CSE 505: Concepts of Programming Languages

Dan Grossman
Fall 2008
Lecture 9— More ST\&C Extensions and Related Topics
Outline

- Continue extending ST\(\lambda\)C – data structures, recursion
- Discussion of “anonymous” types
- Consider termination informally
- Next time (a break from types): Curry-Howard Isomorphism, Evaluation Contexts, Abstract Machines, Continuations
Review

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

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

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

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

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

\[
\Gamma \vdash e_1\ e_2 : \tau_1
\]

Preservation: If \( \cdot \vdash e : \tau \) and \( e \rightarrow e' \), then \( \cdot \vdash e' : \tau \).

Progress: If \( \cdot \vdash e : \tau \), then e is a value or \( \exists e' \) such that \( e \rightarrow e' \).
Booleans and Conditionals

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

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

\[ e_1 \rightarrow e'_1 \]

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

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

\[ \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \]

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

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

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

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

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

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

\[
\begin{align*}
e_1 \to e'_1 \\
(e_1, e_2) \to (e'_1, e_2)
\end{align*}
\]

\[
\begin{align*}
e_2 \to e'_2 \\
(v_1, e_2) \to (v_1, e'_2)
\end{align*}
\]

\[
\begin{align*}
e \to e' \\
e.1 \to e'.1
\end{align*}
\]

\[
\begin{align*}
e \to e' \\
e.2 \to e'.2
\end{align*}
\]

\[
\begin{align*}
(v_1, v_2).1 \to v_1 \\
(v_1, v_2).2 \to v_2
\end{align*}
\]

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

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \]
\[ \Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2 \]
\[ \Gamma \vdash e : \tau_1 \ast \tau_2 \]
\[ \Gamma \vdash e.1 : \tau_1 \]
\[ \Gamma \vdash e.2 : \tau_2 \]

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}. 
Records

Records seem like pairs with *named fields*

\[
e ::= \ldots \mid \{l_1 = e_1; \ldots; l_n = e_n\} \mid e.l
\]

\[
\tau ::= \ldots \mid \{l_1 : \tau_1; \ldots; l_n : \tau_n\}
\]

\[
v ::= \ldots \mid \{l_1 = v_1; \ldots; l_n = v_n\}
\]

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)

(Caml has only *named* record types with *disjoint* fields.)

More on this when we study *subtyping*
Sums

What about ML-style datatypes:

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

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

\[
e ::= \ldots | A(e) | B(e) | \text{match } e \text{ with } A x. \ e | B x. \ e
\]

\[
v ::= \ldots | A(v) | B(v)
\]

\[
\tau ::= \ldots | \tau_1 + \tau_2
\]

• 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

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

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

\[
\frac{e \rightarrow e'}{A(e) \rightarrow A(e')}
\]

\[
\frac{e \rightarrow e'}{B(e) \rightarrow B(e')}
\]

\[
\frac{e \rightarrow e'}{\text{match } e \text{ with } A x. \ e_1 \mid B y. \ e_2 \rightarrow \text{match } e' \text{ with } A x. \ e_1 \mid B y. \ e_2}
\]

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)

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

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., \texttt{bool} = \texttt{int} + \texttt{int}, \texttt{true} = A(0), \texttt{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 \texttt{int + (int \rightarrow int)} with \texttt{int * (int * (int \rightarrow int))}

- "logical duals" (as we’ll see soon and the typing rules show)
  - To make a \( \tau_1 \ast \tau_2 \) you need a \( \tau_1 \) and a \( \tau_2 \).
  - To make a \( \tau_1 + \tau_2 \) you need a \( \tau_1 \) or a \( \tau_2 \).
  - Given a \( \tau_1 \ast \tau_2 \), you can get a \( \tau_1 \) or a \( \tau_2 \) (or both; your “choice”).
  - Given a \( \tau_1 + \tau_2 \), you must be prepared for either a \( \tau_1 \) or \( \tau_2 \) (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: \(\text{concat} : \text{string} \rightarrow \text{string} \rightarrow \text{string}\)
\(\text{toInt} : \text{float} \rightarrow \text{int}\)
\(\text{“hello”} : \text{string}\)

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.

E.g., \texttt{let rec f x = e}

Do typed recursive functions need to be bound to variables or can they be anonymous?

In Caml, you need variables, but it’s unnecessary:

\[
e ::= \ldots \mid \text{fix } e
\]

\[
e \rightarrow e' \\
\text{fix } e \rightarrow \text{fix } e' \\
\text{fix } \lambda x. e \rightarrow e[\text{fix } \lambda x. e/x]
\]
Using fix

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

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

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

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

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 insantiated 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 $\Gamma \vdash x \ x : \tau$. 