CS611 Lecture 31 **The Simply Typed Lambda Calculus** 14 November, 2003

Scribe: Revelino Guron, Jr., Milos Hasan Lecturer: Radu Rugina

1 Call with Current Continuation

In the last lecture, we introduced constructs to express the call-with-current-continuation mechanism:

$$e \quad ::= \quad \ldots \quad | \quad \mathsf{throw}\ e_\kappa\ e_v \quad | \quad \mathsf{callcc}\ e_f$$

When callcc occurs in the program, it is passed an expression that evaluates to a function. That function is then invoked with the current continuation as its argument. When throw is invoked, e_κ is evaluated to a continuation, e_v is evaluated to a value and the continuation is called with the value as its argument. We can define this process formally.

$$\begin{array}{lcl} \mathcal{C}[\![\mathsf{callcc}\ e_f]\!]\rho\kappa & = & \mathcal{C}[\![e_f]\!]\rho\ (\mathit{check\_fun}(\lambda f.\ f\ \kappa\ \kappa)) \\ \mathcal{C}[\![\mathsf{throw}\ e_\kappa\ e_v]\!]\rho\kappa & = & \mathcal{C}[\![e_\kappa]\!]\rho\ (\mathit{check\_cont}(\lambda\kappa'. \\ & & \mathcal{C}[\![e_v]\!]\rho\ (\mathit{check\_val}(\lambda v.\ \kappa'\ v)))) \end{array}$$

Notice that in the denotation of callcc, the function f is passed κ twice. The first times as the function's regular argument, and the second time as the function's continuation argument. Also, notice in throw that we don't use the current continuation. This is because we want use the continuation where callcc was called and not the continuation when throw is invoked.

2 Exceptions

The final non-local control flow constructs we are going to add to the language are exceptions. They are similar to the exceptions used in Java.

$$e \quad ::= \quad \ldots \quad | \quad \mathsf{throw}\ s\ e \quad | \quad \mathsf{try}\ e\ \mathsf{catch}\ (s\ x)\ e'$$

Intuitively, when throw is invoked, e is evaluated to a value, and that value is bound to the variable in a catch which matches the throw's exception name s. When try and catch are used, the expression under the try is evaluated with s bound to the exception handler e'. When an exception is caught, e' is evaluated with variable x bound to the passed value (the value evaluated in the throw statement). We will need to add and change some of our definitions. First, we need to define handler environments as follows:

$$h \quad \in \quad \mathit{Handler} \quad = \quad \mathit{ExcName} \to \mathit{Cont}$$

We look up an exception name in h and we expect it to return a continuation. If there is no continuation currently associated with an exception name, then trying to look up that name results in an error. Second, we need to change our definition of the denotation of an expression and the signature of functions. They must additionally take in a handler environment:

$$\begin{array}{lcl} \mathcal{C}[\![e]\!] & : & \mathit{Env} \to \mathit{Handler} \to \mathit{Cont} \to \mathit{Result} \\ \mathit{Function} & = & \mathit{Value} \to \mathit{Handler} \to \mathit{Cont} \to \mathit{Result} \end{array}$$

Next, let's look at the newly added constructs' denotations:

$$\begin{array}{lcl} \mathcal{C}[\![\mathsf{throw}\ s\ e_v]\!]\rho h\kappa & = & \mathcal{C}[\![e_v]\!]\rho h(\mathit{check\_val}(\lambda v.\ h\ s\ v)) \\ \mathcal{C}[\![\mathsf{try}\ e\ \mathsf{catch}\ (s\ x)\ e']\!]\rho h\kappa & = & \mathcal{C}[\![e]\!]\rho h[s \mapsto \mathcal{C}[\![\lambda x.\ e']\!]\rho h\kappa]\ \kappa \end{array}$$

Notice that in the denotation of throw, we assume that s already maps to some valid continuation, otherwise an error continuation is returned. The name s is bound in the denotation of the try and catch construct.

