Type Preserving Compilation in Haskell

Stefan Monnier
DIRO - Université de Montréal
Overview

Motivation: why show type-preservation?

Representing CPS for STLC in Haskell

Showing CPS’s type preservation

Extending to System F

Alternative representations
Program verification

Many techniques exist to verify properties exhaustively

- Hoare logic: Encode in logic the semantics of the language, the specification of the program, then prove correctness
- Automatic program derivation: Write a clean and naive version of the program, plus automatically-verified transformations
- Model checking: Abstract a model of some aspect of the code; exhaustively and automatically check it against a specification
- Ad-hoc analyses: Look for known error patterns; often imprecise
- Type checking: modular verification against a predefined set of rules

Usually can’t be all used at the same time
Type checking

By far the most popular formal method

Fully integrated in the toolchain: “no extra work”

Good error reporting, directly pointing to offending line

Early detection $\Rightarrow$ cheap fix

Guaranteed 100% pure formal: checks the actual code

Example, to eliminate NULL dereferences:

```haskell
data Maybe a = Nothing | Just a
```

- **String**: type of strings; can’t be “NULL”!
- **Maybe String**: this one can be “NULL”!
Trusted Computing Base (TCB)

Type checking can verify important properties about source code.

But is the machine code really faithful to your source code?

What about the runtime functions?

Usually all these are part of the TCB.

The compiler and runtime are complex programs, hard to trust.

Proving their full semantic correctness is hard and costly.
Type preservation

Rather than ensure full correctness, concentrate on *type preservation*

- Input and output terms of code transformations have related type

Model-checking or Hoare-style proofs might still be broken

*But* type-based properties of source code are faithfully preserved!

Sweet spot:

- Ensuring type preservation should be cheap(er)
- Most programs have no formal property to preserve other than types
Type preservation

Rather than ensure full correctness, concentrate on type preservation

- Input and output terms of code transformations have related type

Model-checking or Hoare-style proofs might still be broken

But type-based properties of source code are faithfully preserved!

Sweet spot:

- Ensuring type preservation should be cheap(er)

- Most programs have no formal property to preserve other than types

Note: do not confuse the object compiler and the meta compiler!
Typed Intermediate Languages

Don’t throw away type info after type checking

Make all intermediate languages typed

Perform type-checking after each compilation phase

• Extra cost to store and compute the type information
• Extra cost of running the type checker
• Lost optimization opportunities
• Amounts to testing type-preservation
• Can only point to a whole compilation phase as the culprit
Type checking can eliminate run-time errors, of course.

It can also provide more guarantees about behavior:

- \texttt{iton: Int \rightarrow String} guarantees we will not get NULL
- \texttt{id: \alpha \rightarrow \alpha} guarantees preservation of type

Rich type systems can enforce precise invariants about programs.

Proofs are intertwined with the program.

When the proof’s structure matches the program’s
\(
\Rightarrow \text{ most of the proof (the boring part) comes for free}
\)
Verifying type preservation

Express type preservation in the types. I.e., instead of:

\[ e : \text{SrcExp} \rightarrow \text{Code Transformation} \rightarrow e' : \text{DstExp} \]

Use:

\[ e : \text{SrcExp } \tau \rightarrow \text{Code Transformation} \rightarrow e' : \text{DstExp } \tau \]

\( e : \text{SrcExp } \tau \) means:

- \( e \) is a Haskell expression that evaluates to an AST
- That AST represents a source expression of type \( \tau \)

We can use type-checking to verify the type preservation!
Type-based verification vs TIL

Expected advantages:

- No extra time spent manipulating types while running the compiler
- Errors detected earlier (during type-checking rather than at runtime)
- Errors linked directly to offending source code
- Potentially less work if proof mirrors code

Might still cause lost optimization opportunities

Requires advanced type systems

Sometimes imposes a change in representation
Two common code transformations in compilers for functional languages

\[ e : \tau \xrightarrow{\text{CPS}} \xrightarrow{\text{Closure Conversion}} e' : \tau \]

**CPS:**
- Only use tail-calls, equivalent to `goto`
- Make the return addresses and stack explicit

**Closure conversion:**
- Make closure construction explicit
- All functions are closed and can be hoisted to top-level
Why CPS and closure conversion

CPS is the ideal case:

- The proof mirrors the code exactly
- Code is never moved from one scope to another

Closure conversion is challenging:

- Safety depends on correctness of some parts
- Scoping is affected, so need to reason explicitly about environments
- Perceived to be impossible with HOAS

