Logic in Computer Software
Course Note #2 (Multi-Stage Programming Language)
Yukiyoshi Kameyama, 2013.

1 Syntax (構文)

$$
\begin{aligned}
& x, y, z, \ldots && \text{variable (変数)} \\
c &::= 0 \mid 1 \mid -1 \mid \cdots \mid \texttt{true} \mid \texttt{false} && \text{constant (定数)} \\
e &::= c \mid x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 = e_2 \mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
&\quad \mid \lambda x.\ e \mid e_1\ e_2 \mid \lambda(x, y, ..., z).\ e \mid e_1\ (e_2, e_3, ..., e_n) \\
&\quad \mid \texttt{fix}\ (f(x).\ e \mid \texttt{fix}\ f(x, y, ..., z).\ e \\
&\quad \mid \langle e \rangle \mid {\sim}e \mid \texttt{run}\ e && \text{追加された項}
\end{aligned}
$$

Abbreviation: $\texttt{let}\ \ x = e_1\ \ \texttt{in}\ \ e_2$ is $(\lambda x.\ e_2)\ e_1$.

Intuition: $\langle e \rangle$ returns a code for computing e, and ${\sim}e$ is used for splicing (composing a code with another code). $\texttt{run}\ e$ (compiles and) executes the generated code (when e evaluates to a code).

Later, we will add a term $\texttt{\%}\ e$ for cross-stage persistence (explained later).

Examples of terms with brackets staged constructs:

$$
\begin{aligned}
\texttt{let}\ \ x = 1 + 2\ \ \texttt{in}\ \ x * 3 &\leadsto^* 9 \\
\langle 1 \rangle &\leadsto^* \langle 1 \rangle && \text{returns a code} \\
\texttt{run}\ \langle 1 \rangle &\leadsto^* 1 \\
\langle 1 + 2 \rangle &\leadsto^* \langle 1 + 2 \rangle && \text{code is not evaluated} \\
\texttt{run}\ \langle 1 + 2 \rangle &\leadsto^* 3 && \text{and can be executed} \\
\langle 1/0 \rangle &\leadsto^* \langle 1/0 \rangle && \text{code is not evaluated} \\
\texttt{run}\ \langle 1/0 \rangle &\leadsto^* \text{(exception: division by zero)} \\
\langle 1 + 2 \rangle + 3 &\leadsto^* \text{(error)} \\
\langle 1 + 2 \rangle + \langle 3 \rangle &\leadsto^* \text{(error)} \\
\langle 1 + x \rangle &\leadsto^* \text{error} \\
\texttt{let}\ \ x = \langle 1 + 2 \rangle\ \ \texttt{in}\ \ \langle 3 \rangle &\leadsto^* \langle 3 \rangle \\
\texttt{let}\ \ x = \langle 1 + 2 \rangle\ \ \texttt{in}\ \ \langle x * 3 \rangle &\leadsto^* \text{(error)} \\
\texttt{let}\ \ x = \langle 1 + 2 \rangle\ \ \texttt{in}\ \ \langle {\sim}x * 3 \rangle &\leadsto^* \langle (1 + 2) * 3 \rangle \\
\texttt{run}\ (\texttt{let}\ \ x = \langle 1 + 2 \rangle\ \ \texttt{in}\ \ \langle {\sim}x * 3 \rangle) &\leadsto^* 9 \\
\texttt{let}\ \ x = \langle 1 + 2 \rangle\ \ \texttt{in}\ \ \langle {\sim}x * {\sim}x \rangle &\leadsto^* \langle (1 + 2) * (1 + 2) \rangle \\
\texttt{let}\ \ x = \langle 1 + 2 \rangle\ \texttt{in}\ \texttt{let}\ \ y = \langle {\sim}x * 3 \rangle\ \ \texttt{in}\ \ \langle {\sim}y + {\sim}x \rangle &\leadsto^* \langle ((1 + 2) * 3) + (1 + 2) \rangle
\end{aligned}
$$

More examples with escapes:

$$\begin{aligned}
\sim 1 &\rightsquigarrow^* \text{(error)} \\
\sim\langle 1\rangle &\rightsquigarrow^* \text{(error)} \\
\langle \sim\langle 1+2\rangle\rangle &\rightsquigarrow^* \langle 1+2\rangle \\
\langle \sim\langle 1+2\rangle + \sim\langle 3*4\rangle\rangle &\rightsquigarrow^* \langle (1+2)+(3*4)\rangle \\
\texttt{let}\ x = \langle\langle 1+2\rangle\rangle\ \texttt{in}\ \langle \sim\sim x\rangle &\rightsquigarrow^* \text{(error)} \\
\texttt{let}\ x = \langle\langle 1+2\rangle\rangle\ \texttt{in}\ \langle\langle \sim\sim x\rangle\rangle &\rightsquigarrow^* \langle\langle 1+2\rangle\rangle
\end{aligned}$$

