Semantics, WS 2005 – Assignment 9

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

Recommended reading: Types and Programming Languages, Chapter 29–30

In this assignment we will explore the higher-order polymorphic lambda calculus F_ω. F_ω is obtained from F by adding procedures taking types to types, and by considering such procedures again as types. Consequently, there are different *kinds* of types, where kinds are formalized as follows:

$$k \in Kind \;=\; \star \;\mid\; k \to k$$

Types of kind $\star$ are called *proper types* and act as types of terms. Types of kind $k \to k'$ are called *type constructors* and act as procedures taking types of kind k to types of kind k'. The syntax of types is defined as follows:

$$\begin{aligned} &X \in TVar \\ &T \in Ty \;=\; X \mid T \to T \mid \forall X{:}k.T \mid \lambda X{:}k.T \mid T\,T' \end{aligned}$$

The well-kinded types are obtained with a kinding relation $\Gamma \vdash T : k$ where the environment Γ maps type variables to kinds. The terms of F_ω are as in F, except that polymorphic procedures can take types of any kind as argument:

$$\begin{aligned} &x \in Var \\ &t \in Ter \;=\; x \mid \lambda x{:}T.t \mid t\,t \mid \lambda X{:}k.t \mid t\,T \end{aligned}$$

The typing relation $\Gamma \vdash t : T$ takes environments Γ mapping type variables to kinds and term variables to types. To provide for the evaluation of type constructors, the following typing rule is added:

$$\textbf{Eq}\quad \frac{\Gamma \vdash t : T \qquad T \equiv T' \qquad \Gamma \vdash T : k \qquad \Gamma \vdash T' : k}{\Gamma \vdash t : T'}$$

There are several possibilities to define *type equivalence* $T \equiv T'$. We use

$$T \equiv T' \;\overset{\text{def}}{\iff}\; T \text{ and } T' \text{ have the same } \beta\text{-normal form}$$

The *β-normal form* of a type is obtained by applying the β-rule

$$(\lambda X{:}k.T)T' \;\to\; T[X := T']$$

as long as it is applicable and wherever it is applicable. The free applicability of the

β-rule can be specified by the following reduction contexts:

$$R = \bullet \mid R \to T \mid T \to R \mid \forall X{:}k.R \mid \lambda X{:}k.R \mid R\,T \mid T\,R$$

We write $\lambda X.t$, $\lambda X.T$ and $\forall X.T$ as abbreviations for the terms $\lambda X:\star.t$, $\lambda X:\star.T$ and $\forall X{:}\star.T$.

Exercise 9.1: Sorting Relations

(a) State the inference rules defining the kinding relation $\Gamma \vdash T : k$ of F_ω. Assume $\Gamma \in \mathit{TVar} \rightharpoonup \mathit{Kind}$.

(b) State the inference rules defining the typing relation $\Gamma \vdash t : T$ of F_ω. Assume that Γ maps type variables to kinds and term variables to types.

(c) Consider environments Γ mapping type variables to kinds and term variables to types. An environment is *well-formed* if the types of the term variables are well-kinded with respect to the kinds of the type variables. Give a formal definition of well-formed finite environments. Use recursion on the size of the environment (number of variables introduced).

Exercise 9.2: Polymorphic Lists We implement polymorphic lists in F_ω.

(a) Give a type $\mathit{List} : \star \to \star$ that implements list over an arbitrary type.

(b) Give the types of the polymorphic operations *nil*, *cons* and *foldl*.

(c) Implement the operations *nil*, *cons* and *foldl*.

(d) Write a procedure $\mathit{length} : \forall X.\ \mathit{List}\ X \to \mathit{Nat}$ that yields the length of a list. You may use *Nat*, *zero*, *succ*, *List*, and *foldl* as names for the corresponding objects.

Exercise 9.3: ADTs in SML We consider the implementation of ADTs for variants (i.e., elements of binary sum types) in SML. We will use the names *Var*, *left*, *right*, *scase* for the abstract type and the operations of the ADT.