Will focus on CPS today
**Source language**

Source language is simply typed lambda calculus (STLC), e.g.:

\[
\begin{align*}
\tau & ::= \text{Int} \mid \text{Bool} \mid \tau_1 \rightarrow \tau_2 \\
e & ::= n \mid \text{if } e_1 \ e_2 \ e_3 \mid e_1 \ e_2 \mid \lambda x \rightarrow e \mid x
\end{align*}
\]

Could be encoded in Haskell as:

```
data Exp where
  Num :: Int \rightarrow Exp
  If  :: Exp \rightarrow Exp \rightarrow Exp \rightarrow Exp
  App :: Exp \rightarrow Exp \rightarrow Exp
  Lam :: Var \rightarrow Exp \rightarrow Exp
  Vref :: Var \rightarrow Exp
```

Stefan Monnier - Typed compiler 13
Annotated source language

Annotate $Exp$ with the source-level type of the expression

```
data $Exp \, \tau$ where

  $Num$ :: $Int \to Exp \, Int$

  $If$ :: $Exp \, Bool \to Exp \, \tau \to Exp \, \tau \to Exp \, \tau$

  $App$ :: $Exp \, (\tau_1 \to \tau_2) \to Exp \, \tau_1 \to Exp \, \tau_2$

  \ldots \ldots

We reuse/abuse Haskell types in red to denote source types

Does it really express a valid typing?
Judgment $\vdash e : \tau$ means $e$ has type $\tau$

\[
\begin{align*}
\vdash e_1 : \text{Bool} & \quad \vdash e_2 : \tau & \quad \vdash e_3 : \tau \\
\vdash n : \text{Int} & \quad \vdash \text{if } e_1 e_3 e_3 : \tau \\
\vdash e_1 : \tau_1 \rightarrow \tau_2 & \quad \vdash e_2 : \tau_1 \\
& \quad \vdash e_1 e_2 : \tau_2
\end{align*}
\]

We can see that

\[
e :: \text{Exp } \tau \quad \sim \quad \vdash [e] : \tau
\]

Where $[e]$ converts $e$ from Haskell back to its STLC source
Typing rules in Haskell

- $\vdash [e] : \tau$
- $e :: \text{Exp } \tau$
- $\vdash n : \text{Int}$
- $\text{Num} :: \text{Int} \rightarrow \text{Exp } \text{Int}$
- $\vdash e_1 : \text{Bool}$
- $\vdash e_2 : \tau$
- $\vdash e_3 : \tau$
- $\vdash \text{if } e_1 e_3 e_3 : \tau$
- $\text{If} :: \text{Exp } \text{Bool} \rightarrow \text{Exp } \tau \rightarrow \text{Exp } \tau \rightarrow \text{Exp } \tau$
- $\vdash e_1 : \tau_1 \rightarrow \tau_2$
- $\vdash e_2 : \tau_1$
- $\vdash e_1 e_2 : \tau_2$
- $\text{App} :: \text{Exp } (\tau_1 \rightarrow \tau_2) \rightarrow \text{Exp } \tau_1 \rightarrow \text{Exp } \tau_2$
Curry-Howard isomorphism states:

\[ \text{Types} \Leftrightarrow \text{Propositions} \]

\[ \text{Expressions} \Leftrightarrow \text{Proofs} \]

In our case:

\[ e :: \text{Exp } \tau \Leftrightarrow \vdash [e] : \tau \]

Our expressions \( e \) of type \( \text{Exp } \tau \) play double role:

- Encode syntax of source term
- Encode typing derivation

We say they are \textit{intrinsically typed}.

Important simplification!
Compilers often represent variables as strings or unique integers

For practical typing we need something else

\[
\text{Lam} :: \ Var \ \tau_1 \rightarrow \ Exp \ \tau_2 \rightarrow \ Exp \ (\tau_1 \rightarrow \tau_2)
\]

\[
\text{Vref} :: \ Var \ \tau \rightarrow \ Exp \ \tau
\]

How do we make sure that a \text{Var} \ \tau indeed refers to a var of type \tau?

- Higher-order abstract syntax: elegant when it works
- DeBruijn indices: frustrating but pragmatic and universal