Finally, let's see how adding exceptions and handlers change the denotation of applications and abstractions.

$$
\begin{array}{lcl}
\mathcal{C}[\![\lambda x.\ e]\!]\rho h\kappa & = & \kappa\ (\lambda v.\ \lambda h'.\ \lambda\kappa'.\ \mathcal{C}[\![e]\!]\rho[x \mapsto v]\ h'\ \kappa') \\
\mathcal{C}[\![e_0\ e_1]\!]\rho h\kappa & = & \mathcal{C}[\![e_0]\!]\rho h(\mathit{check\_fun}(\lambda f. \\
 & & \mathcal{C}[\![e_1]\!]\rho h(\mathit{check\_val}(\lambda v.\ f\ v\ h\ \kappa))))
\end{array}
$$

To conclude the presentation of non-local flow constructs, we say that the semantics of such constructs could also be expressed using operational semantics, but CPS makes the semantics more compact and more elegant, because it makes continuations explicit in the semantics.

3 Static Program Analysis

Until now, we were concerned with the *execution* of programs in languages like IMP[1], lambda calculus and FUN. The rest of the lectures will deal with *static program analysis* – reasoning about properties of programs without actually executing them (i.e., at compile time). There are several techniques that fall into this category – type systems, abstract interpretations and proving theorems about programs.

We will first deal with type systems. Typing is present in both imperative and functional languages; its main purpose is to guarantee absence of certain kinds of errors in a program.

4 The Simply Typed Lambda Calculus ($\lambda^{\rightarrow}$)

Since lambda calculus provides a theoretical basis for functional programming languages, it is not surprising that type systems for these languages are based on *typed lambda calculus*.

4.1 Syntax of Typed Lambda Calculus

To extend lambda calculus with types, we slightly extend the grammar for untyped lambda calculus. A lambda abstraction will now also contain the type of its argument variable. So the set of *expressions* in typed lambda calculus is defined as:

$$e ::= x \mid e_0\ e_1 \mid \lambda x : \tau.e \mid p$$

where p denotes *primitive values*, defined as:

$$p ::= n \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{unit}$$

and τ denotes *types*, defined as:

$$\tau ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{unit} \mid \tau_1 \rightarrow \tau_2$$

The set of *values* is defined as the set of all primitive values and abstractions:

$$v ::= p \mid \lambda x : \tau.e$$

Note that this is a core calculus, useful for understanding the foundations of typing. At this point, it doesn't contain arithmetic or logic operators to manipulate integer and boolean values. In the following lectures, we will add other constructs on top of this core calculus.

The *dynamic* (operational) semantics does not change compared to the untyped calculus. In other words, computation in typed lambda calculus is the same as in untyped lambda calculus. (In fact, the operational semantics rules are slightly different, to incorporate the syntax extensions.)

[1] With the exception of axiomatic semantics.

4.2 Static Semantics of Typed Lambda Calculus

Next, we define typing rules for expressions in our calculus – the goal of these rules is to only make typable those expressions that will not result in a stuck evaluation. The set of these typing rules is called a *type system*, or *static semantics*.

In the following, we use the notion of a typing environment, which is a function that assigns types to variables:

$$\Gamma : \mathit{Var} \to \mathit{Type}$$

We will denote the fact that an expression e has type τ in an environment Γ as:

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

Here are the typing rules for expressions in the typed lambda calculus:

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \quad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \quad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \quad \frac{}{\Gamma \vdash \mathsf{unit} : \mathsf{unit}}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad \frac{\Gamma[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau . e) : \tau \to \tau'} \quad \frac{\Gamma \vdash e_0 : \tau' \to \tau \quad \Gamma \vdash e1 : \tau'}{\Gamma \vdash e_0\ e_1 : \tau}$$

In the rule for abstractions, we need to extend the type environment in the premise with a type τ for x. In other words, before type-checking the body of a function, we must add the type of its argument to the type environment.

Notice that type-checking a lambda expression can be done with a simple deterministic algorithm that always terminates. This follows from the fact that, in each rule, the expressions in the premise (if any) are strict subexpressions of the expression in the conclusion. Because it is guaranteed to terminate, the process of type-checking can be performed statically, i.e., at compile-time.

4.3 Soundness of Typed Lambda Calculus

Of course, the typing rules are not chosen arbitrarily. The goal is to make the type system sound, in the sense that the evaluation of a well-typed expression should never get "stuck". More precisely, this means that the evaluation either terminates with a value or can be reduced further. This *soundness* property can be formally expressed as:

$$(\vdash e : \tau \ \wedge \ e \to_1^* e') \ \Rightarrow \ (e' \in \mathit{Value} \ \vee \ \exists e''. e' \to_1 e'')$$

The notation $\vdash e : \tau$ is shorthand for $\emptyset \vdash e : \tau$.