Advanced Programming Languages — Assignment 5
CS 6110 S16 — Due Friday, 29 April 2016

---

1. Denesting Loops

   Here is a basic loop-denesting transformation that an optimizing compiler might perform:

   $$\begin{array}{l}\textsf{while } b \textsf{ do } \{\\ \quad p;\\ \quad \textsf{while } c \textsf{ do } q\\ \}\end{array} \quad \Rightarrow \quad \begin{array}{l}\textsf{if } b \textsf{ then } \{\\ \quad p;\\ \quad \textsf{while } b \vee c \textsf{ do } \{\\ \quad\quad \textsf{if } c \textsf{ then } q \textsf{ else } p\\ \quad \}\\ \}\end{array}$$

   Assume that the tests b and c have no side effects. Prove by equational reasoning in KAT (Lecture 18) that this transformation is correct.

2. Type Soundness

   The type preservation property, often called "subject reduction", states that if $\Gamma \vdash e : \tau$ and $e \to e'$ then $\Gamma \vdash e' : \tau$. Does the "subject expansion" property hold for $\lambda^{\to}$? That is if $e \to e'$ and $\Gamma \vdash e' : \tau$ then $\Gamma \vdash e : \tau$? If yes, prove it by induction. If not, give a concrete counter-example.

3. Type Preservation for References

   Suppose that we extend $\lambda^{\to}$ with references by adding the following expressions, types, and evaluation contexts to the language,

   $$\begin{array}{rcl} e & ::= & \ldots \mid \textsf{ref}\, e \mid\ !e \mid e_1 := e_2 \mid \ell \\ \tau & ::= & \ldots \mid \tau\ \textsf{ref} \\ E & ::= & \ldots \mid \textsf{ref}\, E \mid\ !E \mid E := e \mid v := E \end{array}$$

   as well as new evaluation rules (extending the other evaluation rules to work on configurations $\langle e, \sigma\rangle$ in the obvious way),

   $$\frac{\langle e, \sigma\rangle \to \langle e', \sigma'\rangle}{\langle E[e], \sigma\rangle \to \langle E[e'], \sigma'\rangle} \qquad \frac{\ell \notin \textsf{dom}\,\sigma}{\langle \textsf{ref}\ v, \sigma\rangle \to \langle \ell, \sigma[v/\ell]\rangle}$$

   $$\frac{}{\langle !\ell, \sigma\rangle \to \langle \sigma(\ell), \sigma\rangle} \qquad \frac{}{\langle \ell := v, \sigma\rangle \to \langle v, \sigma[v/\ell]\rangle}$$

   and typing rules:

   $$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textsf{ref}\, e : \tau\ \textsf{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \textsf{ref}}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \textsf{ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \tau}$$

   (a) Unlike the simply-typed λ-calculus, this extension is not strongly normalizing. In particular, it is simple to encode arbitrary recursive functions using references. Give an expression that computes the following function:

   $$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \textit{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \textit{if } m > 0 \text{ and } n > 0 \end{cases}$$

   You may assume that conditionals and suitable comparison functions on naturals are provided. However, make sure that your expression is well-typed according to the typing rules above!

(b) **Karma Problem:** Show how to encode the $\mathsf{rec}\ f : \sigma \to \tau . \lambda x : \sigma .\ e$ construct from Lecture 27.

(c) Unfortunately this type system does not enjoy the usual type preservation property,

$$\Gamma \vdash e : \tau \wedge e \to e' \implies \Gamma \vdash e' : \tau$$

This is easy to see because there is no typing rule for locations, which are introduced during the evaluation of $\mathsf{ref}\ e$ expressions.

To repair this deficiency, we can introduce a location context Σ to keep track of the types of locations. We extend the typing relation from a three-place relation $\Gamma \vdash e : \tau$ to a four-place relation $\Sigma; \Gamma \vdash e : \tau$ where Σ is a finite map from locations to types. We also add a new typing rule for locations:

$$\frac{\Sigma(\ell) = \tau}{\Sigma; \Gamma \vdash \ell : \tau\ \mathsf{ref}}$$

To capture the relationship between stores σ and location contexts Σ we say that a store σ is well-typed with respect to Γ and Σ, written $\Sigma; \Gamma \vdash \sigma$, if and only if

$$\mathsf{dom}\,(\sigma) = \mathsf{dom}\,(\Sigma) \quad \text{and} \quad \forall \ell \in \mathsf{dom}\,(\sigma).\ \Sigma; \Gamma \vdash \sigma(\ell) : \Sigma(\ell).$$

State the type preservation property for this language. Be careful! Some seemingly obvious variants of preservation do not hold.

