CSE 505: Concepts of Programming Languages

Frances Perry pretending to be Dan Grossman

Fall 2009

Lecture 9— More STλC Extensions and Related Topics

5 years of my life in 2 slides

Thesis topic: Proving fault-tolerance properties of assembly programs using typed assembly languages.

- Cosmic rays flip bits resulting in errors.
- One solution – duplicate all computation and check for consistency before making any permanent changes.
- Compilers are tricky beasts. Wouldn't it be nice to know that your program is really redudant?
- Use a assembly-language type system to prove that values are duplicated and always checked when needed.

Sexy Job Market Spiel: cosmic rays, random bit flips, millions of dollars lost, provably secure solution, flashy logo, ...

In Reality: lots and lots of type safety proofs

5 years of my life in 2 slides

1. Define an operational semantics to describe how a abstract machine executes (heap, stack, registers, etc).
2. Figure out which states are "bad" and what good invariants prevent badness.
3. Define a type system to track invariants are maintained.
4. Prove the type system is sound with respect to #1 and #2 using Progress and Preservation (with our good friends the substitution lemma, canonical forms, etc, etc).
5. Provide a translation from a well typed source language into the assembly language to show the type system isn't too restrictive.
6. Rinse and repeat 3 times to generate 150 pages of prose, 100 pages of judgments, 220 pages of ascii proofs, and 1 Phd.

Outline

- Continue extending STλC– booleans and conditionals, data structures (pairs, records, sums), recursion
- Discussion of “anonymous” types
- Consider termination informally
- Next time (two extended digressions): Curry-Howard Isomorphism, Evaluation Contexts, Abstract Machines, Continuations

Extending ST$\boldsymbol{\lambda}$C

- Extend Syntax: $\boldsymbol{e}$, $\boldsymbol{v}$, $\boldsymbol{\tau}$, $\boldsymbol{\ldots}$
- Extend Operational Semantics: $\boldsymbol{e \rightarrow e}$
- Extend Typing Rules: $\boldsymbol{\Gamma \vdash e : \tau}$
- Extend Proofs: Progress, Preservation, Canonical Forms, Substitution

STλC Review

$$e ::= \lambda x.\ e \mid x \mid e\ e \mid c \qquad v ::= \lambda x.\ e \mid c$$

$$\tau ::= \mathsf{int} \mid \tau \to \tau \qquad \Gamma ::= \cdot \mid \Gamma, x : \tau$$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

$e[e'/x]$: capture-avoiding substitution of e' for free x in e

$$\frac{}{\Gamma \vdash c : \mathsf{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

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

Type Safety Proof Hierarchy

Safety: Well-typed programs never get stuck.

By induction on the number of steps.

- **Progress:** Well-typed programs are done or can take a step.
  If $\cdot \vdash e : \tau$, then e is a value or $\exists\ e'$ such that $e \rightarrow e'$.
  By induction on $\Gamma \vdash e : \tau$
  - **Canonical Forms:** If it's a duck, then it has feathers.
    By inspection of values.
- **Preservation:** Making progress preserves the type.
  If $\cdot \vdash e : \tau$ and $e \rightarrow e'$, then $\cdot \vdash e' : \tau$.
  By induction on $\Gamma \vdash e : \tau$
  - **Substitution:** Things stay well-typed after stapple-gunning.
    By induction on $\Gamma, x : \tau' \vdash e_1 : \tau$
    * **Exchange:** Reordering scoping is ok.
    * **Weakening:** It's ok to drop unused variables on the floor.

Booleans and Conditionals

$$e \;::=\; \ldots \mid \textsf{true} \mid \textsf{false} \mid \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3$$

$$\tau \;::=\; \ldots \mid \textsf{bool} \qquad v \;::=\; \ldots \mid \textsf{true} \mid \textsf{false}$$

$$\frac{e_1 \rightarrow e_1'}{\textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 \rightarrow \textsf{if } e_1' \textsf{ then } e_2 \textsf{ else } e_3}$$

$$\frac{}{\textsf{if true then } e_2 \textsf{ else } e_3 \rightarrow e_2} \qquad \frac{}{\textsf{if false then } e_2 \textsf{ else } e_3 \rightarrow e_3}$$

$$\frac{\Gamma \vdash e_1 : \textsf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{if } e_1 \textsf{ then } e_2 \textsf{ else } e_3 : \tau}$$

$$\frac{}{\Gamma \vdash \textsf{true} : \textsf{bool}} \qquad \frac{}{\Gamma \vdash \textsf{false} : \textsf{bool}}$$

Notes: new Canonical Forms case, all lemma cases easy

(Also need to extend definition of substitution (will stop writing that)...)