Code with variables:

$$\begin{aligned}
\texttt{let}\ x = \langle 1+2\rangle\ \texttt{in}\ \langle \lambda y.\ y + \sim x\rangle &\rightsquigarrow^* \langle \lambda y.\ y + (1+2)\rangle \\
\lambda x.\ \langle \lambda y.\ y + \sim x\rangle &\rightsquigarrow^* \lambda x.\ \langle \lambda y.\ y + \sim x\rangle \\
\langle \texttt{let}\ x = 10\ \texttt{in}\ \sim(\texttt{let}\ y = \langle 20\rangle\ \texttt{in}\ \langle \sim y + x\rangle)\rangle &\rightsquigarrow^* \langle \texttt{let}\ x = 10\ \texttt{in}\ 20 + x\rangle \\
\langle \texttt{let}\ x = 10\ \texttt{in}\ \sim(\texttt{let}\ y = \langle x\rangle\ \texttt{in}\ \langle \sim y + x\rangle)\rangle &\rightsquigarrow^* \langle \texttt{let}\ x = 10\ \texttt{in}\ x + x\rangle
\end{aligned}$$

Computation Rules (informally given)

$$\begin{aligned}
\langle \cdots \sim\langle v\rangle \cdots\rangle &\rightsquigarrow \langle \cdots v \cdots\rangle \\
\texttt{run}\ \langle v\rangle &\rightsquigarrow v
\end{aligned}$$

Note: the notion of a value v should be re-defined for the extended language. We omit its definition here for brevity.

More examples of computation:

$$\begin{aligned}
\texttt{let}\ x = \langle 1+2\rangle\ \texttt{in}\ \texttt{run}\ \langle \sim x * (\sim x * 3)\rangle &\rightsquigarrow \texttt{run}\ \langle \sim\langle 1+2\rangle * (\sim\langle 1+2\rangle * 3)\rangle \\
&\rightsquigarrow \texttt{run}\ \langle (1+2) * (\sim\langle 1+2\rangle * 3)\rangle \\
&\rightsquigarrow \texttt{run}\ \langle (1+2) * ((1+2) * 3)\rangle \\
&\rightsquigarrow (1+2) * ((1+2) * 3) \\
&\rightsquigarrow^* 27
\end{aligned}$$

2 Examples of programs

Ordinary (unstaged) power function:

$$\begin{aligned}
\texttt{power} &= \lambda x.\ \texttt{fix}\ f(n).\ \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ x * (f(n-1)) \\
\texttt{power}\ 5\ 3 &= 125
\end{aligned}$$

Staged power function and its usage (1st **buggy** version):

$$\begin{aligned}\texttt{spower} &= \lambda x.\ \texttt{fix}\ f(n).\ \texttt{if}\ n = 0\ \texttt{then}\ \langle 1\rangle\ \texttt{else}\ \langle{\sim}x * {\sim}(f(n-1))\rangle \\ \texttt{spower}\ y\ 3 &\rightsquigarrow^* \text{(error)} \\ \texttt{spower}\ \langle y\rangle\ 3 &\rightsquigarrow^* \text{(error)}\end{aligned}$$

Unfortunately, we cannot write the term $\langle y\rangle$, which has a free variable y; the compiler does not allow such a term. The definition of `spower` is OK, so its usage should be blamed.

Usage of staged power function (2nd **buggy** version):

$$\lambda y.\ (\texttt{spower}\ y\ 3) \rightsquigarrow^* \lambda y.\ (\texttt{spower}\ y\ 3)(???)$$

Still, there is a problem – the body of a function is not evaluated, so the code is not generated.

Usage of staged power function (3rd version):

$$\begin{aligned}\langle\lambda y.\ {\sim}(\texttt{spower}\ \langle y\rangle\ 3)\rangle &\rightsquigarrow^* \langle\lambda y.\ {\sim}\langle y * (y * (y * 1))\rangle\rangle \\ &\rightsquigarrow^* \langle\lambda y.\ y * (y * (y * 1))\rangle\end{aligned}$$

Good! We succeeded in generating a straightforward (not recursive) program for computing the power of the argument.