(a) Declare a structure *Var* that implements a plain ADT providing variants carrying integers or Booleans.

(b) Declare a functor *GVar* that implements a generic ADT providing variants.

(c) Declare a structure *Var* that implements a plain ADT providing variants carrying reals or integers. Use the functor *GVar*.

(d) Declare a structure *PVar* that implements a polymorphic ADT providing variants.

Hint: www.ps.uni-sb.de/courses/sem-ws05/assignments/adt.sml

Exercise 9.4: ADTs in F_ω We will implement several ADTs for lists in F_ω. The signature of an ADT is represented with a type

$$\begin{aligned} Sig &: \star \to \star \\ &= \lambda Z.\ \forall X{:}k.\ T_1 \to \cdots \to T_n \to Z \end{aligned}$$

where X represents the abstract type and $T_1, \ldots, T_n$ are the types of the operations. The variable Z must not occur free in $T_1, \ldots, T_n$. The ADT can then be implemented as a procedure

$$\begin{aligned} imp &: \forall Z.\ Sig\ Z \to Z \\ &= \lambda Z.\ \lambda f{:}Sig\ Z.\ \ f\ T\ t_1\ \ldots\ t_n \end{aligned}$$

where T and $t_1, \ldots, t_n$ are the implementations of the abstract type and the operations. Finally, a term t that uses the ADT and yields a result of type T can be realized with the application

$$imp\ T\ (\lambda X{:}k.\lambda f_1{:}T_1.\ \cdots\ \lambda f_n{:}T_n.\ t)$$

where X and $f_1, \ldots, f_n$ act as names for the abstract type and the operations.

Now consider the signature of a plain ADT that implements lists over *Nat*:

$$\begin{aligned} L &: \star \\ nil &: L \\ cons &: Nat \to L \to L \\ foldl &: L \to \forall Z.\ (Nat \to Z \to Z) \to Z \to Z \end{aligned}$$

(a) Give a type *Sig* that represents the signature of the ADT.

(b) Write a procedure *imp* that implements the ADT. Use the code from Exercise 9.2.

(c) Consider a generic ADT that implements Lists over a type given as parameter.

  (i) Give a type Sig' such that $Sig'\ T$ represents the signature of an ADT that implements lists over T.

  (ii) Write a procedure imp' such that $imp'\ T$ implements an ADT that implements lists over T.

(d) Consider an ADT that implements polymorphic lists as in Exercise 9.2.

  (i) Give the signature of the ADT.

  (ii) Give a type Sig'' that represents the signature of the ADT.

  (iii) Write a procedure imp'' that implements the ADT.

Exercise 9.5: Uniform Presentation We have already seen a uniform presentation of F, which represents terms and types uniformly as expressions,

$$\begin{aligned} x &\in \mathit{Var} \\ e \in \mathit{Exp} &= x \mid \lambda x{:}e.e \mid e\,e \mid \Pi x{:}e.e \mid \star \mid \Box \end{aligned}$$

and formalizes the respective notions through a sorting relation $\Gamma \vdash e : e'$ where $\Gamma \in \mathit{Var} \rightharpoonup \mathit{Exp}$. The same can be done for F_ω.

(a) Show how the kinds, types and terms of F_ω can be represented with expressions.

(b) State the sorting rules needed for F_ω. The following should be satisfied:

$$\begin{aligned} e \text{ is a kind} &\overset{\text{def}}{\iff} \emptyset \vdash e : \Box \\ e \text{ is a type} &\overset{\text{def}}{\iff} \exists \Gamma, e' :\ \Gamma \vdash e : e' \ \wedge\ e' \text{ is a kind} \\ e \text{ is a term} &\overset{\text{def}}{\iff} \exists \Gamma, e' :\ \Gamma \vdash e : e' \ \wedge\ e' \text{ is a type} \end{aligned}$$

(c) State the sorting rules needed for F_ω if the expressions represent variables in de Bruijn style:

$$\begin{aligned} n &\in \mathit{Var} = \mathbb{N} \\ e \in \mathit{Exp} &= n \mid \lambda\, e.e \mid e\,e \mid \Pi\, e.e \mid \star \mid \Box \end{aligned}$$

Hint: Consult www.ps.uni-sb.de/courses/sem-ws05/assignments/f.pdf.

