Université de Montréal

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:

```
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-Based Verification

Type checking can eliminate run-time errors, of course

It can also provide more guarantees about behavior:

- `iton: Int` $\rightarrow$ `String` guarantees we will not get NULL
- `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

$\Rightarrow$ most of the proof (the boring part) comes for free

Verifying type preservation

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

Use:

e : *SrcExp* τ means:

- e is a Haskell expression that evaluates to an AST
- That AST represents a source expression of type τ

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

Code transformations

Two common code transformations in compilers for functional languages

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.:

$$\tau ::= \mathit{Int} \mid \mathit{Bool} \mid \tau_1 \to \tau_2$$

$$e ::= n \mid \text{if } e_1\ e_2\ e_3 \mid e_1\ e_2 \mid \lambda x \to e \mid x$$

Could be encoded in Haskell as:

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

Annotated source language

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

$$
\begin{array}{l}
\textbf{data}\ \mathit{Exp}\ \tau\ \textbf{where} \\
\quad \mathit{Num} :: \mathit{Int} \to \mathit{Exp}\ \mathit{Int} \\
\quad \mathit{If} :: \mathit{Exp}\ \mathit{Bool} \to \mathit{Exp}\ \tau \to \mathit{Exp}\ \tau \to \mathit{Exp}\ \tau \\
\quad \mathit{App} :: \mathit{Exp}\ (\tau_1 \to \tau_2) \to \mathit{Exp}\ \tau_1 \to \mathit{Exp}\ \tau_2 \\
\quad \ldots \quad \ldots
\end{array}
$$

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

Does it really express a valid typing?

STLC typing rules

Judgment $\boxed{\vdash e : \tau}$ means e has type τ

$$\frac{}{\vdash n : \mathit{Int}} \qquad \frac{\vdash e_1 : \mathit{Bool} \quad \vdash e_2 : \tau \quad \vdash e_3 : \tau}{\vdash \text{if } e_1\ e_3\ e_3 : \tau}$$

$$\frac{\vdash e_1 : \tau_1 \to \tau_2 \quad \vdash e_2 : \tau_1}{\vdash e_1\ e_2 : \tau_2}$$

We can see that

$$e :: \mathit{Exp}\ \tau \quad \simeq \quad \vdash [\![e]\!] : \tau$$

Where $[\![e]\!]$ converts e from Haskell back to its STLC source

Typing rules in Haskell

$$\boxed{\vdash [\![e]\!] : \tau} \qquad \boxed{e :: \mathit{Exp}\ \tau}$$

$$\frac{}{\vdash n : \mathit{Int}} \qquad \mathit{Num} :: \mathit{Int} \to \mathit{Exp}\ \mathit{Int}$$

$$\frac{\vdash e_1 : \mathit{Bool} \quad \vdash e_2 : \tau \quad \vdash e_3 : \tau}{\vdash \mathsf{if}\ e_1\ e_3\ e_3 : \tau} \qquad \mathit{If} :: \mathit{Exp}\ \mathit{Bool} \to \mathit{Exp}\ \tau \to \mathit{Exp}\ \tau \to \mathit{Exp}\ \tau$$

$$\frac{\vdash e_1 : \tau_1 \to \tau_2 \quad \vdash e_2 : \tau_1}{\vdash e_1\ e_2 : \tau_2} \qquad \mathit{App} :: \mathit{Exp}\ (\tau_1 \to \tau_2) \to \mathit{Exp}\ \tau_1 \to \mathit{Exp}\ \tau_2$$

Curry-Howard

Curry-Howard isomorphism states:

$$\textit{Types} \quad \Leftrightarrow \quad \textit{Propositions}$$

$$\textit{Expressions} \quad \Leftrightarrow \quad \textit{Proofs}$$

In our case:

$$e :: \textit{Exp}\,\tau \quad \simeq \quad \vdash [\![e]\!] : \tau$$

Our expressions e of type $\textit{Exp}\,\tau$ play double role:

- Encode syntax of source term
- Encode typing derivation

We say they are *intrinsically typed* Important simplification!