Pairs (CBV, left-right)

$$
\begin{array}{rcl}
e & ::= & \ldots \mid (e, e) \mid e.1 \mid e.2 \\
v & ::= & \ldots \mid (v, v) \\
\tau & ::= & \ldots \mid \tau * \tau
\end{array}
$$

$$\frac{e_1 \rightarrow e_1'}{(e_1, e_2) \rightarrow (e_1', e_2)} \qquad \frac{e_2 \rightarrow e_2'}{(v_1, e_2) \rightarrow (v_1, e_2')}$$

$$\frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \qquad \frac{e \rightarrow e'}{e.2 \rightarrow e'.2}$$

$$\frac{}{(v_1, v_2).1 \rightarrow v_1} \qquad \frac{}{(v_1, v_2).2 \rightarrow v_2}$$

Small-step can be a pain (more concise notation next lecture)

Pairs continued

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

Canonical Forms: If $\cdot \vdash v : \tau_1 * \tau_2$, then v has the form (v_1, v_2).

Progress: New cases using C.F. are $v.1$ and $v.2$.

Preservation: For primitive reductions, inversion gives the result *directly*.

Records

Records seem like pairs with *named fields*

$$\begin{array}{lcl} e & ::= & \ldots \mid \{l_1 = e_1; \ldots; l_n = e_n\} \mid e.l \\ \tau & ::= & \ldots \mid \{l_1 : \tau_1; \ldots; l_n : \tau_n\} \\ v & ::= & \ldots \mid \{l_1 = v_1; \ldots; l_n = v_n\} \end{array}$$

Fields do *not* α-convert.

Names might let us reorder fields, e.g.,
$\cdot \vdash \{l_1 = 42; l_2 = \mathbf{true}\} : \{l_2 : \mathbf{bool}; l_1 : \mathbf{int}\}$.

Nothing wrong with this, but many languages disallow it. (Why? Run-time efficiency and/or type inference)

More on this when we study *subtyping*

Sums

What about ML-style datatypes:

`type t = A | B of int | C of int*t`

1. Tagged variants (i.e., discriminated unions)
2. Recursive types
3. Type constructors (e.g., `type 'a mylist = ...`)
4. Names the type

Today we'll model just (1) with (anonymous) sum types...

Sum syntax and overview

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \mathsf{A}(e) \mid \mathsf{B}(e) \mid \textbf{match } e \textbf{ with A}x.\ e \mid \mathsf{B}x.\ e \\
v & ::= & \ldots \mid \mathsf{A}(v) \mid \mathsf{B}(v) \\
\tau & ::= & \ldots \mid \tau_1 + \tau_2
\end{array}
$$

- Only two constructors: **A** and **B**
- All values of any sum type built from these constructors
- So $\mathsf{A}(e)$ can have any sum type allowed by e's type
- No need to declare sum types in advance
- Like functions, will "guess the type" in our rules

Sum semantics

$$\frac{}{\textsf{match } \mathsf{A}(v) \textsf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \rightarrow e_1[v/x]}$$

$$\frac{}{\textsf{match } \mathsf{B}(v) \textsf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \rightarrow e_2[v/y]}$$

$$\frac{e \rightarrow e'}{\mathsf{A}(e) \rightarrow \mathsf{A}(e')} \qquad \frac{e \rightarrow e'}{\mathsf{B}(e) \rightarrow \mathsf{B}(e')}$$

$$\frac{e \rightarrow e'}{\textsf{match } e \textsf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \rightarrow \textsf{match } e' \textsf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2}$$

match has binding occurrences, just like pattern-matching.

(Definition of substitution must avoid capture, just like functions.)

What is going on

Feel free to think about *tagged values* in your head:

- A tagged value is a pair of a tag (A or B, or 0 or 1 if you prefer) and the value
- A match checks the tag and binds the variable to the value

This much is just like Caml in lecture 1 and related to homework 2.

Sums in other guises:

- C: use an `enum` and a `union`
  - More space than ML, but supports in-place mutation
- OOP: use an abstract superclass and subclasses

Sum Type-checking

Inference version (not trivial to infer; can require annotations)

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x{:}\tau_1 \vdash e_1 : \tau \qquad \Gamma, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \textsf{match } e \textsf{ with A}x.\ e_1 \mid \textsf{B}y.\ e_2 : \tau}$$

Key ideas:

- For constructor-uses, "other side can be anything"
- For match, both sides need same type since don't know which branch will be taken, just like an if.

Can encode booleans with sums. E.g., **bool** $=$ **int** $+$ **int**, **true** $=$ $\mathsf{A}(0)$, **false** $=$ $\mathsf{B}(0)$.

Type Safety

Canonical Forms: If $\cdot \vdash v : \tau_1 + \tau_2$, then there exists a v_1 such that either v is $\mathsf{A}(v_1)$ and $\cdot \vdash v_1 : \tau_1$ or v is $\mathsf{B}(v_1)$ and $\cdot \vdash v_1 : \tau_2$.

The rest is induction and substitution...