Executing the generated code:

$$\begin{aligned}\texttt{run}\ \langle\lambda y.\ y * (y * (y * 1))\rangle &\rightsquigarrow^* \lambda y.\ y * (y * (y * 1)) \\ (\texttt{run}\ \langle\lambda y.\ y * (y * (y * 1))\rangle)\ 5 &\rightsquigarrow^* 5 * (5 * (5 * 1)) \rightsquigarrow^* 125\end{aligned}$$

Type System (型システム)

For multi-stage languages, type system plays a crucial role, to exclude

- a code-generator which does not typecheck,
- a code-generator which typechecks, but generates a non-well-formed code (syntax error),
- a code-generator which typechecks, but generates a code which does not typecheck, and
- a code-generator which typechecks, but generates a code which has free variables.

To simplify things, we study a type system for the run-free subset of the language.

A type is defined as follows:

$$\begin{aligned}\sigma, \tau ::= &\ \texttt{int} \mid \texttt{bool} \mid \sigma \to \tau \mid (\sigma_1, \sigma_2, \cdots, \sigma_n) \to \tau \\ &\mid \langle\sigma\rangle\end{aligned}$$

$\langle\sigma\rangle$ is the type for codes whose "contents" have type σ. For instance, $\langle 1+2\rangle$ has type $\langle\texttt{int}\rangle$, and $\langle\lambda x.\ x+1\rangle$ has type $\langle\texttt{int} \to \texttt{int}\rangle$.

We need to take care of the level of a variable and a term. For instance, $\lambda x.\ (x, \langle x \rangle)$ is an error – since x and $\langle x \rangle$ cannot live in the same world (level).

We write $(x : \sigma)^n$ to express x has level n where n is a natural number (non-negative integers, including 0). A typing context Γ is a list of such forms.

Judgements are also extended:

$$\Gamma \vdash^n e : \sigma$$

n denotes the level of the term e. For instance, if the term $\texttt{let}\ x = \langle y + 1 \rangle\ \texttt{in}\ z$ has the level 0, then the levels of x, y, z, resp., are 0, 1, 0, and their types are $\langle \texttt{int} \rangle$, $\texttt{int}$, and $\langle \texttt{int} \rangle$, resp.

All typing rules are extended straightforwardly, for instance:

$$\frac{(x : \sigma)^n \text{ is an element of } \Gamma}{\Gamma \vdash^n x : \sigma}\ \text{var} \qquad \frac{(m \text{ is an integer constant})}{\Gamma \vdash^n m : \texttt{int}}\ \text{const1} \qquad \frac{(b \text{ is a boolean constant})}{\Gamma \vdash^n b : \texttt{bool}}\ \text{const2}$$

$$\frac{\Gamma \vdash^n e_1 : \texttt{int} \quad \Gamma \vdash^n e_2 : \texttt{int}}{\Gamma \vdash^n e_1 + e_2 : \texttt{int}}\ \text{plus} \qquad \frac{\Gamma \vdash^n e_1 : \texttt{int} \quad \Gamma \vdash^n e_2 : \texttt{int}}{\Gamma \vdash^n e_1 - e_2 : \texttt{int}}\ \text{minus}$$

$$\frac{\Gamma, (x : \sigma)^n \vdash^n e : \tau}{\Gamma \vdash^n \lambda(x).\ e\ :\ \sigma \to \tau}\ \text{fun} \qquad \frac{\Gamma \vdash^n e : \sigma \to \tau \quad \Gamma \vdash^n f : \sigma}{\Gamma \vdash^n e\ f : \tau}\ \text{apply}$$

$$\frac{\Gamma, (x_1 : \sigma_1)^n, \cdots, (x_n : \sigma_k)^n \vdash^n e : \tau}{\Gamma \vdash^n \lambda(x_1, \cdots, x_k).\ e\ :\ \sigma \to \tau}\ \text{fun2} \qquad \frac{\Gamma \vdash^n e : \sigma \to \tau \quad \Gamma \vdash^n f : \sigma}{\Gamma \vdash^n e\ f : \tau}\ \text{apply2}$$

$$\frac{\Gamma \vdash^n e_1 : \texttt{bool} \quad \Gamma \vdash^n e_2 : \sigma \quad \Gamma \vdash^n e_3 : \sigma}{\Gamma \vdash^n \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \sigma}\ \text{if}$$

$$\frac{\Gamma, (f : \sigma \to \tau)^n, (x : \sigma)^n \vdash^n e : \tau}{\Gamma \vdash^n \texttt{fix}\ f(x).\ e\ :\ \sigma \to \tau}\ \text{fix}$$

Of course, the only interesting things happen in the following rules:

$$\frac{\Gamma \vdash^{n+1} \quad e : \tau}{\Gamma \vdash^n \quad \langle e \rangle : \langle \tau \rangle}\ \text{brackets} \qquad \frac{\Gamma \vdash^n \quad e : \langle \tau \rangle}{\Gamma \vdash^{n+1} \quad {\sim} e : \tau}\ \text{escape} \qquad \frac{\Gamma \vdash^n \quad e : \langle \tau \rangle}{\Gamma \vdash^n \quad \texttt{run}\ e : \tau}\ \text{run(*)}$$

The asterisk (*) in the run rule indicates that its a simplified rule, which may not be sound.

