
[\![\lambda x.\, M]\!] \triangleright k &= k\,(\lambda x\, k.\, [\![M]\!] \triangleright k) \\
[\![M\,N]\!] \triangleright k &= [\![M]\!] \triangleright \lambda m.\, [\![N]\!] \triangleright \lambda n.\, m\,n\,k
\end{aligned}
$$

Plotkin Optimized

Slightly optimized version of Plotkin's original:

$$
\begin{aligned}
[\![x]\!] \triangleright k &= k\,x \\
[\![\lambda x.\,M]\!] \triangleright k &= k\,(\lambda x\,k.\,[\![M]\!] \triangleright k) \\
[\![M\,N]\!] \triangleright k &= [\![M]\!] \triangleright \lambda m.\,[\![N]\!] \triangleright \lambda n.\,m\,n\,k
\end{aligned}
$$

Generates fewer administrative redexes:

$$
\begin{aligned}
[\![(\lambda x.\,x)\,y]\!] \triangleright k &= (\lambda m.\,(\lambda n.\,m\,n\,k)\,y)\,(\lambda x\,k.\,k\,x) \\
&\to_\beta (\lambda n.\,(\lambda x\,k.\,k\,x)\,n\,k)\,y \\
&\to_\beta (\lambda x\,k.\,k\,x)\,y\,k
\end{aligned}
$$

Program Compilation

Traditionally:

1. Source program is proven correct
2. Source program is compiled to binary code
3. Binary code is executed

Program Compilation

Traditionally:

1. Source program is proven correct
2. Source program is compiled to binary code
3. Binary code is executed

Observation: No guarantees on correctness of executed program.

Roads to Correct Compilers

Gap between correct source code and correct binary code. Ways to fill it:

- Prove correctness of the compiler
- Validate compilation result
- Use proof-carrying code

We will focus on the first.

Proving a Compiler Correct

Compiler is a function C : Source code $\rightarrow$ Target code

Correctness of a compiler

Amounts to:

- Semantics(S) = Semantics$(C(S))$
- Observable behaviour(S) = Observable behaviour$(C(S))$
- $P(S) \Rightarrow P(C(S))$
- $P(S) \Rightarrow P'(C(S))$
- ...

Proving a Compiler Correct

Compiler is a function C : Source code $\rightarrow$ Target code

Correctness of a compiler

Amounts to:

- Semantics(S) = Semantics($C(S)$)
- Observable behaviour(S) = Observable behaviour($C(S)$)
- $P(S) \Rightarrow P(C(S))$
- $P(S) \Rightarrow P'(C(S))$
- ...

C is composed of many stages, one of which may be a CPS transformation.

Mechanized Verification of CPS Transformations

Zaynah Dargaye and Xavier Leroy: *Mechanized verification of CPS transformations* (LPAR 2007)

- Part of the Compcert project (INRIA)
- Use Coq to implement and verify two CPS transformations
- Relatively interesting source and target languages
- No source code available (yet)

We will look at a simplified version of their setting.

Source Language

$$
\begin{array}{rll}
M ::= & x_n & \text{variable} \\
\mid & \lambda^n.M & \text{abstraction of arity } n+1 \\
\mid & M(M, \ldots, M) & \text{function application} \\
\mid & \texttt{let}\ M\ \texttt{in}\ M & \text{binding}
\end{array}
$$

Example term: $(\lambda^0.\, x_0)\,(\lambda^1.\, x_0)$

- de Bruijn indices
- $\lambda^n.\, M$ binds $x_n, \ldots, x_0$ in M
- $\texttt{let}\ M\ \texttt{in}\ N$ binds x_0 to M in N
- Big-step operational semantics (next slide)

Source Language Semantics

$$\frac{}{\lambda^n.M \Rightarrow \lambda^n.M} \qquad \frac{M \Rightarrow v_1 \qquad N\{v_1\} \Rightarrow v}{(\mathtt{let}\ M\ \mathtt{in}\ N) \Rightarrow v}$$

$$\frac{M \Rightarrow \lambda^n.P \qquad N_i \Rightarrow v_i \qquad P\{v_n, \ldots, v_0\} \Rightarrow v}{M(N_0, \ldots, N_n) \Rightarrow v}$$

