CSE 505: Programming Languages

Lecture 13 — Safely Extending STLC: Sums, Products, Bools

Zach Tatlock
Fall 2013

Review

$$\begin{array}{lcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \\ v & ::= & \lambda x.\ e \mid c \end{array} \qquad \begin{array}{lcl} \tau & ::= & \mathsf{int} \mid \tau \to \tau \\ \Gamma & ::= & \cdot \mid \Gamma, x : \tau \end{array}$$

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

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

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

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

Preservation: If $\cdot \vdash e : \tau$ and $e \to e'$, then $\cdot \vdash e' : \tau$.
Progress: If $\cdot \vdash e : \tau$, then e is a value or $\exists\ e'$ such that $e \to e'$.

Adding Stuff

Time to use STLC as a foundation for understanding other common language constructs

We will add things via a *principled methodology* thanks to a *proper education*

- Extend the syntax
- Extend the operational semantics
  - Derived forms (syntactic sugar), or
  - Direct semantics
- Extend the type system
- Extend soundness proof (new stuck states, proof cases)

In fact, extensions that add new types have even more structure

Base Types and Primitives, in general

What about floats, strings, ...?
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* $(p_1 : \tau_1, \ldots, p_n : \tau_n)$. 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 $p_i\ v_1 \ldots v_n$ where p_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, but our lambda-calculus encoding won't type-check, nor will any encoding of equal expressive power

- So instead add an explicit construct for recursion
- You might be thinking **let rec** $f\ x\ = e$, but we will do something more concise and general but less intuitive

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

$$\frac{e \to e'}{\textbf{fix}\ e \to \textbf{fix}\ e'} \qquad \frac{}{\textbf{fix}\ \lambda x.\ e \to e[\textbf{fix}\ \lambda x.\ e/x]}$$

No new values and no new types

Using fix

To use **fix** like **let rec**, just pass it a two-argument function where the first argument is for recursion

- Not shown: **fix** and tuples can also encode mutual recursion

Example:

$$(\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))\ 5$$

$$\to$$

$$(\lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(n-1))))\ 5$$

$$\to$$

$$\textbf{if}\ (5{<}1)\ 1\ (5 * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(5-1))$$

$$\to^2$$

$$5 * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(5-1))$$

$$\to^2$$

$$5 * ((\lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(n-1))))\ 4)$$

$$\to$$

...

Why called fix?

In math, a fix-point of a function g is an x such that $g(x) = x$

- This makes sense only if g has type $\tau \to \tau$ for some τ
- A particular g could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type **int** $\to$ **int**:
  - $\lambda x.\ x + 1$ has no fix-points
  - $\lambda x.\ x * 0$ has one fix-point
  - $\lambda x.$ absolute_value(x) has an infinite number of fix-points
  - $\lambda x.$ **if** $(x < 10\ \&\&\ x > 0)\ x\ 0$ has 10 fix-points

Higher types

At higher types like $(\mathbf{int} \rightarrow \mathbf{int}) \rightarrow (\mathbf{int} \rightarrow \mathbf{int})$, the notion of fix-point is exactly the same (but harder to think about)

- For what inputs f of type $\mathbf{int} \rightarrow \mathbf{int}$ is $g(f) = f$

Examples:

- $\lambda f.\ \lambda x.\ (f\ x) + 1$ has no fix-points
- $\lambda f.\ \lambda x.\ (f\ x) * 0$ (or just $\lambda f.\ \lambda x.\ 0$) has 1 fix-point
  - The function that always returns 0
  - In math, there is exactly one such function (cf. equivalence)
- $\lambda f.\ \lambda x.$ absolute_value$(f\ x)$ has an infinite number of fix-points: Any function that never returns a negative result

Back to factorial

Now, what are the fix-points of
$\boldsymbol{\lambda f.\ \lambda x.}$ **if** $\boldsymbol{(x < 1)\ 1\ (x * (f(x-1)))}$?

It turns out there is exactly one (in math): the factorial function!

And **fix** $\boldsymbol{\lambda f.\ \lambda x.}$ **if** $\boldsymbol{(x < 1)\ 1\ (x * (f(x-1)))}$ behaves just like the factorial function

- That is, it behaves just like the fix-point of $\boldsymbol{\lambda f.\ \lambda x.}$ **if** $\boldsymbol{(x < 1)\ 1\ (x * (f(x-1)))}$
- In general, **fix** takes a function-taking-function and returns its fix-point

(This isn't necessarily important, but it explains the terminology and shows that programming is deeply connected to mathematics)

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** $\lambda x.\ e'$ need the same type
- The result means e' and **fix** $\lambda x.\ e'$ need the same type

Note: The τ in the typing rule is usually insantiated 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 let, 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
- OCaml sum types and record types have names

A never-ending debate:

- Structual types allow more code reuse: good
- Named types allow less code reuse: good
- Structural types allow generic type-based code: good
- Named types let type-based code distinguish names: good

The theory is often easier and simpler with structural types

Termination

Surprising fact: If $\cdot \vdash e : \tau$ in STLC 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

The proof requires more advanced techniques than we have learned so far because the size of expressions and typing derivations does not decrease with each program step

- Could present it in about an hour if desired

Non-proof:

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