Types in Compilation: Scenes from an Invited Lecture

Mitchell Wand
Northeastern University

June 8, 1997
(revised 11/12/97)

Some Perspectives

Not a play, but some scenes:

- Prologue: Why types?
- Typed vs. Untyped Equivalences
- Type Analysis vs. Flow Analysis
- Analysis-Based Program Transformations

1. Prologue: Why Types?

What are types for?

A type system is a compositional analysis that ensures safety from some class of inappropriate operations.

Examples of Inappropriate Operations

```
(3 x)
(car 3)
((lambda (x y) ...) 6)
((lambda (x) ...) 6 7)

(* #\a #\b)
(+ #\a 1)
(- #\a #\b)

ld 0                  % reference an illegal location
0                     % execute a non-instruction
```

Operations plausible on implementations, but they break abstraction barriers.

Inappropriate operations may or may not be detectable by implementation.

Why typed intermediate languages in compilers for HOT languages?

Source code is typed, low-level abstract machine is typed; so why throw type info away in between?

Compilers for imperative languages have "always" done it:

1. symbol table keeps track of type/size (scalar, array, etc.) for each symbol

2. but the types are largely the same at each level

HOT world has focused on control structures, so haven't thought much about data.

The Uniform Representation Assumption

All values are represented using the same number of bits, with the same alignment constraints. [Clinger]

So only need one implementation for `map`, `append`

A *very* powerful assumption

Pervaded thinking about parametric polymorphism in Lisp/ML world for many years.

But *not* assumed in Ada, C++, etc.

Not always true for Lisps, *e.g.*, fullword data, cdr-coded lists, *ad hoc* polymorphism (*e.g.* Common Lisp).

Liberating HOT languages from URA

Peyton Jones/Launchbury [91], Leroy [92] et al started studying unboxed reps for efficient numerical calculation.

Shao/Appel [94,95], Steckler/Wand [94], Hannan [95], Minamide/Morrissett/Harper [96] studied multiple coexisting closure representations.

Harper/Morrissett [95] introduced “intensional polymorphism” – expressing ad hoc polymorphism using dependent types.

FLINT, TIL, Gallium, etc.

. . . and here we are.

2. Typed vs. Untyped Equivalences

Think about compiler as a sequence of optimizations and translations:

$$\begin{array}{cc}
M_1 \in L_1 & \text{optimize} \\
\downarrow & \text{translate} \\
M_2 \in L_2 & \text{optimize} \\
\downarrow & \text{translate} \\
M_3 \in L_3 & \text{optimize} \\
\vdots &
\end{array}$$

Translations and optimizations must all be "meaning-preserving."

But what does "meaning-preserving" mean?

What's "meaning-preserving"?

Contextual equivalence:

$M \simeq_L N$ iff for all contexts C in L, $C[M]$ and $C[N]$ behave the same.

M and N are *distinguishable* iff they are not contextually equivalent.

$\simeq_L$ is robust wrt definition of behavior.

Types restrict the contexts, and therefore allow more terms to become equivalent.

Untyped λ-Calculus

Define "same behavior" by $M \to^* T = \lambda x \lambda y.x$ iff $N \to^* T$.

Bohm's Theorem: any two distinct normal forms can be distinguished.

Consider the Church numerals:

$$
\begin{aligned}
\lceil 0 \rceil &= \lambda s.\lambda z.z \\
\lceil 1 \rceil &= \lambda s.\lambda z.sz \\
\lceil 2 \rceil &= \lambda s.\lambda z.s(sz) \\
\lceil 3 \rceil &= \lambda s.\lambda z.s(s(sz))
\end{aligned}
$$

$$\lceil n \rceil \approx \lambda s.s^{(n)}$$

By Bohm's theorem, $\lceil m \rceil \simeq \lceil n \rceil$ iff $m = n$.

Well known that there are two terms that work for successor:

$$\begin{array}{l} succ_1 = \lambda n.\lambda s.\lambda z.s(nsz) \\ succ_2 = \lambda n.\lambda s.\lambda z.ns(sz) \end{array}$$

Then for all integers n,

$$(succ_1 \lceil n \rceil) = (succ_2 \lceil n \rceil) = \lceil n + 1 \rceil$$

(Easy induction on n)

But we can also distinguish $succ_1$ and $succ_2$:

$$\begin{array}{l} succ_1(\lambda sz.F)(\lambda x.T)x \to^* T \\ succ_2(\lambda sz.F)(\lambda x.T)x \to^* F \end{array}$$

Are $succ_1$ and $succ_2$ the same or different?

Distinguishing context is "illegal" – $\lambda sz.F$ isn't a numeral.

