Simply Typed Lambda Calculus

Scribe: Alan Shieh, Kevin Walsh

Lecturer: Radu Rugina

1 Exceptions Using CPS Semantics

As a final example of the use of CPS semantics, we will add Java-style exceptions to the lambda calculus:

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

Where $\textsf{throw}\ s\ e$ will mean:

- evaluate expression e in the current context
- pass the result to the exception handler bound to s

And $\textsf{try}\ e\ \textsf{catch}\ (s\ x)\ e'$ will mean:

- evaluate expression e with s bound to the exception handler e'
- when handling an exception, evaluate e' with x bound to the received value.

1.1 Denotational Semantics

We will need to add a new domain for the current mappings of exception handlers. We also modify the denotation and function domains, passing the binding of symbols to exception continuations.

$$\begin{aligned}
\mathit{Handlers} &::= \mathit{Sym} \to \mathit{Cont} \\
\mathit{Function} &::= \mathit{Value} \to \mathit{Handlers} \to \mathit{Cont} \to \mathit{Result} \\
\mathcal{C}[\![e]\!] &::= \mathit{Env} \to \mathit{Handlers} \to \mathit{Cont} \to \mathit{Result}
\end{aligned}$$

We can now define the denotation of exception handling constructs. The additional parameter h in the denotation stores the current binding of symbols to exception continuations. For the throw construct, we evaluate e, check that the result is a Value , then pass the result to the continuation $h(s)$. For the catch construct, we simply evaluate e with s bound to the appropriate exception handler.

$$\begin{aligned}
\mathcal{C}[\![\textsf{throw}\ s\ e]\!]\rho h\kappa &= \mathcal{C}[\![e]\!]\rho h(\mathit{check_val}\,(\lambda v\ .\ h\ s\ v)) \\
\mathcal{C}[\![\textsf{try}\ e\ \textsf{catch}\ (s\ x)\ e']\!]\rho h\kappa &= \mathcal{C}[\![e]\!]\rho h[s \mapsto (\lambda v\ .\ \mathcal{C}[\![e']\!]\rho[x \mapsto v]h\kappa)]\kappa
\end{aligned}$$

We must additionally modify the other rules, passing the current handler bindings h. Function application becomes:

$$\mathcal{C}[\![e_0\ e_1]\!]\rho h\kappa \quad = \quad \mathcal{C}[\![e_0]\!]\rho h(\mathit{check\_fun}\,(\lambda f\ .\ \mathcal{C}[\![e_1]\!]\rho h(\mathit{check\_val}\,(\lambda v\ .\ f\ v\ h\ \kappa))))$$

The denotation for abstraction is left as an exercise. We will now leave continuation passing style semantics, and move on to types and type-checking.

2 Typed Lambda Calculus ($\lambda^{\rightarrow}$)

We have seen how the use of booleans, functions, errors, and other constructs force the semantic rules to check that a program is of the correct form. In denotational semantics, errors are handled explicitly. In operational semantics, errors are represented by stuck configurations. Both approaches are dynamic mechanisms, in that they detect errors during execution. Also, these mechanisms allow us to have programs that are correct for some inputs but encounter errors on others.

We would like an alternative: static guarantees that a program will execute without errors. These guarantees should be provided without having to run the program. A complete solution would allow us to solve the halting problem, so instead we provide an approximate solution. Using types (program annotations) and typing rules (inference rules) we can provide static guarantees:

- If a program is *well typed* , then it will not produce an error during execution.
- If a program is not *well typed*, the program may or may not produce an error during execution.

We can compare program correctness under typing rules to correctness under axiomatic semantics. The former is a lightweight mechanism, checking only for non-error states. The latter is heavyweight, checking that a program produces the desired output. In practice, the light-weight type-checking mechanism is still powerful enough to be useful.

2.1 Syntax of $\lambda^{\rightarrow}$

To formalize this notion of types, we will introduce a typed lambda calculus, denoted as $\lambda^{\rightarrow}$, and with the following syntax:

$$e := x \;\mid\; e_0\ e_1 \;\mid\; \lambda x\!:\!\tau.e \;\mid\; p_e$$

Note the new syntax for abstractions: $x\!:\!\tau$ means "bind x to type τ." We will introduce primitive values p_e for numbers and booleans, instead of constructing them from abstractions as before.

$$p_e := 0 \;\mid\; 1 \;\mid\; \ldots \;\mid\; \mathsf{true} \;\mid\; \mathsf{false} \;\mid\; \textbf{unit}$$

We shall define types as

$$\begin{aligned} \tau \in \mathit{Type} \quad &::= \quad P_t \;\mid\; \tau_o \rightarrow \tau_1 \\ P_t \quad &::= \quad \mathsf{int} \;\mid\; \mathsf{bool} \;\mid\; 1 \end{aligned}$$

where P_t are the primitive types, and $\tau_o \rightarrow \tau_1$ is a composition of types to represent the type of an abstraction. Note that 1 represents the type of **unit**, and has nothing to do with the integer 1.