If $\vdash^0 e : \tau$ is derived using the rules above, then we say e is a (complete) program. (Note Γ should be empty, which means e does not have free variables. The level n should be 0, which means e is a level-0 term. All terms of level$\rangle$0 are premature (incomplete) programs.)

Example of type derivation:

$$\texttt{spower} = \lambda x.\ \texttt{fix}\ f(n).\ \texttt{if}\ n = 0\ \texttt{then}\ \langle 1 \rangle\ \texttt{else}\ \langle {\sim}x * {\sim}(f(n-1)) \rangle$$

Let $\Gamma = (x : \langle \texttt{int} \rangle)^0, (f : \texttt{int} \to \langle \texttt{int} \rangle)^0, (n : \texttt{int})^0$.

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma \vdash^0 n = 0 : \texttt{bool}\end{array} \quad \dfrac{\dfrac{}{\Gamma \vdash^1 1 : \texttt{int}}}{\Gamma \vdash^0 \langle 1 \rangle : \langle \texttt{int} \rangle} \quad \begin{array}{c}\vdots\\ \Gamma \vdash^0 \langle {\sim}x * {\sim}(f(n-1)) \rangle : \langle \texttt{int} \rangle\end{array}}{\Gamma \vdash^0 \texttt{if}\ n = 0\ \texttt{then}\ \langle 1 \rangle\ \texttt{else}\ \langle {\sim}x * {\sim}(f(n-1)) \rangle : \langle \texttt{int} \rangle}}{(x : \langle \texttt{int} \rangle)^0 \vdash^0 \texttt{fix}\ f(n).\ \texttt{if}\ n = 0\ \texttt{then}\ \langle 1 \rangle\ \texttt{else}\ \langle {\sim}x * {\sim}(f(n-1)) \rangle : \texttt{int} \to \langle \texttt{int} \rangle}}{\vdash^0 \lambda x.\ \texttt{fix}\ f(n).\ \texttt{if}\ n = 0\ \texttt{then}\ \langle 1 \rangle\ \texttt{else}\ \langle {\sim}x * {\sim}(f(n-1)) \rangle : \langle \texttt{int} \rangle \to (\texttt{int} \to \langle \texttt{int} \rangle)}$$

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash^0 x : \langle \texttt{int} \rangle}{\Gamma \vdash^1 {\sim}x : \texttt{int}} \quad \begin{array}{c}\vdots\\ \Gamma \vdash^1 {\sim}(f(n-1)) : \texttt{int}\end{array}}{\Gamma \vdash^1 {\sim}x * {\sim}(f(n-1)) : \texttt{int}}}{\Gamma \vdash^0 \langle {\sim}x * {\sim}(f(n-1)) \rangle : \langle \texttt{int} \rangle}$$

Exercise 1. (練習問題)

(1) Fill the missing part of the above type derivation (for staged power).

(2) Consider the terms in Page 1 of this text. Try to type the “correct” terms and explain why the erroneous terms do not have types.

(3) (optional) Derive the type for the following one (completed staged power function):

$$\texttt{run}\ \langle \lambda y.\ {\sim}(\texttt{spower}\ \langle y \rangle\ 3) \rangle$$

Can you type the following one ?

$$\lambda x.\ \texttt{run}\ \langle \lambda y.\ {\sim}(\texttt{spower}\ \langle y \rangle\ x) \rangle$$

(4) (optional) The run rule is not completely satisfactory, since a code-generator which generates a code with free variables may be run (executed), which is obviously an error.

Can you construct such an example ?

(5) Type the following variant of staged power. Compare the generated codes with the one generated by the previous one.

$$\texttt{spower} = \texttt{fix}\ f(n).\ \texttt{if}\ n = 0\ \texttt{then}\ \langle \lambda x.\ 1 \rangle\ \texttt{else}\ \langle \lambda x.\ x * ({\sim}(f(n-1))x) \rangle$$