Computer Science 131, Fall 2000
Assignment 7: Polymorphism, Subtyping, and λ-Calculus

Out: Friday, November 10

Due: Friday, November 17, 5:00pm

This assignment involves no programming. Your written answers must be given directly to the professor (or put under his office door, 1253 Olin). *Your work must be clearly legible.* If you cannot write neatly, use LaTeX or otherwise typeset the proofs.

If you submit your solution late, be sure to mark it with the date and time that it was handed in.

1 Subtyping (20%)

Give the most general conditions on t_1 and t_2 for each of the following subtyping judgments to be safe: whether $t_1 \preceq t_2$ or $t_1 \succeq t_2$ or $t_1 = t_2$. Give a brief and clear explanation why your answer is correct.

1. t_1 `list` $\preceq t_2$ `list`
2. t_1 `foo` $\preceq t_2$ `foo`, defined by `type 'a foo = 'a -> 'a`
3. t_1 `bar` $\preceq t_2$ `bar`, defined by `type 'a bar = ('a -> int) -> int`
4. t_1 `cont` $\preceq t_2$ `cont`

2 Fixed Points (10%)

In class you saw the Y combinator for finding fixed points, which satisfies $YM \longleftrightarrow^*_\beta M(YM)$ for any term M. There are actually many terms other than Y which compute fixed points. The Turing fixed-point combinator Θ is defined to be

$$(\lambda x.\lambda y.y(xxy))(\lambda x.\lambda y.y(xxy))$$

Prove that for any term M we have $\Theta M \rightarrow^*_\beta M(\Theta M)$.

3 Predecessor (20%)

For more practice working with λ-calculus, in this problem you will prove that the definition of predecessor that you saw in class is correct.

1. Show that $\overline{0}\ M_1\ M_2 \longleftrightarrow^*_\beta M_1$ and that $\overline{n+1}\ M_1\ M_2 \longleftrightarrow^*_\beta M_2(\overline{n}\ M_1\ M_2)$.

2. In class the predecessor function was defined as

$$\begin{array}{lcl} \mathit{pred} & = & \lambda n.(\mathit{pred}'\ n).1 \\ \mathit{pred}' & = & \lambda m.m\ \langle\,\overline{0}\,,\,\overline{0}\,\rangle\ (\lambda p.\langle p.2, \mathit{succ}(p.2)\rangle) \end{array}$$

   Prove that $\forall m \geq 0.\ \mathit{pred}'\ \overline{m+1} \longleftrightarrow^*_\beta \langle\,\overline{m}\,,\,\overline{m+1}\,\rangle$.

3. Prove that $\forall m \geq 0.\ \mathit{pred}\ \overline{m+1} \longleftrightarrow^*_\beta \overline{m}\,$.

4 Lambda Calculus Encodings (30%)

For this problem you will devise an encoding for lists within the untyped λ-calculus. Recall that a list is either empty or it has a head (first element) and a tail (a list containing the rest of the elements, possibly empty).

1. Find a lambda term *nil* to represent the empty list and a lambda term *cons* such that $\mathit{cons}\ M\ N$ represents the non-empty list with head M and with tail N. Then a finite list $[M_1, \ldots, M_n]$ would be represented as $\mathit{cons}\ M_1\ (\mathit{cons}\ M_2\ (\cdots(\mathit{cons}\ M_n\ \mathit{nil})\cdots))$. Explain how to define the following functions for your encoding:

   - The function *isnil* that returns *tt* if given *nil* and returns *ff* if given a non-empty list.
   - The function *hd* that returns the head of a non-empty list.
   - The function *tl* that returns the tail of a non-empty list.

   Verify that

$$\begin{array}{l} \mathit{isnil}\ \ \mathit{nil} \longleftrightarrow^*_\beta \mathit{tt} \\ \mathit{isnil}\ (\mathit{cons}\ M_1\ M_2) \longleftrightarrow^*_\beta \mathit{ff} \\ \mathit{hd}\ (\mathit{cons}\ M_1\ M_2) \longleftrightarrow^*_\beta M_1 \\ \mathit{tl}\ (\mathit{cons}\ M_1\ M_2) \longleftrightarrow^*_\beta M_2 \end{array}$$

2. Find a lambda term *length* such that $\mathit{length}\ M \longleftrightarrow^*_\beta \overline{n}$ when M represents a finite list with n elements.