Encoding bindings

Compilers often represent variables as strings or unique integers

For practical typing we need something else

$$\mathit{Lam} :: \mathit{Var}\,\tau_1 \to \mathit{Exp}\,\tau_2 \to \mathit{Exp}\,(\tau_1 \to \tau_2)$$

$$\mathit{Vref} :: \mathit{Var}\,\tau \to \mathit{Exp}\,\tau$$

How do we make sure that a $\mathit{Var}\,\tau$ indeed refers to a var of type τ?

- 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:

$$\frac{\begin{array}{c}\vdash x : \tau_1 \\ \vdots \\ \vdash e : \tau_2\end{array}}{\vdash \lambda x \to e : \tau_1 \to \tau_2}$$

$$\mathit{Lam} :: \begin{array}{l}(\mathit{Exp}\,\tau_1 \to \mathit{Exp}\,\tau_2) \\ \to \mathit{Exp}\,(\tau_1 \to \tau_2)\end{array}$$

No *Vref* case!

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

$$\mathit{Lam}\,(\lambda f \to \mathit{Lam}\,(\lambda x \to \mathit{App}\ f\ x))$$

CPS transformation

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

Functions never return

$$\begin{array}{l} \mathit{inc}\, x = x + 1 \\ \ldots \\ \quad \text{let } z = \mathit{inc}\, a \\ \quad \text{in } \ldots \end{array} \quad \text{becomes} \quad \begin{array}{l} \mathit{inc}\, (x, k) = k\, (x + 1) \\ \ldots \\ \quad \mathit{inc}\, (a, \\ \qquad \lambda z \rightarrow \ldots) \end{array}$$

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

$$
\begin{array}{lcl}
[\![\tau_1 \to \tau_2]\!] & \Rightarrow & ([\![\tau_1]\!], [\![\tau_2]\!] \to \emptyset) \to \emptyset \\
[\![\mathit{Int}]\!] & \Rightarrow & \mathit{Int}
\end{array}
$$

$$
\begin{array}{lcl}
[\![x]\!]\ k & \Rightarrow & k\ x \\
[\![\lambda x \to e]\!]\ k & \Rightarrow & k\ (\lambda(x, k') \to [\![e]\!]\ k') \\
[\![e_1\ e_2]\!]\ k & \Rightarrow & [\![e_1]\!]\ (\lambda x_1 \to [\![e_2]\!]\ (\lambda x_2 \to x_1(x_2, k)))
\end{array}
$$

Type preservation lemma looks like

$$\vdash e : \tau \quad \Rightarrow \quad \vdash [\![e]\!] : ([\![\tau]\!] \to \emptyset) \to \emptyset$$

Representing CPS converted code in Haskell

$$\begin{array}{l}
\textbf{data}\ \mathit{ValK}\ \tau\ \textbf{where} \\
\quad \mathit{NumK} :: \mathit{Int} \to \mathit{ValK}\ \mathit{Int} \\
\quad \mathit{PairK} :: \mathit{ValK}\ \tau_1 \to \mathit{ValK}\ \tau_2 \to \mathit{ValK}\ (\tau_1, \tau_2) \\
\quad \mathit{LamK} :: (\mathit{ValK}\ \tau_1 \to \mathit{ExpK}) \to \mathit{ValK}\ (\tau_1 \to \emptyset) \\
\quad \ldots \quad \ldots \\
\textbf{data}\ \mathit{ExpK}\ \textbf{where} \\
\quad \mathit{IfK} :: \mathit{ValK}\ \mathit{Bool} \to \mathit{ExpK} \to \mathit{ExpK} \to \mathit{ExpK} \\
\quad \mathit{AppK} :: \mathit{ValK}\ (\tau_1 \to \emptyset) \to \mathit{ValK}\ \tau_1 \to \mathit{ExpK} \\
\quad \ldots \quad \ldots
\end{array}$$

CPS type preservation in Haskell

type family $CPS\ \tau$ — Implements $[\![\tau]\!]$

type instance $CPS\ Int = Int$

type instance $CPS\ (\tau_1 \to \tau_2) = (CPS\ \tau_1, (CPS\ \tau_2) \to \emptyset) \to \emptyset$

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

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:

$$\mathit{cps}\ ::\ \mathit{Exp}\ \tau \rightarrow ((\mathit{ValK}\ ((\mathit{CPS}\ \tau) \rightarrow \emptyset)) \rightarrow \mathit{ExpK})$$

$$\mathit{cps}\ (\mathit{App}\ e_1\ e_2)\ k =$$
$$\mathit{cps}\ e_1\ (\lambda v_1 \rightarrow$$
$$\mathit{cps}\ e_2\ (\lambda v_2 \rightarrow$$
$$\mathit{AppK}\ v_1\ (\mathit{PairK}\ v_2\ (\mathit{LamK}\ k))$$

Look ma! No types! Type preservation for free!