So untyped LC has *too many* contexts, because it distinguishes terms that we intend to be equivalent.

The Simply-Typed λ-Calculus

Define the Church numerals in STLC by

$$\mathsf{int} = (o \to o) \to (o \to o)$$

$$\begin{aligned}
\lceil 0 \rceil &= \lambda s : (o \to o).\lambda z : o.z \\
\lceil 1 \rceil &= \lambda s : (o \to o).\lambda z : o.sz \\
\lceil 2 \rceil &= \lambda s : (o \to o).\lambda z : o.s(sz) \\
\lceil 3 \rceil &= \lambda s : (o \to o).\lambda z : o.s(s(sz))
\end{aligned}$$

and the booleans by

$$\begin{aligned}
T &= \lambda x : o.\lambda y : o.x \\
F &= \lambda x : o.\lambda y : o.y
\end{aligned}$$

Simply-typed lambda calculus has too *few* contexts.

Proposition 1 *In the STLC,* $\lceil 1 \rceil$ *and* $\lceil 3 \rceil$ *are contextually equivalent: there is no STLC context C such that* $C[\lceil 1 \rceil] \rightarrow^* T$ *and* $C[\lceil 3 \rceil] \rightarrow^* F$*.*

Proof: Consider the "full frame" over the two-element set $\{0, 1\}$. Then the type $o \rightarrow o$ consists of all the functions from $\{0, 1\}$ to itself. There are exactly 4 of these: the constant 0, the constant 1, the identity, and the permuter. And for each of these f, we have $f \circ f \circ f = f$. Hence in this frame, 1 and 3 denote the same thing. Hence, if we had such a C, we'd have $T = C[1] = C[3] = F$, so T and F would co-denote. Contradiction.

System F

System F provides an intermediate number of contexts.

Define the Church numerals by

$$\mathsf{int} = (\forall a)((a \to a) \to (a \to a))$$

$$\begin{aligned}
0 &= \Lambda t.\lambda s : (t \to t).\lambda z : t.z \\
1 &= \Lambda t.\lambda s : (t \to t).\lambda z : t.sz \\
2 &= \Lambda t.\lambda s : (t \to t).\lambda z : t.s(sz) \\
3 &= \Lambda t.\lambda s : (t \to t).\lambda z : t.s(s(sz))
\end{aligned}$$

$$\begin{aligned}
succ_1 &= \lambda n : \mathsf{int}\,.\Lambda t.\lambda s : (t \to t).\lambda z : t.s(nsz) \\
succ_2 &= \lambda n : \mathsf{int}\,.\Lambda t.\lambda s : (t \to t).\lambda z : t.ns(sz)
\end{aligned}$$

$$T = \Lambda t.\lambda x : t.\lambda y : t.x$$

Define behavior by reducibility to T as before.

Again, have $\lceil m \rceil \simeq \lceil n \rceil$ iff $m = n$, but what about $succ_1 \simeq succ_2$?

To show $succ_1 \simeq succ_2$, need some infrastructure:

Theorem 2 *System F is* applicatively extensional*: i.e. if two terms are equivalent in all* applicative *contexts, then they are equivalent in all contexts.*

For int $\rightarrow$ int, this means that M and N are contextually equivalent iff for every closed term P of type int, MP and NP are equivalent terms of type int.

Proof needs coinductive definition of extensional equivalence; then use Howe's method.

Theorem 3 *In System F, $succ_1$ and $succ_2$ are contextually equivalent.*

Proof Sketch:

1. Show that the only closed normal-form terms of type int are the Church numerals (combinatorial argument), and that numerals are equivalent iff they are identical. Hence, by SN, two terms of type int are equivalent iff they reduce to the same numeral.

2. Finally, show that $succ_1$ and $succ_2$ agree on every numeral (trivial induction, as in untyped LC).

A More Realistic Example: Assignment Elimination in Scheme

Scheme variables denote only locations, because a Scheme program can always mutate a variable by performing a set! on it.

Semantics of Scheme automatically allocates heap storage at every procedure call.

Automatically dereferences each variable reference.

Molto inefficient!

Assignment Elimination

Solution: translate into intermediate language with explicit (ML-like) storage model (Scheme$_{cell}$).

Don't allocate heap storage for variables that are demonstrably never mutated.

Difficult to prove correct, because it radically changes the store behavior of the program.