3. Find a lambda term *zeros* that represents an infinite list whose elements are all $\overline{0}$. Show that it satisfies $\mathit{hd}(\mathit{tl}^{(n)}\ \mathit{zeros}) \longleftrightarrow^*_\beta \overline{0}$ for every $n \geq 0$.

5 Theorems for Free (20% or more)

As mentioned briefly in class, in languages with purely parametric polymorphism one can prove interesting properties about the behavior of a function based only its type. The purpose of this problem is to show how the *Parametricity Theorem* can be used to derive these properties.

First we need to define the notion of a *logical relation*. This is actually a family of relations indexed by types which "fit together" properly. Logical relations are very useful tools for studying typed languages.

For this problem, we will simplify matters by considering a NQSML-like language with explicit polymorphism (the `Fn` α `=>` `e` and `e[t]` constructs) but without recursion, exceptions, assignment, or other side-effects. It follows that every well-typed expression evaluates to a unique value; there are no non-terminating programs. To keep the notation managable, for any closed expression e let $\lfloor e \rfloor$ denote the value to which e evaluates. (The fact that every program terminates is not immediately obvious; interestingly, the nicest proof of this fact also uses logical relations.) Then we have a language whose type system looks like:

$$\begin{array}{rl} t ::= & \texttt{Int} \\ | & \texttt{Bool} \\ | & t_1\texttt{->}t_2 \\ | & t_1 \times t_2 \\ | & t\ \texttt{list} \\ | & \alpha \\ | & \forall\alpha.t \end{array}$$

For every type t, we can define a relation $\texttt{Rel}_\texttt{V}(t)$ between closed values and a relation $\texttt{Rel}(t)$ between closed expressions. The definition of the latter is easy: two expressions are related at a type if they evaluate to related values:

$$(e_1, e_2) \in \texttt{Rel}(t) \quad \text{if } (\lfloor e_1 \rfloor, \lfloor e_2 \rfloor) \in \texttt{Rel}_\texttt{V}(t)$$

The relation on values can be intuitively considered as a definition for when two values "act the same". For integers and booleans, this is just simple equality on constants.

$$\begin{array}{ll} (\overline{n_1}, \overline{n_2}) \in \texttt{Rel}_\texttt{V}(\texttt{Int}) & \text{if } n_1 = n_2 \\ (\overline{b_1}, \overline{b_2}) \in \texttt{Rel}_\texttt{V}(\texttt{Bool}) & \text{if } b_1 = b_2 \end{array}$$

At a product type we consider two pairs to be equivalent if their corresponding components are equivalent

$$((v_1, v_2), (v_1', v_2')) \in \texttt{Rel}_\texttt{V}(t_1 \times t_2) \quad \text{if } (v_1, v_1') \in \texttt{Rel}_\texttt{V}(t_1) \text{ and } (v_2, v_2') \in \texttt{Rel}_\texttt{V}(t_2)$$

and similarly two lists are equivalent if they have the same length and are related elementwise

$$([v_1, \ldots, v_n], [v_1', \ldots, v_n']) \in \texttt{Rel}_\texttt{V}(t\ \texttt{list}) \quad \text{if } (v_i, v_i') \in \texttt{Rel}_\texttt{V}(t) \text{ for all } i \in 1..n$$

The part that makes logical relations interesting is how functions are compared. We could say that two function values are equivalent if they have the same definition, but this is a very limited notion of equivalence: the functions `fn (n:Int)=>n+1` and `fn (n:Int)=>1+n` would not be considered the same, even though they are interchangeable in any program. The definition we will use here, which is a defining characteristic of logical relations, is that two functions are equivalent if they send related arguments to related results.

$$(v, v') \in \mathtt{Rel_V}(t_1\mathtt{->}t_2) \quad \text{if } \forall (v_1, v_1') \in \mathtt{Rel_V}(t_1).\ (v\, v_1, v'\, v_1') \in \mathtt{Rel}(t_2)$$

Finally, we must define how the relation works for polymorphic types. The definition written out formally is a bit confusing, but the intuition is fairly simple:

> $(v_1, v_2) \in \mathtt{Rel_V}(\forall \alpha.u)$ if for all types t_1 and t_2, and *every relation* $\mathcal{R}$ *on values*, if you use $\mathcal{R}$ as the meaning of $\mathtt{Rel_V}(\alpha)$ then $(v_1[t_1], v_2[t_2]) \in \mathtt{Rel}(u)$.

After all this setup, we can then state the Parametricity Theorem:

> If e is a closed expression of closed type t, then $(e, e) \in \mathtt{Rel}(t)$.