Example evaluation:

$$(\lambda^0.\, x_0)\,(\lambda^1.\, x_0) \;\Rightarrow\; (\lambda^1.\, x_0)$$

Target Language

$$
\begin{aligned}
M' ::= {} & x_n && \text{source-level variable} \\
\mid {} & \kappa_n && \text{continuation variable} \\
\mid {} & \lambda^n . M' && \text{abstraction of arity } n+1 \\
\mid {} & M'(M', \ldots, M') && \text{function application} \\
\mid {} & \texttt{let } M' \texttt{ in } M' && \text{binding}
\end{aligned}
$$

Example term: $\lambda^0 .\, \kappa_0$ (the initial continuation)
Example term: $\lambda^0 .\, (\lambda^1 .\, \kappa_0\,(x_0))\,(\kappa_0, (\lambda^2 .\, \kappa_0\,(x_0)))$

- Two sorts of variables (indepenently numbered)
- $\lambda^n .\, M$ binds $\kappa_0, x_{n-1}, \ldots, x_0$ in M

Target Language Semantics

$$\frac{}{\lambda^n.M' \Rightarrow \lambda^n.M'} \qquad \frac{M' \Rightarrow v_1 \qquad N'\{\}\{v_1\} \Rightarrow v}{(\mathtt{let}\ M'\ \mathtt{in}\ N') \Rightarrow v}$$

$$\frac{M' \Rightarrow \lambda^n.P' \qquad N'_i \Rightarrow v_i \qquad P'\{v_0\}\{v_n, \ldots, v_1\} \Rightarrow v}{M'(N'_0, \ldots, N'_n) \Rightarrow v}$$

Example evaluation:

$$(\lambda^1.\, \kappa_0\,(x_0))\,(\kappa_0, (\lambda^2.\, \kappa_0\,(x_0))) \;\Rightarrow\; \kappa_0\,(\lambda^2.\, \kappa_0\,(x_0))$$

CPS Transformation

$[\![\cdot]\!]$ is an extension of Plotkin's original transformation:

$$
\begin{aligned}
\Psi(x_n) &= x_n \\
\Psi(\lambda^n.M) &= \lambda^{n+1}.[\![M]\!](\kappa_0)
\end{aligned}
$$

$$
\begin{aligned}
[\![A]\!] &= \lambda^0.\kappa_0(\Psi(A)) \\
[\![M(N_1, \ldots, N_n)]\!] &= \lambda^0.[\![M.N_1 \ldots N_n \texttt{ then } \kappa_n(\kappa_{n+1}, \kappa_{n-1}, \ldots, \kappa_0)]\!] \\
[\![\texttt{let } M \texttt{ in } N]\!] &= \lambda^0.[\![M]\!](\lambda^0.\texttt{let } \kappa_0 \texttt{ in } [\![N]\!](\kappa_1)) \\
[\![M_1 \ldots M_n \texttt{ then } N']\!] &= [\![M_1]\!](\lambda^0 \ldots [\![M_n]\!](\lambda^0.N') \ldots)
\end{aligned}
$$

CPS Transformation Example

On a slide like this, just $[\![(\lambda^0.x_0)(\lambda^1.x_0)]\!]$ is quite a term:

$$\begin{aligned}&\lambda^0.\,\big(\lambda^0.\kappa_0(\lambda^1.\,(\lambda^0.\kappa_0(x_0))\,(\kappa_0)\,)\big)\\&\qquad\big(\lambda^0.\,(\lambda^0.\kappa_0(\lambda^2.(\lambda^0.\kappa_0(x_0))(\kappa_0)))\,(\lambda^0.\kappa_1(\kappa_2,\kappa_0))\,\big)\end{aligned}$$

Correctness of $[\![\cdot]\!]$

What does it mean for $[\![\cdot]\!]$ to be correct?

Correctness of $[\![\cdot]\!]$

What does it mean for $[\![\cdot]\!]$ to be correct?

- $[\![\cdot]\!]$ preserves semantics

Correctness of $[\![\cdot]\!]$

What does it mean for $[\![\cdot]\!]$ to be correct?