Exercise 9.6: Implementation of F_ω We will implement F_ω using uniform syntax. As starting point, we take the implementation of F that we discussed last week (www.ps.uni-sb.de/courses/sem-ws05/assignments/f.sml).

(a) Write a procedure $\mathit{norm} : \mathit{exp} \to \mathit{exp}$ that yields the β-normal form of well-kinded types.

(b) Give an ill-kinded type that doesn't have a β-normal form.

(c) Modify the procedure $\mathit{check} : \mathit{exp}\ \mathit{list} \to \mathit{exp} \to \mathit{exp}$ so that it becomes a type checker for F_ω. This means that *check nil e* should yield

- the type of e if e is a closed and well-typed term.
- the kind of e if e is a closed and well-kinded type.
- $\Box$ if e is a kind.

In all other cases *check nil e* should raise an exception. Proceed as follows:

(i) Extend the rule for Π so that types of all kinds become admissible (in F only so-called proper types of kind $\star$ are admissible).

(ii) Modify the rules such that they put only β-normal types on the stack (the first argument of *check* implementing the type environment). Make sure that the procedure *norm* is only applied to expressions that have been established as well-kinded types or kinds before (to avoid non-termination). Hint: Only the rules for Π and λ put types on the stack.

(iii) Modify the rules such that check returns only β-normal expressions (under the provision that the stack contains only β-normal expressions). Hint: Only the rules for λ and applications need to be considered. If done right, the procedure *check* will contain exactly three applications of *norm*, one in each of the rules for λ, Π and applications.

(d) Write a procedure *verify* : *exp* → *exp* → *bool* such that *verify e s* returns *true* if and only if e is a closed term, s is a closed type, and e has type s.

(e) Type-check your answers to Exercises 9.2 and 9.4 with the procedure *verify*.

(f) Write a procedure *checkEnv* : *exp list* → *bool* that checks whether an environment is well-formed (see Exercise 9.1).

Exercise 9.7: $\beta\eta$-Type Equivalence If F is a type constructor $k \to k'$, then $\lambda X{:}k.FX$ is a different type constructor that behaves the same as F. However, with our definition of type equivalence, F and $\lambda X{:}k.FX$ are not equivalent. This can be fixed by defining type equivalence with respect to $\beta\eta$-normal forms, which are obtained with the β-rule and the so-called *η-rule*:

$$\lambda X{:}k.TX \;\to\; T \qquad \text{if } X \notin FV\ T$$

(a) Find a closed term that is ill-typed under β-type equivalence and well-typed under $\beta\eta$-type equivalence.

(b) Extend your implementation of F_ω (Exercise 9.6) so that it employs $\beta\eta$-type equivalence. Hint: Use *subst* to write a procedure that checks whether a variable occurs free in an expression.

Exercise 9.8: New Year's Challenge: Multiplicative closure in linear time Consider the following problem: Given natural numbers a, b, n such that $a, b \geq 2$, compute the first n elements of the set $M_{a,b} = \{\, a^k \cdot b^l \mid k, l \in \mathbb{N} \,\}$ in $O(n)$ time. For instance, if $a = 2$ and $b = 3$, the first 10 elements of $M_{a,b}$ are 1, 2, 3, 4, 6, 8, 9, 12, 16, 18. Try to come up with a algorithm and you will realize that the problem isn't as easy as it looks (due to the $O(n)$ requirement).