Examples

The expression

$$\mathit{true}_{\lambda} = \lambda x\ .\ \lambda y\ .\ x$$

may be written in $\lambda^{\rightarrow}$ as

$$\mathit{true}_{int} = \lambda x\!:\!\mathsf{int}\ .\ \lambda y\!:\!\mathsf{int}\ .\ x$$

with type

$$\mathit{true}_{int}\!:\!\mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int}$$

Primitive values are also typed:

$$2\!:\!\mathsf{int}$$

2.2 Operational semantics

We will not need to change the semantics for program execution. Instead, the semantics for the typed lambda calculus will impose constraints on the parameters that functions can take.

$$\frac{e_o \to e_0'}{e_0\ e_1 \to e_0'\ e_1} \qquad \frac{}{(\lambda x\!:\!\tau.e)e' \to e[x \to e']}$$

The definition of *Value* is changed to handle the new primitives:

$$v \in \mathit{Value} = \lambda x\!:\!\tau.e \ \mid\ p_e$$

We will now try to express formally our earlier claim that if an expression is well-typed, then no stuck configuration may result during execution.

$$(e\!:\!\tau \wedge e \to^* e') \to (e' \in \mathit{Value} \vee \exists e''.e' \to e'')$$

In other words, if e evaluates to e', then e' must be a primitive or an abstraction, or can be evaluated to another form e''.

Examples

Define

$$\mathit{test}_{int} = \lambda b\!:\!\mathsf{int} \to \mathsf{int} \to \mathsf{int}\ .\ \lambda x\!:\!\mathsf{int}\ .\ \lambda y\!:\!\mathsf{int}\ .\ b\ x\ y$$

Let us try evaluating an application of this abstraction in a situation that we intuitively know to be well-typed:

$$\mathit{test}_{int}\ \mathit{true}_{int}\ 2\ 3\ \ \to^*\ \ \mathit{true}_{int}\ 2\ 3\ \ \to^*\ \ 2$$

In this case, the evaluation terminates at a value. Now, try a similar example but using the primitive value true instead of true_{int}:

$$\mathit{test}_{int}\ \mathsf{true}\ 2\ 3 \to^* \mathsf{true}\ 2\ 3$$

The result is neither a value nor an application, but a mal-formed sequence of primitive values. This is a stuck configuration, and indeed the expression on the left was not well-typed since true is not of type $\mathsf{int} \to \mathsf{int} \to \mathsf{int}$ as required by test_{int}.

2.3 Type-checking

We will formalize our ideas about typing with a proof system consisting of typing rules. We will require the notion of a typing context, defined as a partial function:

$$\Gamma : \mathit{Var} \rightharpoonup \mathit{Types}$$

which assigns types to some of the variables. The empty context ϕ denotes a context with no type mappings. The structure of our proof system will be as follows:

$\Gamma \vdash e\!:\!\tau$ reads as: "with respect to context Γ, e is well-typed and has type τ".

This requires us to inductively define rules for each possible e.

Typing Rules:

$$\frac{}{\Gamma \vdash n\!:\!\mathsf{int}} \qquad \frac{}{\Gamma \vdash \mathsf{true}\!:\!\mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false}\!:\!\mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{unit}\!:\!1}$$

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

Example

As an example we will construct a proof tree using these rules to demonstrate that $(\lambda x : \mathsf{int}.x)\ 2$ is well-typed, with type int.

$$\dfrac{\dfrac{\overline{\Gamma[x \rightarrow \mathsf{int}] \vdash x : \mathsf{int}}}{\Gamma \vdash (\lambda x : \mathsf{int}.x) : \mathsf{int} \rightarrow \mathsf{int}} \quad \overline{\Gamma \vdash 2 : \mathsf{int}}}{\Gamma \vdash (\lambda x : \mathsf{int}.x)\ 2 : \mathsf{int}}$$

Non-termination

The following example demonstrates how an expression whose evaluation doesn't terminate in the untyped lambda calculus, is provably not well-typed in $\lambda^{\rightarrow}$. Recall:

$$\begin{aligned} \mathit{omega}_\lambda &= (\lambda x.xx)(\lambda x.xx) \\ \mathit{self}_\lambda &= \lambda x.xx \end{aligned}$$

In the untyped lambda calculus, the evaluation of *omega* doesn't terminate.

We will attempt to derive a type for $\lambda x : \tau\ .\ x\ x$ as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{\boxed{\Gamma[x \rightarrow \tau] \vdash x : \tau'' \rightarrow \tau'}} \quad \dfrac{\vdots}{\boxed{\Gamma[x \rightarrow \tau] \vdash x : \tau''}}}{\Gamma[x \rightarrow \tau] \vdash x\ x : \tau'}}{\Gamma \vdash (\lambda x : \tau.x\ x) : \tau \rightarrow \tau'}$$

This is the only possible form for a proof tree. Looking at the boxed expressions, we must have $\tau'' = \tau'' \rightarrow \tau'$. There is no finite solution satisfying this constraint, since τ'' is included in itself. Hence, no proof tree is constructible, and *self* is not well-typed.