Higher-order abstract syntax (HOAS)

Use meta-level variables to represent object-level variables:

\[
\begin{align*}
\vdash x : \tau_1 \\
\vdash \vdots \\
\vdash e : \tau_2 \\
\vdash \lambda x \to e : \tau_1 \to \tau_2
\end{align*}
\]

\[
\begin{align*}
Lam :: (Exp \, \tau_1 \to Exp \, \tau_2) \\
\to Exp (\tau_1 \to \tau_2)
\end{align*}
\]

No \textit{Vref} case!

E.g. a source code like \( \lambda f \to \lambda x \to f \, x \) would be represented as

\[
Lam (\lambda f \to Lam (\lambda x \to App f \, x))
\]
CPS transformation

Make the return addresses and stack explicit; Eliminate non-tail calls

Functions never return

\[ \text{inc } x = x + 1 \quad \text{becomes} \quad \text{inc } (x, k) = k (x + 1) \]

\[ \ldots \quad \text{let} \ z = \text{inc } a \quad \text{becomes} \quad \ldots \quad \text{inc } (a, \lambda z \rightarrow \ldots) \]

Name all intermediate values

Explicit ordering of evaluation

Remaining (tail) calls correspond to simple branch instructions

Can be used to compile exception handling and call/cc
Sketching type preservation of CPS

\[
\begin{align*}
\llbracket \tau_1 \rightarrow \tau_2 \rrbracket & \Rightarrow (\llbracket \tau_1 \rrbracket, \llbracket \tau_2 \rrbracket \rightarrow \emptyset) \rightarrow \emptyset \\
\llbracket \text{Int} \rrbracket & \Rightarrow \text{Int} \\
\llbracket x \rrbracket k & \Rightarrow k \; x \\
\llbracket \lambda x \rightarrow e \rrbracket k & \Rightarrow k \; (\lambda (x, k') \rightarrow \llbracket e \rrbracket k') \\
\llbracket e_1 \; e_2 \rrbracket k & \Rightarrow \llbracket e_1 \rrbracket (\lambda x_1 \rightarrow \llbracket e_2 \rrbracket (\lambda x_2 \rightarrow x_1 (x_2, k)))
\end{align*}
\]

Type preservation lemma looks like

\[
\vdash e : \tau \quad \Rightarrow \quad \vdash \llbracket e \rrbracket : (\llbracket \tau \rrbracket \rightarrow \emptyset) \rightarrow \emptyset
\]
Representing CPS converted code in Haskell

data \( \text{ValK} \ \tau \) where

\[
\begin{align*}
\text{NumK} &: \text{Int} \rightarrow \text{ValK} \ \text{Int} \\
\text{PairK} &: \text{ValK} \ \tau_1 \rightarrow \text{ValK} \ \tau_2 \rightarrow \text{ValK} (\tau_1, \tau_2) \\
\text{LamK} &: (\text{ValK} \ \tau_1 \rightarrow \text{ExpK}) \rightarrow \text{ValK} (\tau_1 \rightarrow \emptyset) \\
\cdots & \quad \cdots
\end{align*}
\]

data \( \text{ExpK} \) where

\[
\begin{align*}
\text{IfK} &: \text{ValK} \ \text{Bool} \rightarrow \text{ExpK} \rightarrow \text{ExpK} \rightarrow \text{ExpK} \\
\text{AppK} &: \text{ValK} (\tau_1 \rightarrow \emptyset) \rightarrow \text{ValK} \ \tau_1 \rightarrow \text{ExpK} \\
\cdots & \quad \cdots
\end{align*}
\]
CPS type preservation in Haskell

<table>
<thead>
<tr>
<th>type family</th>
<th>CPS $\tau$ — Implements $[\tau]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>type instance</td>
<td>CPS Int = Int</td>
</tr>
<tr>
<td>type instance</td>
<td>CPS $(\tau_1 \rightarrow \tau_2) = (CPS \tau_1, (CPS \tau_2) \rightarrow \emptyset) \rightarrow \emptyset$</td>
</tr>
</tbody>
</table>

