Semantics, WS 2003: Solutions for assigment 6

Prof. Dr. Gert Smolka, Dipl.-Inform. Guido Tack

Exercise 6.1: State Big-step reduction rules:

$$\frac{t_1|\mu \Downarrow \lambda x:T.t|\mu' \quad t_2|\mu' \Downarrow v_2|\mu'' \quad t[x := v_2]|\mu'' \Downarrow v|\mu'''}{t_1\ t_2 \mid \mu \Downarrow v|\mu'''} \quad \text{(E-APP)}$$

$$\frac{t|\mu \Downarrow v|\mu' \quad l \notin \text{dom}(\mu')}{\text{ref } t|\mu \Downarrow l|\mu'[l := v]} \quad \text{(E-REF)}$$

$$\frac{t|\mu \Downarrow l|\mu' \quad \mu'(l) = v}{!t|\mu \Downarrow v|\mu'} \quad \text{(E-DEREF)}$$

$$\frac{t_1|\mu \Downarrow l_1|\mu' \quad t_2|\mu' \Downarrow v|\mu''}{t_1 := t_2|\mu \Downarrow \text{unit}|\mu[l := v]} \quad \text{(E-ASSIGN)}$$

Exercise 6.2: Curry-Howard in SML

(a)

```
structure CurryHoward :> CURRYHOWARD =
  struct
    datatype ('a, 'b) sum = INL of 'a | INR of 'b
    type n = unit
    fun null _ = raise Empty
    fun dneg _ = raise Empty
  end
```

(b) Example (from exercise 5.5, d):

```
fn f => dneg (fn g => g(INL (fn x => g (INR (f x)))))
```

SML infers the following type:

```
val ('a, 'b) it = fn : ('a -> 'b) -> ('a -> n, 'b) sum
```

(c) A proof may look like this:

```
fn (nx, ny) => fn (x,y) => (nx x)
```

SML infers

```
val ('a, 'b, 'c, 'd) it = fn : ('a -> 'b) * 'c -> 'a * 'd -> 'b
```

as its type. This is (by construction of SML's type checking algorithm) the most general type of this term. Interpreted logically, this means that this term proves a *family* of logical formulas: The original type is one instance, but the term is also a proof of e.g. the following formula: $(X \longrightarrow Z) \wedge (Y \longrightarrow Z) \;\longrightarrow\; ((X \wedge Y) \longrightarrow Z)$.

Exercise 6.3: Big-step semantics with error The new rules are:

$$\frac{}{\texttt{error} \Downarrow \texttt{error}}$$

$$\frac{t_1 \Downarrow \texttt{error}}{t_1\ t_2 \Downarrow \texttt{error}}$$

$$\frac{t_1 \Downarrow v \qquad t_2 \Downarrow \texttt{error}}{t_1\ t_2 \Downarrow \texttt{error}}$$

Exercise 6.4: Recursion with state and error

(a) Assuming syntactic sugar for let, the following term diverges:

let
  $l \;=\; \text{ref}(\lambda x:\text{Unit}.x)$
in
  $l := \lambda y:\text{Unit}.(!l)\ y;$
  $(!l)\ \text{unit}$

(b) let
  $\mathit{fixref} = \text{ref}(\lambda f:((T_0 \longrightarrow T_0) \longrightarrow T_0 \longrightarrow T_0).\lambda x:T_0.x)$
in
  $\mathit{fixref} := \lambda f:((T_0 \longrightarrow T_0) \longrightarrow T_0 \longrightarrow T_0).\lambda x:T_0.f((!\mathit{fixref})f)\ x;$
  $!\mathit{fixref}$

(c) let
  $\mathit{fixref} = \text{ref}(\lambda f:((T_0 \longrightarrow T_1) \longrightarrow T_0 \longrightarrow T_1).\lambda x:T_0.\text{error as } T_1)$
in
  $\mathit{fixref} := \lambda f:((T_0 \longrightarrow T_1) \longrightarrow T_0 \longrightarrow T_1).\lambda x:T_0.f((!\mathit{fixref})f)\ x;$
  $!\mathit{fixref}$

(d)
```
val fix = fn f =>
    let
        val fixref = ref (fn f => fn x => raise Empty)
        val fix' = fn f => fn x => (f ((!fixref) f) x)
    in
        fixref := fix';
        fix' f
    end
```

Exercise 6.5: Type inhabitation The proof is by induction on the structure of types.

For $T = \text{Unit}$, we have that $\emptyset \vdash \text{unit} : \text{Unit}$.

For $T = T_1 \longrightarrow T_2$, we know by induction hypothesis that there is a term t with $\emptyset \vdash t : T_2$. Then it follows from the typing rule for abstraction that $\emptyset \vdash (\lambda x : T_1 . t) : T$.

For $T = T_1 \times T_2$, we know by induction hypothesis that there exist terms t_1 and t_2 such that $\emptyset \vdash t_1 : T_1$ and $\emptyset \vdash t_1 : T_1$. Hence, with the typing rule for products, $\emptyset \vdash \{t_1, t_2\} : T$.