CS611 Lecture 27: Simply-typed lambda calculus 11/01/99
Vladimir Kolmogorov

1 Syntax of simply-typed λ-calculus ($\lambda^{\rightarrow}$)

$$\begin{array}{rrcl} \text{expressions:} & e & ::= & e_0\, e_1 \mid x \mid \lambda x : \tau\, .\, e \\ \text{types:} & \tau & ::= & B \mid \tau_0 \rightarrow \tau_1 \\ \text{basetypes:} & B & ::= & \mathbf{bool} \mid \mathbf{int} \mid \mathbf{unit} \mid \ldots \end{array}$$

We will also assume that there are some predefined expressions with predefined types (such as constants, operation of addition, etc.).

For example, TRUE expression can be written as

$$TRUE_{\mathbf{int}} \equiv (\lambda t : \mathbf{int}{\rightarrow}\mathbf{int}{\rightarrow}\mathbf{int}\, .\, (\lambda a : \mathbf{int}\, .\, (\lambda b : \mathbf{int}\, .\, (t\ a\ b))))$$

Expressions that are not type-checked (e.g. $(3\,(\lambda x : \mathbf{int}\, .\, x))$) correspond to expressions that get stuck in the semantics for untyped λ-calculus.

2 Typing assertions

Instead of checking arbitrary assertions as in axiomatic semantics we'd like to be able to check typing assertions of the form $\vdash e : \tau$. Since variables can have an arbitrary type we need to have a type context

$$\Gamma : var \rightarrow \tau$$

such that $\Gamma \vdash x : \Gamma(x)$ is an axiom. Γ can also be represented by a set of pairs $\{x_1 : \tau_1, x_2 : \tau_2, \ldots\}$, then we have the rule

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

or $\Gamma, x : \tau \vdash x : \tau$ where $\Gamma, x : \tau$ denotes $\Gamma\,[x \mapsto \tau]$ - substitution the type τ for variable x in Γ.

A program is typeable (or well-formed) if it can be assigned a type τ such that $\emptyset \vdash e : \tau$

2.1 Typing rules

$$\Gamma \vdash x : \tau \qquad (\text{where } \Gamma(x) = \tau)$$

$$\frac{\Gamma \vdash e_0 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_0\, e_1 : \tau_2}$$

$$\frac{\Gamma\,[x \mapsto \tau] \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau\, .\, e : \tau \rightarrow \tau'}$$

These rules give static semantics for $\lambda^{\rightarrow}$. Later we'll add rules allowing new syntax: $\tau ::= \ldots \mid \tau + \tau \mid \tau \times \tau$.

2.2 Example of type-checking

Consider expression $((\lambda a : \mathbf{int}\,(\lambda b : \mathbf{int}\, .\, b))\, 1)$. The derivation tree for it is the following:

$$\dfrac{\dfrac{\dfrac{\{a : \mathbf{int}, b : \mathbf{int}\} \vdash b : \mathbf{int}}{\{a : \mathbf{int}\} \vdash (\lambda b : \mathbf{int}\, .\, b) : \mathbf{int} \rightarrow \mathbf{int}}}{\emptyset \vdash (\lambda a : \mathbf{int}\,(\lambda b : \mathbf{int}\, .\, b)) : \mathbf{int} \rightarrow \mathbf{int} \rightarrow \mathbf{int}} \qquad \dfrac{}{\emptyset \vdash 1 : \mathbf{int}}}{\emptyset \vdash ((\lambda a : \mathbf{int}\,(\lambda b : \mathbf{int}\, .\, b))\, 1) : \mathbf{int} \rightarrow \mathbf{int}}$$

3 Soundness and expressive power

Two questions can be asked about typing rules:

- Are these rules sound?
- Do we loose the expressive power of untyped λ-calculus?

The answer to the former question is "yes", and to the latter one is "no".

3.1 Expressive power

In addition to the Church-Rosser property the property of strong normalization holds for $\lambda^{\rightarrow}$: all reductions starting from a well-formed expression terminate. Thus, expressions that don't terminate in untyped λ-calculus cannot be written in $\lambda^{\rightarrow}$.

As an example consider the expression $(\lambda x\,(x\,x))$ in untyped λ-calculus. Suppose we can write an equivalent expression in $\lambda^{\rightarrow}$: $(\lambda x : \tau\,(x\,x))$ of type $\tau \rightarrow \tau'$. Its derivation tree is as follows.

$$\frac{\dfrac{\{x:\tau\} \vdash x : \tau \rightarrow \tau' \qquad \{x:\tau\} \vdash x : \tau}{\{x:\tau\} \vdash x\,x : \tau'}}{\emptyset \vdash (\lambda x : \tau\,(x\,x)) : \tau \rightarrow \tau'}$$

Thus, types τ and $\tau \rightarrow \tau'$ should be equal. However, it is possible only if τ is an infinite type.

3.2 Soundness

Soundness of typing rules means that we won't get stuck while evaluating well-formed expressions. It can be expressed by the following statement:

$$\vdash e : \tau \;\Rightarrow\; (\forall e' \,.\, e \rightarrow^* e' \;\Rightarrow\; (\vdash e' : \tau) \;\wedge\; (value(e') \vee \exists e'' \,.\, e' \rightarrow e''))$$

In order to show this statement we need to show that the following two properties hold:

- Preservation
$$\vdash e : \tau \;\Rightarrow\; (e \rightarrow e' \Rightarrow \vdash e' : \tau)$$
- Progress
$$\vdash e : \tau \;\Rightarrow\; (\exists e' \,.\, e \rightarrow e') \vee value(e)$$