Here is a naive algorithm. Start with a set $S = \{1\}$ and iterate n-times as follows: Remove the smallest element x from the set and replace it with ax and bx. Upon termination, the removed elements are the first n elements of $M_{a,b}$. Here you can see what happens for $a = 2$, $b = 3$ and $n = 5$:

$$\{1\} \to \{2,3\} \to \{3,4,6\} \to \{4,6,9\} \to \{6,8,9,12\} \to \{8,9,12,18\}$$

To improve the algorithm, we implement the set S with two queues Q_a and Q_b, where Q_a takes ax and Q_b takes bx when x is removed. Initially, both queues contain just 1. Since the queues contain their elements in strictly ascending order, the smallest element of S always appears as the first element of at least one of the queues. Now it is routine to implement a linear time algorithm with imperative data structures. An implementation with functional data structures requires a clever data structure for the queues.

We implement the queues with streams (see Assignment 7) as given by the signature:

$$\mathit{stream} : (\mathit{unit} \to \alpha * \alpha\ \mathit{stream}) \to \alpha\ \mathit{stream}$$
$$\mathit{decons} : \alpha\ \mathit{stream} \to \alpha * \alpha\ \mathit{stream}$$

To obtain a linear time algorithm, we implement streams as follows:

```
datatype 'a state = E of 'a * 'a stream | O of unit -> 'a * 'a stream
withtype 'a stream = 'a state ref
fun stream f = ref(O f)
fun decons s = case s of ref(E p) => p
                       | ref(O f) => (s:=E(f()); decons s)
```

This avoids recomputing $f()$ each time *decons* is applied to a stream.

In the following, use streams only according to the above signature, which hides their imperative implementation.

(a) Write procedures $\mathit{head} : \alpha\ \mathit{stream} \to \alpha$, $\mathit{tail} : \alpha\ \mathit{stream} \to \alpha\ \mathit{stream}$ and $\mathit{take} : \mathit{int} \to \alpha\ \mathit{stream} \to \alpha\ \mathit{list}$ for streams.

(b) Write a procedure $\mathit{times} : \mathit{int} \to \mathit{int\ stream} \to \mathit{int\ stream}$ such that $\mathit{times}\ k$ yields a stream $kx_1, kx_2, kx_3, \ldots$ when applied to a stream $x_1, x_2, x_3, \ldots$.

(c) Write a procedure $\mathit{merge} : \mathit{int\ stream} \to \mathit{int\ stream} \to \mathit{int\ stream}$ that merges two strictly ascending streams into one strictly ascending stream. A stream $x_1, x_2, x_3, \ldots$ is strictly ascending if $x_1 < x_2 < x_3 < \ldots$.

(d) Write a procedure $\mathit{rstream} : \alpha \to (\alpha\ \mathit{stream} \to \alpha\ \mathit{stream}) \to \alpha\ \mathit{stream}$ such that $\mathit{rstream}\ x\ f$ yields a stream s such that $s = x :: f\ s$ if f is plain. A procedure $f : \alpha\ \mathit{stream} \to \alpha\ \mathit{stream}$ is called *plain* if, for all n and s, accessing the n-th element of the result stream $f\ s$ will at most access the first n elements of the argument stream s.

(e) Declare a constant time procedure $mul : int \to int \to int\ stream$ such that $mul\ a\ b$ yields a strictly ascending stream whose elements are the elements of $M_{a,b}$.

(f) Convince yourself that $take\ n\ (mul\ a\ b)$ runs in $O(n)$ time.

(g) Our problem simplifies the problem of computing the so-called Hamming numbers. Read http://en.wikipedia.org/wiki/Hamming_number and write a procedure that yields the n-th Hamming number in $O(n)$ time.

(h) A general formulation of the problem is as follows. Let $X \subseteq \mathbb{N}$. Then the *multiplicative closure of X* is the least set $MC[X]$ such that

$$MC[X] = \{1\} \cup \{\, mx \mid m \in MC[X] \land x \in X \,\}$$

Write a procedure $mc : int \to int\ list \to int\ list$ that yields, in ascending order, the first n elements of the multiplicative closure of a finite, nonempty set that doesn't contain 0 or 1. The procedure should run in $O(kn)$ where k is the size of the set.