CSE 505: Concepts of Programming Languages

Frances Perry pretending to be Dan Grossman
Fall 2009
Lecture 9— More ST\text{AC} Extensions and Related Topics
5 years of my life in 2 slides


- 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 redundant?
- 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 STLαC– 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\textit{\textlambda}C

- Extend Syntax: \( e, v, \tau, \ldots \)

- Extend Operational Semantics: \( e \rightarrow e \)

- Extend Typing Rules: \( \Gamma \vdash e : \tau \)

- Extend Proofs: Progress, Preservation, Canonical Forms, Substitution
STλC Review

\[ e ::= \lambda x. \, e \mid x \mid e \, e \mid c \quad v ::= \lambda x. \, e \mid c \]

\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \quad \Gamma ::= \cdot \mid \Gamma, x : \tau \]

\[
\begin{align*}
(\lambda x. \, e) \, v & \rightarrow e[v/x] \\
e_1 \rightarrow e'_1 & \quad e_1 \, e_2 \rightarrow e'_1 \, e_2 \\
e_2 \rightarrow e'_2 & \quad v \, e_2 \rightarrow v \, e'_2
\end{align*}
\]

\[ e[e'/x] : \text{capture-avoiding substitution of } e' \text{ for free } x \text{ in } e \]

\[
\begin{align*}
\Gamma \vdash c : \text{int} & \\
\Gamma \vdash x : \Gamma(x) & \\
\Gamma, x : \tau_1 \vdash e : \tau_2 & \\
\Gamma, x : \tau_1 \vdash e_1 : \tau_2 & \\
\Gamma, x : \tau_1 \vdash e_2 : \tau_2 & \\
\Gamma \vdash e_1 \, e_2 : \tau_1
\end{align*}
\]
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

\[ e ::= \ldots | \text{true} | \text{false} | \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

\[ \tau ::= \ldots | \text{bool} \quad v ::= \ldots | \text{true} | \text{false} \]

\[
\frac{e_1 \rightarrow e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3}
\]

\[
\frac{\text{if true then } e_2 \text{ else } e_3 \rightarrow e_2}{\text{if false then } e_2 \text{ else } e_3 \rightarrow e_3}
\]

\[
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}
\]

\[
\frac{\Gamma \vdash \text{true} : \text{bool}}{\Gamma \vdash \text{false} : \text{bool}}
\]

Notes: new Canonical Forms case, all lemma cases easy
(Also need to extend definition of substitution (will stop writing that)...)
Pairs (CBV, left-right)

\[ e ::= \ldots \mid (e, e) \mid e.1 \mid e.2 \]

\[ v ::= \ldots \mid (v, v) \]

\[ \tau ::= \ldots \mid \tau \ast \tau \]

\[ e_1 \rightarrow e'_1 \]
\[ (e_1, e_2) \rightarrow (e'_1, e_2) \]

\[ e_2 \rightarrow e'_2 \]
\[ (v_1, e_2) \rightarrow (v_1, e'_2) \]

\[ e \rightarrow e' \]
\[ e.1 \rightarrow e'.1 \]

\[ e.2 \rightarrow e'.2 \]

\[ (v_1, v_2).1 \rightarrow v_1 \]
\[ (v_1, v_2).2 \rightarrow v_2 \]

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

\[
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2}
\]

\[
\frac{\Gamma \vdash e : \tau_1 \ast \tau_2}{\Gamma \vdash e.1 : \tau_1}
\quad \frac{\Gamma \vdash e : \tau_1 \ast \tau_2}{\Gamma \vdash e.2 : \tau_2}
\]

Canonical Forms: If \( \cdot \vdash v : \tau_1 \ast \tau_2 \), 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 \textit{directly}. 

Dan Grossman  
CSE505 Fall 2009, Lecture 9
Records

Records seem like pairs with *named fields*

\[
\begin{align*}
e & ::= \ldots | \{ l_1 = e_1 ; \ldots ; l_n = e_n \} | e.l \\
\tau & ::= \ldots | \{ l_1 : \tau_1 ; \ldots ; l_n : \tau_n \} \\
v & ::= \ldots | \{ l_1 = v_1 ; \ldots ; l_n = v_n \}
\end{align*}
\]

Fields do *not* $\alpha$-convert.