System F

STLC is very limited $\Rightarrow$ extend with parametric polymorphism:

$$\tau ::= \ldots \mid \tau_1 \to \tau_2 \mid \forall t.\tau \mid t$$

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

2 new bindings and 1 new kind of variable

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

$$\mathit{TLam} :: (\text{forall}\ t.\mathit{Exp}\ \tau) \to \mathit{Exp}\ (\forall(\lambda t.\tau))$$

$$\mathit{TApp} :: \mathit{Exp}\ (\forall f) \to \mathit{Exp}\ (f\ \tau)$$

But GHC does not support type-level λ !

System F in Haskell

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

data Z; data $S\,\tau$ — To represent natural numbers

data $Var\,\tau$; data $All\,\tau$

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

$$TLam :: (\text{forall}\ t.Exp\,(Subst\,\tau\ t\ Z)) \to Exp\,(All\,\tau)$$

$$TApp :: Exp\,(All\,\tau_1) \to Exp\,(Subst\,\tau_1\ \tau_2\ Z)$$

We still use HOAS for the Λ binding

We have to implement the type-level function *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:

$$\begin{aligned} &subst\_cps\_commute :: TypeRep\,\tau_1 \to TypeRep\,\tau_2 \\ &\quad \to Equiv\,(CPS\,(Subst\,\tau_1\;\tau_2\;n))(Subst\,(CPS\,\tau_1)\,(CPS\,\tau_2)\,n) \end{aligned}$$

If we could use HOAS, we'd get *Subst* and commutativity for free

CPS without CPS

CPS type-level function can be eliminated:

- Keep $\tau_1 \to \tau_2$ as the representation for "normal functions"
- Replace $\tau \to \emptyset$ with new $\mathit{Cont}\,\tau$ for continuations

We lose the equivalence $\tau_1 \to \tau_2 \equiv \mathit{Cont}\,(\tau_1, \mathit{Cont}\,\tau_2)$

This equivalence is not really used

We need two different *LamK* and two different *AppK*

A bit of code duplication, but avoid need for *subst_cps_commute*

Still need to define *Subst* and type-level naturals

CPS with de Bruijn

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

Other typing judgment $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Corresponding Haskell judgment $\boxed{e :: \mathit{Exp}\ \Gamma\ \tau}$

```
data Exp Γ τ where
  Lam  :: Exp (Γ, τ1) τ2 → Exp Γ (τ1 → τ2)
  Vref :: VarIn Γ τ → Exp Γ τ
```

de Bruijn indices

Represent the proof of membership by the index in the set

$$\begin{aligned}&\textbf{data } \mathit{VarIn}\ \Gamma\ \tau \textbf{ where}\\ &\quad \mathit{VarZ} :: \mathit{VarIn}\ (\Gamma, \tau)\ \tau\\ &\quad \mathit{VarS} :: \mathit{VarIn}\ \Gamma\ \tau_1 \to \mathit{VarIn}\ (\Gamma, \tau_2)\ \tau_1\end{aligned}$$

But Γ in source and Γ after CPS are different

Need extra rewriting and corresponding proofs

A lot more work than with HOAS

Had to use it for closure conversion

Conclusion

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