That is, every well-typed program is related to itself. Because of the definition of the logical relation for function types and polymorphic types, this is not immediately obvious and requires proof. In particular, it doesn't just follow by induction on types.

What good is this theorem? Well, suppose we have a function `id` of type $\forall\alpha.\alpha\mathtt{->}\alpha$. The parametricity theorem says that

$$(\mathtt{id}, \mathtt{id}) \in \mathtt{Rel_V}(\forall\alpha.\alpha\mathtt{->}\alpha)$$

Expanding out the definition of the relation, this means that for all types t_1 and t_2 and every relation $\mathcal{R}$,

$$(\mathtt{id}[t_1], \mathtt{id}[t_2]) \in \mathtt{Rel}(\alpha\mathtt{->}\alpha)$$

when we use $\mathcal{R}$ as the meaning of $\mathtt{Rel_V}(\alpha)$. Then for all types t_1 and t_2 and every $\mathcal{R}$,

$$(\lfloor \mathtt{id}[t_1] \rfloor, \lfloor \mathtt{id}[t_2] \rfloor) \in \mathtt{Rel_V}(\alpha\mathtt{->}\alpha)$$

when $\mathcal{R}$ is used as the definition of $\mathtt{Rel_V}(\alpha)$. Expanding this out further, this says that for all types t_1 and t_2 and every relation $\mathcal{R}$, and all values v_1 and v_2 such that $(v_1, v_2) \in \mathcal{R}$, we have

$$(\lfloor \lfloor \mathtt{id}[t_1] \rfloor\, v_1 \rfloor, \lfloor \lfloor \mathtt{id}[t_2] \rfloor\, v_2 \rfloor) \in \mathcal{R}$$

Since $\lfloor \lfloor \mathtt{id}[t_1] \rfloor\, v_1 \rfloor = \lfloor \mathtt{id}[t_1]\, v_1 \rfloor$, we finally get: for all types t_1 and t_2 and every relation $\mathcal{R}$, and all values v_1 and v_2 such that $(v_1, v_2) \in \mathcal{R}$, we have

$$(\lfloor \mathtt{id}[t_1]\, v_1 \rfloor, \lfloor \mathtt{id}[t_2]\, v_2 \rfloor) \in \mathcal{R}$$

Suppose we have a type t and a value v of this type. Use this last property by setting $t_1 = t_2 = t$ and $v_1 = v_2 = v$. Let $\mathcal{R}$ be the relation that relates v to itself and relates nothing else to anything. By the parametricity theorem it therefore follows that

$$(\lfloor \mathtt{id}[t]\, v \rfloor, \lfloor \mathtt{id}[t]\, v \rfloor) \in \mathcal{R}$$

Since the only thing $\mathcal{R}$ does is relate v to itself, we immediately have that $\lfloor \mathtt{id}[t]\, v \rfloor = v$. Therefore, by knowing only the *type* of `id` we know that it must act like an identity function. (The name was a hint as well, but hardly conclusive evidence.)

1. Assume $f : \forall\alpha.\alpha\texttt{->}(\alpha\texttt{->}\alpha)$. Use parametricity to show that for any type t: either $\forall v_1, v_2 : t.\ \lfloor f[t](v_1)(v_2)\rfloor = v_1$ or else $\forall v_1, v_2 : t.\ \lfloor f[t](v_1)(v_2)\rfloor = v_2$. Hint: Construct a relation between booleans and values of type t.

2. **[10% EXTRA CREDIT]** Recall that the function `map` applies a given function to every element of a list and returns the list of results. Assume $f : \forall\alpha.(\alpha\ \texttt{list})\texttt{->}(\alpha\ \texttt{list})$. Show that for any types t and u, function $g : t \to u$, and list $v : t\ \texttt{list}$, it must be the case that $\lfloor \texttt{map}\ g\ (f[t](v))\rfloor = \lfloor f[u](\texttt{map}\ g\ v)\rfloor$ evaluate to the same value. Hint: a function can be viewed as a relation between values.

3. **[10% EXTRA CREDIT]** Assume $\texttt{sw} : \forall\alpha_1.\forall\alpha_2.(\alpha_1 \times \alpha_2)\texttt{->}(\alpha_2 \times \alpha_1)$. Show that if $v_1 : t_1$ and $v_2 : t_2$ then $\lfloor \texttt{sw}[t_1][t_2]\,(v_1, v_2)\rfloor = (v_2, v_1)$