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

*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:

\[
\text{type } t = A \mid B \text{ of } \text{int} \mid C \text{ of } \text{int}*t
\]

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

\[ e ::= \ldots | A(e) | B(e) | \text{match } e \text{ with } A x. \ e | B x. \ e \]
\[ v ::= \ldots | A(v) | B(v) \]
\[ \tau ::= \ldots | \tau_1 + \tau_2 \]

- Only two constructors: \( A \) and \( B \)
- All values of any sum type built from these constructors
- So \( A(e) \) 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

match $A(v)$ with $Ax. \ e_1 \ | \ By. \ e_2 \rightarrow e_1[v/x]$ 

match $B(v)$ with $Ax. \ e_1 \ | \ By. \ e_2 \rightarrow e_2[v/y]$ 

$$
\begin{align*}
 & e \rightarrow e' \\
\frac{}{A(e) \rightarrow A(e')} & e \rightarrow e' \\
\frac{}{B(e) \rightarrow B(e')} & e \rightarrow e'
\end{align*}
$$

match $e$ with $Ax. \ e_1 \ | \ By. \ e_2 \rightarrow \text{match } e' \text{ with } Ax. \ e_1 \ | \ By. \ e_2$

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)

\[
\begin{align*}
\Gamma & \vdash e : \tau_1 \\
\Gamma & \vdash A(e) : \tau_1 + \tau_2 \\
\Gamma & \vdash e : \tau_2 \\
\Gamma & \vdash B(e) : \tau_1 + \tau_2 \\
\Gamma & \vdash e : \tau_1 + \tau_2 \\
\Gamma, x:\tau_1 & \vdash e_1 : \tau \\
\Gamma, y:\tau_2 & \vdash e_2 : \tau \\
\Gamma & \vdash \text{match } e \text{ with } A x. e_1 \mid B y. e_2 : \tau
\end{align*}
\]

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 = A(0), false = 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 $A(v_1)$ and $\cdot \vdash v_1 : \tau_1$ or $v$ is $B(v_1)$ 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 int + (int → int) with int * (int * (int → int))
- “logical duals” (as we’ll see soon and the typing rules show)
  - To make a τ₁ * τ₂ you need a τ₁ and a τ₂.
  - To make a τ₁ + τ₂ you need a τ₁ or a τ₂.
  - Given a τ₁ * τ₂, you can get a τ₁ or a τ₂ (or both; your “choice”).
  - Given a τ₁ + τ₂, you must be prepared for either a τ₁ or τ₂ (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 $ST\lambda C$.

$$
\begin{align*}
e &::= \ldots \mid \text{fix } e \\
\text{fix } e &\rightarrow \text{fix } e' \\
\text{fix } \lambda x. \ e &\rightarrow e[\text{fix } \lambda x. \ e/x]
\end{align*}
$$
Using fix

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

\[ \text{fix } \lambda f. \lambda n. \text{ if } n < 1 \text{ then } 1 \text{ else } n \ast (f(n - 1)) \]

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 $\lambda f. \lambda n. \text{if } n < 1 \text{ then } 1 \text{ else } n \ast (f(n - 1))$.

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

And $\text{fix } g$ is equivalent to that function. That is, $\text{fix } g$ is the fix-point of $g$. 
Typing fix

\[
\Gamma \vdash e : \tau \rightarrow \tau \\
\frac{}{\Gamma \vdash \text{fix } e : \tau}
\]

Math explanation: If \( e \) is a function from \( \tau \) to \( \tau \), then \( \text{fix } e \), the fixed-point of \( e \), is some \( \tau \) with the fixed-point property. So it's something with type \( \tau \).

Operational explanation: \( \text{fix } \lambda x. e' \) becomes \( e'[\text{fix } \lambda x. e'/x] \). The substitution means \( x \) and \( \text{fix } \lambda x. e' \) better have the same type. And the result means \( e' \) and \( \text{fix } \lambda x. e' \) better have the same type.

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

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 $\tau$ there are:

- Introduction forms (ways to make values of type $\tau$)
- Elimination forms (ways to use values of type $\tau$)

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 ST\(\lambda\)C 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 \(\lambda\) calculus requires some sort of self-application. Easy fact: For all $\Gamma$, $x$, and $\tau$, we cannot derive $\Gamma \vdash x \; x : \tau$. 