- $[\![\cdot]\!]$ preserves semantics
- How do we express this?
  (This will not work: $[\![M]\!] \Rightarrow v$ whenever $M \Rightarrow v$)

Correctness of $[\![\cdot]\!]$

What does it mean for $[\![\cdot]\!]$ to be correct?

- $[\![\cdot]\!]$ preserves semantics
- How do we express this?
  (This will not work: $[\![M]\!] \Rightarrow v$ whenever $M \Rightarrow v$)

Correctness Theorem

If $M \Rightarrow v$ in the source language, then $[\![M]\!](\lambda^0.\kappa_0) \Rightarrow \Psi(v)$ in the target language.

Proof of Correctness Theorem

Correctness Theorem

If $M \Rightarrow v$ in the source language, then $[\![M]\!](\lambda^0.\kappa_0) \Rightarrow \Psi(v)$ in the target language.

More general result needed for proof by induction:

Lemma 1

Let $K = \lambda^0.P$ be a κ-closed, one-argument abstraction of the target language. If $M \Rightarrow v$ in the source language, and $P\{\Psi(v)\}\{\} \Rightarrow v'$ in the target language, then $[\![M]\!](K) \Rightarrow v'$ in the target language.

Proof of Lemma 1

Lemma 1

Let $K = \lambda^0.P$ be a κ-closed, one-argument abstraction of the target language. If $M \Rightarrow v$ in the source language, and $P\{\Psi(v)\}\{\} \Rightarrow v'$ in the target language, then $[\![M]\!](K) \Rightarrow v'$ in the target language.

Proof.

By induction on the evaluation derivation of $M \Rightarrow v$ and case analysis over the term M:

Base case $M = \lambda^n.M_1$

Inductive case $M = M_1(N_0, \ldots, N_n)$

Premises: $M_1 \Rightarrow \lambda^n.Q$, $N_i \Rightarrow v_i$, $Q\{v_n, \ldots, v_0\} \Rightarrow v$

Inductive case $M = \texttt{let}\ M_1\ \texttt{in}\ M_2$

Premises: $M_1 \Rightarrow v_1$, $M_2\{v_1\} \Rightarrow v$ $\square$

Coq Development

Zaynah Dargaye and Xavier Leroy: *Mechanized verification of CPS transformations* (LPAR 2007)

- Mechanization of proof in Coq
- Extraction of executable Caml code
- Also: recursive functions, pattern matching, optimized transformation
- $\approx$ 9000 lines of Coq
- Goal: verified mini-ML to Cminor compiler (part of Compcert)

Design Choices

(Two-sorted) de Bruijn indices:

- Binding and α-conversion (POPLmark challenge)
- Lifting

Design Choices

(Two-sorted) de Bruijn indices:

- Binding and α-conversion (POPLmark challenge)
- Lifting

Big-step operational semantics:

- Administrative redexes
- No diverging source programs

Design Choices

(Two-sorted) de Bruijn indices:

- Binding and α-conversion (POPLmark challenge)
- Lifting

Big-step operational semantics:

- Administrative redexes
- No diverging source programs

Coq:

- Code extraction
- Preference

Related Work

Minamide and Okuma, 2003: *Verifying CPS Transformations in Isabelle/HOL*

- Named variables, no α-conversion or explicit renaming
- Small-step operational semantics

Tian, 2006: *Mechanically Verifying Correctness of CPS Compilation*

- Twelf
- Higher-order abstract syntax
- Combination of big-step and small-step operational semantics

Chlipala, 2007: *A Certified Type-Preserving Compiler from Lambda Calculus to Assembly Language*

- Coq
- de Bruijn indices
- Denotational semantics
- Dependent types with focus on automated proofs

Conclusion

- Verifying transformations such as CPS is tedious, but necessary
- α-conversion is a big problem in mechanized proofs
- Closer look at related work is needed for more conclusions

Questions and Further Reading

Questions?

Further Reading

- Plotkin, 1975: *Call-by-name, Call-by-value and the lambda-calculus*
- Danvy and Filinski, 1992: *Representing control: a study of the CPS transformation*
- Appel, 1992: *Compiling with Continuations*
- Compcert: `http://pauillac.inria.fr/~xleroy/compcert/`
- Lambda Tamer: `http://ltamer.sourceforge.net/`