Pairs vs. sums

- You need both in your language
  - With only pairs, you clumsily use dummy values, waste space, and rely on unchecked tagging conventions
  - Example: replace $\mathbf{int} + (\mathbf{int} \rightarrow \mathbf{int})$ with $\mathbf{int} * (\mathbf{int} * (\mathbf{int} \rightarrow \mathbf{int}))$
- "logical duals" (as we'll see soon and the typing rules show)
  - To make a $\tau_1 * \tau_2$ you need a τ_1 and a τ_2.
  - To make a $\tau_1 + \tau_2$ you need a τ_1 or a τ_2.
  - Given a $\tau_1 * \tau_2$, you can get a τ_1 or a τ_2 (or both; your "choice").
  - Given a $\tau_1 + \tau_2$, you must be prepared for either a τ_1 or τ_2 (the value's "choice").

Base Types, in general

What about floats, strings, enums, ...? Could add them all or do something more general...

Parameterize our language/semantics by a collection of *base types* $(b_1, \ldots, b_n)$ and *primitives* $(c_1 : \tau_1, \ldots, c_n : \tau_n)$.

Examples: concat : string→string→string
toInt : float→int
"hello" : string

For each primitive, *assume* if applied to values of the right types it produces a value of the right type.

Together the types and assumed steps tell us how to type-check and evaluate $c_i\ v_1 \ldots v_n$ where c_i is a primitive.

We can prove soundness *once and for all* given the assumptions.

Recursion

We won't prove it, but every extension so far preserves termination. A Turing-complete language needs some sort of loop. What we add won't be encodable in STλC.

$$e ::= \ldots \mid \textsf{fix}\ e$$

$$\frac{e \rightarrow e'}{\textsf{fix}\ e \rightarrow \textsf{fix}\ e'} \qquad \frac{}{\textsf{fix}\ \lambda x.\ e \rightarrow e[\textsf{fix}\ \lambda x.\ e/x]}$$

Using fix

It works just like `let rec`, e.g.,

$$\textbf{fix } \lambda f.\ \lambda n.\ \textbf{if } n < 1 \textbf{ then } 1 \textbf{ else } n * (f(n-1))$$

Note: You can use it for mutual recursion too.

Pseudo-math digression

Why is it called fix? In math, a fixed-point of a function g is an x such that $g(x) = x$.

Let g be $\lambda f.\ \lambda n.$ **if** $n < 1$ **then** 1 **else** $n * (f(n-1))$.

If g is applied to a function that computes factorial for arguments $\leq m$, then g returns a function that computes factorial for arguments $\leq m + 1$.

Now g has type $(\textbf{int} \rightarrow \textbf{int}) \rightarrow (\textbf{int} \rightarrow \textbf{int})$. The fix-point of g is the function that computes factorial for *all* natural numbers.

And **fix** g is equivalent to that function. That is, **fix** g is the fix-point of g.

Typing fix

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \textbf{fix}\ e : \tau}$$

Math explanation: If e is a function from τ to τ, then **fix** e, the fixed-point of e, is some τ with the fixed-point property. So it's something with type τ.

Operational explanation: **fix** $\lambda x.\ e'$ becomes $e'[$**fix** $\lambda x.\ e'/x]$. The substitution means x and **fix** $\lambda x.\ e'$ better have the same type. And the result means e' and **fix** $\lambda x.\ e'$ better have the same type.

Note: The τ in the typing rule is usually instantiated with a function type e.g., $\tau_1 \to \tau_2$, so e has type $(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)$.

Note: Proving soundness is straightforward!

General approach

We added lets, booleans, pairs, records, sums, and fix. Let was syntactic sugar. Fix made us Turing-complete by "baking in" self-application. The others *added types*.

Whenever we add a new form of type τ there are:

- Introduction forms (ways to make values of type τ)
- Elimination forms (ways to use values of type τ)

What are these forms for functions? Pairs? Sums?

When you add a new type, think "what are the intro and elim forms"?

Anonymity

We added many forms of types, all *unnamed* a.k.a. *structural*.

Many real PLs have (all or mostly) *named* types:

- Java, C, C++: all record types (or similar) have names (omitting them just means compiler makes up a name)
- Caml sum-types have names.

A never-ending debate:

- Structural types allow more code reuse, which is good.
- Named types allow less code reuse, which is good.
- Structural types allow generic type-based code, which is good.
- Named types allow type-based code to distinguish names, which is good.

The theory is often easier and simpler with structural types.

Termination

Surprising fact: If $\cdot \vdash e : \tau$ in the STλC with all our additions *except* fix, then there exists a v such that $e \rightarrow^* v$.

That is, all programs terminate.

So termination is trivially decidable (the constant "yes" function), so our language is not Turing-complete.

Proof is in the book. It requires cleverness because the size of expressions does *not* "go down" as programs run.

Non-proof: Recursion in λ calculus requires some sort of self-application. Easy fact: For all Γ, x, and τ, we *cannot* derive $\Gamma \vdash x\ x : \tau$.