Example

$$
\begin{array}{l}
\text{(lambda } (x\ y) \\
\quad \ldots \text{(set! } x\ 5) \\
\quad \ldots (f\ x\ y) \ldots)
\end{array}
$$

is converted to

$$
\begin{array}{l}
\text{(lambda } (x\ y) \\
\quad \text{(let } ((x\ \text{(make-cell } x)) \\
\qquad\quad (y\ y)) \\
\quad \ldots \text{(cell-set! } x\ 5) \\
\quad \ldots (f\ \text{(cell-deref } x)\ y) \\
\quad \ldots))
\end{array}
$$

How to prove the correctness of transformation?

Give semantics to Scheme and Scheme$_{cell}$ by translation into a common metalanguage ml, and show that translation preserves meaning:

$$
\begin{array}{ccc}
\text{Scheme} & \xrightarrow{AE} & \text{Scheme}_{cell} \\
[\![-]\!]_{scm}\downarrow & & \downarrow[\![-]\!]_{cell} \\
ml & = & ml
\end{array}
$$

where the equality at the bottom denotes contextual equivalence, $\simeq_{ml}$ in the metalanguage ml.

The Metalanguage ml

Metalanguage ml is process calculus based on λ-calculus

Extended by adding cps store and input-output operations

Behavior is input-output equivalence

Have applicative extensionality theorem (even without types)

Semantics is cps translation

But . . .

But . . .

Untyped metalanguage is too rich: can distinguish $[\![((\text{lambda}\,(x)\,x)\ e)]\!]_{scm}$ and $[\![e]\!]_{scm}$.

Distinguishing context used “metacontinuations.”

Solution: Give ml a type structure (with sums, products, and recursive types).

Then all goes through.

3. Type Analysis vs. Flow Analysis

Types are:	Flows are:
Inductive	Co-inductive
Local	Global
Denotational	Operational

But this distinction may not hold up under closer scrutiny.

A simple framework for flow analysis

Deals with a language of labelled terms.

Definition 1 *A* flow Φ *is a map from* $\mathsf{Lab} \cup \mathsf{Var}$ *to* $P(\mathsf{Lab}_\lambda)$*, where* Lab_λ *is the set of labels of* λ*-abstractions.*

$\Phi(u)$ describes labels of the possible values of u.

Must define $\models$: a relation that determines when a labelled term e^l is consistent with a flow Φ (or when Φ is an admissible description of e^l).

0CFA (a la [Nielson-Nielson 97])

Simplest version of $\models$ (due to Jones, Sestoft, Shivers, *et al.*):

$$(\lambda x.{e_1}^{l_1})^l \models \Phi \iff l \in \Phi(l)$$

$$\begin{array}{lll}
({e_1}^{l_1}\ {e_2}^{l_2})^l \models \Phi & \iff & {e_1}^{l_1} \models \Phi \\
 & & \wedge\ {e_2}^{l_2} \models \Phi \\
 & & \wedge\ \forall l' \quad l' \in \Phi(l_1) \\
 & & \qquad\quad \wedge\ l' \text{ labels } (\lambda x.{e_3}^{l_3}) \\
 & & \qquad\ \Rightarrow \\
 & & \qquad\quad {e_3}^{l_3} \models \Phi \\
 & & \qquad\quad \wedge\ \Phi(l_2) \subseteq \Phi(x) \\
 & & \qquad\quad \wedge\ \Phi(l_3) \subseteq \Phi(l)
\end{array}$$

This is global: it refers to the global labelling of the term.

Fixed Points Everywhere

Definition of $\models$ is recursive: so what fixed point do we want?

For any term, want as many admissible flows as possible (so long as they don't lie), so that more accurate flows are admitted.

Hence take gfp interpretation of definition of $\models$.

Can show $\models$ is preserved under computation (like subject reduction).

Job of analyzer is to find *smallest* admissible Φ (via lfp).

Type Reconstruction

Type reconstruction system generates a set of constraints among type variables.

Start with labelled terms and use the labels and variables to index type variables (*e.g.* τ_l, τ_x).

$$\frac{{e_1}^{l_1} \triangleright C_1}{(\lambda x.{e_1}^{l_1})^l \triangleright C_1 \cup \{(\tau_x \to \tau_{l_1}) \sqsubseteq \tau_l\}}$$

$$\frac{{e_1}^{l_1} \triangleright C_1 \quad {e_2}^{l_2} \triangleright C_2}{({e_1}^{l_1}\ {e_2}^{l_2})^l \triangleright C_1 \cup C_2 \cup \{\tau_{l_1} \sqsubseteq (\tau_{l_2} \to \tau_l)\}}$$

If $e^l \triangleright C$, then solutions to C correspond to the type derivations for e.

This works with or without subtyping.

From type reconstruction to flow analysis

Define $C \models t \sqsubseteq t'$ iff $t \sqsubseteq t'$ is true in any solution to C.

