Semantics, WS 2005 – Assignment 4

Prof. Dr. Gert Smolka, Dipl.-Inform. Andreas Rossberg
http://www.ps.uni-sb.de/courses/sem-ws05/

Recommended reading: Types and Programming Languages, Chapter 8–9, Section 11.11

We consider the simply typed language SL whose abstract syntax and typing relation are defined as follows:

$$
\begin{aligned}
T \in Ty &= T \to T \mid Bool \mid Int \\
x \in Var & \\
t \in Ter &= x \mid \lambda x{:}T.t \mid t\ t \mid \mathit{fix}\ t \\
&\quad \mid \mathit{false} \mid \mathit{true} \mid \mathit{if}\ t\ \mathit{then}\ t\ \mathit{else}\ t \\
&\quad \mid 0 \mid t+1 \mid t-1 \mid t=0 \\
v \in Val \subseteq Ter &= \lambda x{:}T.t \mid \mathit{false} \mid \mathit{true} \mid 0 \mid v+1 \\
\Gamma \in TE &= Var \rightharpoonup Ty
\end{aligned}
$$

$$
\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma[x := T] \vdash t : T'}{\Gamma \vdash \lambda x{:}T.t : T \to T'} \qquad \frac{\Gamma \vdash t_1 : T \to T' \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : T'} \qquad \frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \mathit{fix}\ t : T}
$$

$$
\frac{}{\Gamma \vdash \mathit{false} : \mathit{Bool}} \qquad \frac{}{\Gamma \vdash \mathit{true} : \mathit{Bool}} \qquad \frac{\Gamma \vdash t_0 : \mathit{Bool} \qquad \Gamma \vdash t_1 : T \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathit{if}\ t_0\ \mathit{then}\ t_1\ \mathit{else}\ t_2 : T}
$$

$$
\frac{}{\Gamma \vdash 0 : \mathit{Nat}} \qquad \frac{\Gamma \vdash t : \mathit{Nat}}{\Gamma \vdash t+1 : \mathit{Nat}} \qquad \frac{\Gamma \vdash t : \mathit{Nat}}{\Gamma \vdash t-1 : \mathit{Nat}} \qquad \frac{\Gamma \vdash t : \mathit{Nat}}{\Gamma \vdash t=0 : \mathit{Bool}}
$$

Exercise 4.1: Procedures in SL Find closed terms as follows:

(a) $\mathit{plus} : \mathit{nat} \to \mathit{nat} \to \mathit{nat}$ such that *plus* describes a procedure that adds two numbers.

(b) $\mathit{times} : \mathit{nat} \to \mathit{nat} \to \mathit{nat}$ such that *times* describes a procedure that multiplies two numbers.

(c) $\mathit{fac} : \mathit{nat} \to \mathit{nat}$ such that *fac* describes a procedure that yields the factorial of a number (i.e. $n!$).

Exercise 4.2: Reduction Relation The reduction relation for SL is defined by inference rules that can be modularized with respect to the different syntactic forms. Furthermore, we distinguish between *descent rules* and *proper reduction rules*. The syntactic form *fix t*, for instance, contributes one descent rule and one proper reduction rule:

$$\frac{t \to t'}{\mathit{fix}\ t \to \mathit{fix}\ t'} \qquad \frac{}{\mathit{fix}(\lambda x{:}T.t) \to t[x := \mathit{fix}(\lambda x{:}T.t)]}$$

(a) Which syntactic forms don't have descent rules?

(b) Which syntactic forms have more than one descent rule?

(c) Which syntactic forms don't have proper reduction rules?

(d) Which syntactic forms have more than one proper reduction rule?

(e) State all descent rules.

(f) State all proper reduction rules.

Exercise 4.3: Key Properties State the following properties for SL:

(a) Uniqueness of types.

(b) Progress.

(c) Preservation.

(d) Normalization.

(e) Functionality of reduction relation.

(f) Substitution lemma for reduction relation.

(g) Substitution lemma for typing relation.

Exercise 4.4: Big-Step Semantics The *evaluation relation* is a relation between terms and values that can be described as follows: $t \Downarrow v \Leftrightarrow t \to^* v$. It is possible to define the evaluation relation independently of the reduction relation by means of inference rules (one speaks of a so-called *big-step semantics*, see [Pierce, Exercise 3.5.17]). Here are the inference rules for the syntactic forms *fix t* and $t - 1$:

$$\frac{t \Downarrow v' \qquad v' = \lambda x{:}T.t' \qquad t'[x := \mathit{fix}\ v'] \Downarrow v}{\mathit{fix}\ t \Downarrow v} \qquad \frac{t \Downarrow 0}{t - 1 \Downarrow 0} \qquad \frac{t \Downarrow v + 1}{t - 1 \Downarrow v}$$

Find the inference rules for the other syntactic forms.

Exercise 4.5: Environment Semantics Often it is useful to formulate the dynamic semantics of a programming language with values that are not represented as terms. For SL, for instance, numeric values may be represented as numbers. Giving values their own representation has the consequence that procedure application cannot be realized by substitution and that procedures cannot be represented as terms. Instead, one uses *value environments* $V \in VE = Var \rightharpoonup Val$ mapping variables to values and represents procedures as so-called *closures* $\langle t, V \rangle$ consisting of an abstraction (the so-called *code*) and a value evironment. We want to formulate a dynamic semantics with value environments for an untyped version of SL. We work with values of the form

$$
\begin{aligned}
& n \in \mathbb{N} \\
& v \in Val \;=\; n \mid \langle \lambda\, x\,.\, t,\; V \rangle \mid \langle \mathit{fix}\, \lambda\, x_1\,.\, \lambda\, x_2\,.\, t,\; V \rangle \\
& V \in VE = Var \rightharpoonup Val
\end{aligned}
$$

and an evaluation relation that contains all triples $\langle V, t, v \rangle$ such that the term t evaluates in the value environment V to the value v. We write $V \vdash t \Rightarrow v$ if $\langle V, t, v \rangle$ is an element of the evaluation relation. The evaluation can be defined by inference rules. Here are the inference rules for variables, abstractions and fix:

$$
\frac{V(x) = v}{V \vdash x \Rightarrow v} \qquad \frac{t = \lambda\, x\,.\, t'}{V \vdash t \Rightarrow \langle t, V \rangle} \qquad \frac{t = \mathit{fix}\, \lambda\, x_1\,.\, \lambda\, x_2\,.\, t'}{V \vdash t \Rightarrow \langle t, V \rangle}
$$

(a) Give a well-typed term t such that *fix t* cannot be evaluated with the rule given for *fix* although its reduction terminates with a value in the small-step semantics.

(b) State the inference rules for the other syntactic forms of SL.

Exercise 4.6: Implementation in Standard ML We implement the abstract syntax of SL in Standard ML as follows:

```
type var = int
datatype ty = Arrow of ty * ty | Bool | Int
datatype ter = V of var | L of var * ty * ter | A of ter * ter
  | Fix of ter | False | True | If of ter * ter * ter
  | O | S of ter | P of ter | Z of ter
```

(a) Write a procedure *check* : (*var* → *ty*) → *ter* → *ty* that computes the type of a term in a type environment. If the term is ill-typed, *check* should throw the exception *Error*. The procedure realising the type environment throws *Error* for unbound variables.

(b) Write a procedure *eval* : (*var* → *value*) → *ter* → *value* that tries to evaluates a term in a value environment. Represent values as follows:

```
datatype value = N of int | C of ter * (var -> value)
```