```
cps :: Exp $\tau$ → ((ValK ((CPS $\tau$) → $\emptyset$)) → ExpK)
```

Intrinsic typing:

- Exp, ExpK, ValK encode typing derivations (and syntax trees)
- cps encodes the proof of type preservation (and the transformation)

**cps** is an executable proof!
Sample of CPS transformation

\[[e_1 e_2] \ k \Rightarrow [e_1] (\lambda x_1 \rightarrow [e_2] (\lambda x_2 \rightarrow x_1(x_2, k)))\]

Ignoring a bit of gymnastic needed to work with HOAS in Haskell:

\[
cps :: Exp \tau \rightarrow ((ValK ((CPS \tau) \rightarrow \emptyset)) \rightarrow ExpK)
\]

\[
cps (App e_1 e_2) \ k =
\]
\[
cps e_1 (\lambda v_1 \rightarrow
cps e_2 (\lambda v_2 \rightarrow
AppK v_1 (PairK v_2 (LamK k)))
\]

Look ma! No types! Type preservation for free!
System F

STLC is very limited \( \Rightarrow \) extend with parametric polymorphism:

\[
\tau ::= \ldots | \tau_1 \to \tau_2 | \forall t.\tau | t
\]

\[
e ::= \ldots | e_1 e_2 | \lambda x \to e | x | \Lambda t.e | e[\tau]
\]

2 new bindings and 1 new kind of variable

Using HOAS, it would be neatly encoded in Haskell as:

\[
TLam :: (\forall t.\text{Exp}\ \tau) \to \text{Exp}\ (\forall (\lambda t.\tau))
\]

\[
TApp :: \text{Exp}\ (\forall f) \to \text{Exp}\ (f \ \tau)
\]

But GHC does not support type-level \( \lambda \)!
System F in Haskell

Use de Bruijn indices to represent the variables bound by $\forall$:

\[
data Z; \text{data } S \tau
\]

\[
data Var \tau; \text{data } All \tau
\]

So $\forall t_1. \forall t_2. t_1 \rightarrow t_2$ is represented as $All (All ((S Z) \rightarrow Z))$

\[
TLam :: (\forall t. \text{Exp} (\text{Subst } \tau t Z)) \rightarrow \text{Exp} (All \tau)
\]

\[
TApp :: \text{Exp} (All \tau_1) \rightarrow \text{Exp} (\text{Subst } \tau_1 \tau_2 Z)
\]

We still use HOAS for the $\Lambda$ binding

We have to implement the type-level function $\text{Subst}$
Substitution

de Bruijn indices use type-level naturals $Z$ and $S \tau$

$Subst$ requires type-level operation on naturals

In the $TApp$ and $TLam$ cases of $cps$, we need to show commutativity

$$CPS (Subst \tau_1 \tau_2 n) \equiv Subst (CPS \tau_1) (CPS \tau_2) n$$

GHC does not offer any way to prove it at the type-level:

$$\text{subst\_cps\_commute :: TypeRep } \tau_1 \rightarrow \text{ TypeRep } \tau_2$$

$$\rightarrow \text{Equiv} (CPS (Subst \tau_1 \tau_2 n))(Subst (CPS \tau_1) (CPS \tau_2) n)$$

If we could use HOAS, we’d get $Subst$ and commutativity for free
CPS without CPS

CPS type-level function can be eliminated:

- Keep $\tau_1 \rightarrow \tau_2$ as the representation for “normal functions”
- Replace $\tau \rightarrow \emptyset$ with new $\text{Cont } \tau$ for continuations

We lose the equivalence $\tau_1 \rightarrow \tau_2 \equiv \text{Cont } (\tau_1, \text{Cont } \tau_2)$

This equivalence is not really used

We need two different $\text{LamK}$ and two different $\text{AppK}$

A bit of code duplication, but avoid need for $\text{subst_cps_commute}$

Still need to define $\text{Subst}$ and type-level naturals
CPS with de Bruijn

HOAS can’t always be used (e.g. for closure conversion)

Other typing judgment \[ \Gamma \vdash e : \tau \]

\[
\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x \rightarrow e : \tau_1 \rightarrow \tau_2}
\]

\[
\Gamma(x) = \tau
\]

\[
\Gamma \vdash x : \tau
\]

Corresponding Haskell judgment \[ e :: Exp \Gamma \tau \]

data Exp \(\Gamma \tau\) where

\[\text{Lam} :: \ Exp (\Gamma, \tau_1) \tau_2 \rightarrow \ Exp \Gamma (\tau_1 \rightarrow \tau_2)\]

\[\text{Vref} :: \ VarIn \Gamma \tau \rightarrow \ Exp \Gamma \tau\]
Represent the proof of membership by the index in the set

\[
\text{data } \textit{VarIn} \; \Gamma \; \tau \; \text{where}
\]

\[
\begin{align*}
\textit{VarZ} &::= \textit{VarIn} \; (\Gamma, \tau) \; \tau \\
\textit{VarS} &::= \textit{VarIn} \; \Gamma \; \tau_1 \rightarrow \textit{VarIn} \; (\Gamma, \tau_2) \; \tau_1
\end{align*}
\]

But $\Gamma$ in source and $\Gamma$ after CPS are different

Need extra rewriting and corresponding proofs

A lot more work than with HOAS

Had to use it for closure conversion
Type-preserving CPS and closure conversion of System F in Haskell

CPS part worked rather well

Closure conversion is painful

The code is executable version of the proof

Need better type system and more automation to make it practical