Theorem 4 *Let* $e^l \triangleright C$. *Define*

$$\Phi(u) = \{l \mid l \in \mathsf{Lab}_\lambda \wedge (C \models \tau_l \sqsubseteq \tau u)\}$$

Then $e^l \models \Phi$.

Proof by coinduction on definition of $\models$; interesting case is

$$({e_1}^{l_1}\ {e_2}^{l_2})^{l} \models \Phi$$

If $l' \in \Phi(l_1)$ and l' labels $(\lambda x.{e_3}^{l_3})$, then C entails:

$$\begin{array}{rll} (\tau_x \to \tau_{l_3}) & \sqsubseteq \tau_{l'} & \text{type rule for } \lambda \\ & \sqsubseteq \tau_{l_1} & l' \in \Phi(l_1) \\ & \sqsubseteq (\tau_{l_2} \to \tau_l) & \text{type rule for app} \end{array}$$

Hence

$$\begin{array}{l} C \models \tau_{l_2} \sqsubseteq \tau_x \\ C \models \tau_{l_3} \sqsubseteq \tau_l \end{array}$$

and therefore

$$\begin{array}{l} \Phi(l_2) \subseteq \Phi(x) \\ \Phi(l_3) \subseteq \Phi(l) \end{array}$$

as required.

What's going on here?

In type reconstruction, information must propagate from one place in the program to another to make deductions.

This is just flow analysis!

Can do same thing replacing $\sqsubseteq$ by $=$; then get cruder flow analysis in almost-linear time.

[Heintze 96] shows that this can be done without loss of accuracy.

This is an old idea

1. [Wand 86], [Johnson-Walz 86] — for locating type errors (with $=$)

2. [Baker 90] – for alias/sharing analysis

3. [Palsberg-O'Keefe 95], [Heintze 95] – showed equivalence with Amadio-Cardelli recursive types with $\top$ and $\bot$.

Question: how to extend this story to polymorphic types (on the type side) and polyvariant analyses (on the analysis side)?

What about "sub-zero" CFA (*a la* Ashley and Dybvig)?

4. Analysis-Based Program Transformation

Type-based equational deductions are congruences

Therefore sound for contextual equivalence

Show that $\Gamma \vdash e_1 = e_2 : t$, then can replace e_1 by e_2 in any context.

Takes advantage of the *denotational* nature of type systems.

Flow-based transformations rely on operational techniques

Original program goes through a sequence of configurations

$$C_0 \to C_1 \to C_2 \to \ldots \to C_n$$

Analyze program, then transform.

Transformed program goes through sequence

$$C_0^* = D_0 \to D_1 \to D_2 \to \ldots \to D_n$$

Would like to show that if C_0 is any starting configuration, then $C_n = D_n$.

Want a relation $\mathcal{R}$ on configurations such that

$$(\forall i \geq 0)(C_i \; \mathcal{R} \; D_i)$$

[need $\mathcal{R}$ to be true on initial configs, to be equality on terminal configs, etc, but that's easy]

How to show $(\forall i \geq 0)(C_i \mathrel{\mathcal{R}} D_i)$?

Would like to show $\mathcal{R}$ preserved under computation:

$$(C \mathrel{\mathcal{R}} D) \wedge (C \rightarrow C') \wedge (D \rightarrow D') \Rightarrow (C' \mathrel{\mathcal{R}} D')$$

But usually it isn't!

Fails just when you hit the transformed part of the program.

That's where the analysis is used!

Instead, find a formula P such that both

$$(C \text{ reachable}) \Rightarrow (C \models P) \tag{1}$$

$$\begin{array}{l}(C \mathrel{\mathcal{R}} D) \wedge \ (C \to C') \wedge \ (D \to D') \\ \boxed{\wedge \ (C \models P)} \\ \Rightarrow (C' \mathrel{\mathcal{R}} D')\end{array} \tag{2}$$

(1) expresses the *soundness* of the analysis.

(2) expresses the *correctness* of the transformation as justified by the analysis.

Story for big-step semantics is similar.

How to find such P?

1. Type-soundness!

2. Flow analysis!

It appears flow analysis can get finer information than type analysis, at the expense of more complex analysis and of more complex proofs of transformation.

Challenge: can we understand this tradeoff better, either theoretically or pragmatically? What is a fair comparison?

Challenge: can we find some unifying scheme that would enable us to get the best of both worlds?

You could find it on the Web

http://www.cs.bc.edu/~muller/TIC97/

http://www.ccs.neu.edu/home/wand

ftp://ftp.ccs.neu.edu/pub/people/wand
/papers/tic-97.ps