CS 411 Lecture 15 Intro to the Simply-Typed Lambda Calculus 03 Oct 2003
Lecturer: Greg Morrisett

In the pure lambda calculus, the only values we have are functions. We can encode just about anything with functions, but the encodings aren't very practical. In real languages, such as ML or Java, we have a set of built in constants and operations, and perhaps built in data structures such as record/object/tuples/etc.

So, let's consider what happens if we add a few built-in things to the lambda calculus:

$$e \in \mathsf{Exp} ::= x \mid \lambda x.e \mid e_1\ e_2 \mid i \mid e_1 + e_2$$

To keep things simple for now, I've only added integers (i) and addition ($e_1 + e_2$). So, our values consist of functions and integers:

$$v \in \mathsf{Value} ::= \lambda x.e \mid i$$

Note that every value is an expression, so values are a subtype of expressions.

We need to update our operational semantics. The (call-by-value) small-step semantics for this language might look the following:

$$(\lambda x.e_1)\ v \rightarrow e_1[v/x]$$

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2}$$

$$\frac{e_2 \rightarrow e_2'}{(\lambda x.e)\ e_2 \rightarrow (\lambda x.e)\ e_2'}$$

$$i_1 + i_2 \rightarrow i \qquad (i = i_1 + i_2)$$

$$\frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2}$$

$$\frac{e_2 \rightarrow e_2'}{i + e_2 \rightarrow i + e_2'}$$

The large-step semantics would look like this:

$$i \Downarrow i$$

$$\lambda x.e \Downarrow \lambda x.e$$

$$\frac{e_1 \Downarrow \lambda x.e \qquad e_2 \Downarrow v_2 \qquad e[v_2/x] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 + e_2 \Downarrow i}(i = i_1 + i_2)$$

All looks good, except that there's a problem. What happens if we attempt to evaluate the expression:

$$\begin{aligned}(\lambda x.\lambda y.x + y)\ 3\ (\lambda z.z) \\ \rightarrow (\lambda y.3 + y)\ (\lambda z.z) \\ \rightarrow 3 + (\lambda z.z)\end{aligned}$$

Now we're stuck. The problem is that we're trying to add a function ($\lambda z.z$) to an integer. But the primitive + is really only defined on integers. There are two ways we can try to fix this problem: (1) make + defined on all possible values, (2) try to rule out these kinds of type errors before running the program.

For example, we could just add an axiom:

$$v_1 + v_2 \to 0 \qquad (v_1 \text{ or } v_2 \text{ not an integer})$$

Of course, that's not the only problem. We can also get stuck in a situation where we attempt to call an integer:

$$\begin{aligned}(\lambda x.\lambda y.y(x))\ 3\ 42 \\ &\to (\lambda y.y(3))\ 42 \\ &\to 42(3)\end{aligned}$$

Again, we could just add an axiom:

$$v_1\ v_2 \to 0 \qquad (v_1 \text{ not a function})$$

Notice that an implementor of the language has to be able to tell functions from integers at run-time and detect these bad situations. In other words, at run-time, we will need type tags and type tests to implement this semantics. This is effectively what Scheme and other dynamically typed languages do. However, instead of returning 0, they usually return some distinguished error value:

$$\begin{aligned}v_1 + v_2 &\to \mathsf{error} \qquad (v_1 \text{ or } v_2 \text{ not an integer}) \\ v_1\ v_2 &\to \mathsf{error} \qquad (v_1 \text{ not a function})\end{aligned}$$

Furthermore, they propagate the error:

$$(\lambda x.e)\ \mathsf{error} \to \mathsf{error}$$

So, if you evaluate a sub-expression and get an error, you'll get an error for the whole expression. Just like, if you evaluate a sub-expression and it doesn't terminate, then the whole expression doesn't terminate.

An alternative is to try to rule out such type-errors at compile time by doing an analysis, similar to what you did for IMP. In the little language above, we need to be able to distinguish integer values from functions so that we won't get stuck with + or application. But we need a bit more than that – we also need to know what kinds of arguments a function might take, and what kind of results it yields in order to ensure that we never get stuck. This gives rise to the following simple type structure:

$$\tau \in \mathsf{Type} ::= \mathsf{int} \mid \tau_1 \to \tau_2$$

Now, we're going to construct a syntax-directed analysis of program to figure out what the type of each expression is. We'll rule out programs of the form $e_1 + e_2$ when e_1 and e_2 might evaluate to non-integers, and we'll rule out expressions of the form $e_1\ e_2$ when e_1 might evaluate to a non-function.

To do this, we'll use a relation of the form:

$$\Gamma \vdash e : \tau$$

Here, Γ is a type environment (a.k.a. symbol table) and is a partial function from variables to types:

$$\Gamma \in \mathsf{TyEnv} : \mathsf{Var} \rightharpoonup \mathsf{Type}$$

The proof rules are as follows:

$\Gamma \vdash i : \mathsf{int}$

$\Gamma \vdash x : \Gamma(x)$ – note that Γ must be defined on x to use this rule

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \qquad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

We say a program e is well-typed if $\emptyset \vdash e : \tau$, usually just written $\vdash e : \tau$. That is, if under an empty set of typing assumptions, we can prove that e has type τ, then e is well-typed.

Theorem 1. *If $\vdash e : \tau$, then either e is a value v or else there exists an e' such that $e \to e'$ and $\vdash e' : \tau$.*

What does this theorem say? It says that if we have a well-typed expression, then it's either a value (an answer) so there's no step to take, or else it can step to some e' and e' will be well-typed.

More importantly, let us write down all the possible terminal states (i.e., states for which there is no small step to take) and classify them as "good" (i.e., answers) or "stuck" (i.e., a type error of some sort):

Good terminal states:

i – okay, a value
$\lambda x.e$ – okay, a value

Bad terminal states:

x – a free variable
$i\ v$ – application of an integer
$(\lambda x.e) + v$ – addition of a function on left
$v + (\lambda x.e)$ – addition of a function on right
$e_1\ e_2$ – where e_1 is stuck
$v\ e_2$ – where e_2 is stuck
$e_1 + e_2$ – where e_1 is stuck
$v + e_2$ – where e_2 is stuck

A corollary of the theorem above is that if $\vdash e : \tau$, then there is no e' such that $e \to^* e'$ and e' is a bad terminal state. In other words, well-typed expressions cannot get stuck. We either evaluate to a value or run forever, but we can't get into one of these stuck states.

If this theorem is true, then an implementor does not need type-tags or type-tests at run-time. They've proven that well-typed programs can't get stuck at say, an addition of a function. So they don't have to check for this bad case at run-time.

The theorem above is usually called "Type Soundness". It's really saying that our little analysis is conservative – it will never say something is well-typed that might end up with a run-time type error.

How do we prove the theorem? We can break it into two pieces:

(a) show that $\vdash e : \tau$ is invariant under evaluation. That is, if $\vdash e : \tau$ and $e \to e'$, then $\vdash e' : \tau$.

(b) show that if $\vdash e : \tau$, then either e is a value (i.e., good terminal state) or else there exists an e' such that $e \to e'$.

Lemma (a) is called Preservation (aka Subject Reduction) and lemma (b) is called Progress. Basically, we're showing that well-typedness is a "loop invariant" for the meta-level interpreter of the language.