(d) Give a careful proof of your type preservation property. You only need to give the cases of the proof for expressions related to references. In addition, you may use any auxiliary lemmas you need (e.g., substitution) without proof. However, you must show the top-level structure of your proof including the induction principle you are using, and clearly state any such lemmas. We will consider the quality and clarity of your proof during grading.

4. Polymorphism

In the early part of this course, we saw how to encode many datatypes such as pairs, booleans, natural numbers, lists, *etc.* in the λ-calculus. In this problem, we will see that parametric polymorphism is powerful enough to encode many of these same same datatypes. We will consider a restricted language which has only a base type, unit, containing a single value null, functions, and first-class polymorphic types. The grammar for this language, System F, is given below:

$$\begin{array}{rcl} e & ::= & \mathsf{null} \mid x \mid \lambda x : \tau .\ e \mid e\ e \mid \Lambda\alpha .\ e \mid e\ \tau \\ \tau & ::= & \alpha \mid \mathsf{unit} \mid \forall \alpha . \tau \mid \tau \to \tau \end{array}$$

The typing rules for System F are given in the Lecture 29 notes.

Recall that in the untyped λ-calculus, we encoded pairs (e_1, e_2) as terms $(\lambda p .\ p\ e_1\ e_2)$. Thus, we could write the function pair that takes e_1 and e_2 and produces (e_1, e_2) as: $(\lambda x, y, f .\ f\ x\ y)$. We can't use the same encoding in the simply-typed λ-calculus (without polymorphism) because we have to write down a type for f, but we don't know what the return type should be. That is, even if e_1 has type τ_1 and e_2 has type τ_2, there is no way to give a type for the "?" below such that term will have the properties needed to encode pairs:

$$\lambda x : \tau_1, y : \tau_2, f : \tau_1 \to \tau_2 \to ? .\ f\ x\ y$$

Polymorphic types, however, give us a reasonable way to give a type for "?". We can encode the type $\tau_1 * \tau_2$ using the polymorphic λ-calculus as follows:

$$\tau_1 * \tau_2 \triangleq \forall \gamma . (\tau_1 \to \tau_2 \to \gamma) \to \gamma$$

The introduction form for products can be encoded via a function $\mathsf{pair}^{\tau_1, \tau_2} : \tau_1 \to \tau_2 \to \tau_1 * \tau_2$ given by:

$$\mathsf{pair}^{\tau_1, \tau_2} \triangleq \lambda x : \tau_1 .\ \lambda y : \tau_2 .\ \Lambda\gamma .\ \lambda f : \tau_1 \to \tau_2 \to \gamma .\ f\ x\ y$$

Likewise, we can encode the elimination forms, $\mathsf{pi}_1^{\tau_1,\tau_2} : \tau_1 * \tau_2 \to \tau_1$ and $\mathsf{pi}_2^{\tau_1,\tau_2} : \tau_1 * \tau_2 \to \tau_2$, as follows:

$$\mathsf{pi}_1^{\tau_1,\tau_2} \triangleq \lambda p : \tau_1 * \tau_2 . (p\ \tau_1\ (\lambda x : \tau_1 . \lambda y : \tau_2 . x))$$
$$\mathsf{pi}_2^{\tau_1,\tau_2} \triangleq \lambda p : \tau_1 * \tau_2 . (p\ \tau_2\ (\lambda x : \tau_1 . \lambda y : \tau_2 . y))$$

In fact, we can do even better. Since the pair function and the projections pi_1 and pi_2 are parameterized by types τ_1 and τ_2, we can create polymorphic versions that take τ_1 and τ_2 as type arguments. That is, we can give pair the type $\forall\alpha.\forall\beta.\alpha \to \beta \to \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma$.

$$\mathsf{pair} \triangleq \Lambda\alpha . \Lambda\beta . \lambda x : \alpha . \lambda y : \beta . \Lambda\gamma . \lambda f : \alpha \to \beta \to \gamma . f\ x\ y$$
$$\mathsf{pi}_1 \triangleq \Lambda\alpha . \Lambda\beta . \lambda p : \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma . (p\ \alpha\ (\lambda x : \alpha . \lambda y : \beta . x))$$
$$\mathsf{pi}_2 \triangleq \Lambda\alpha . \Lambda\beta . \lambda p : \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma . (p\ \beta\ (\lambda x : \alpha . \lambda y : \beta . y))$$

(a) Define the type bool using the types available in the polymorphic λ-calculus. Give closed terms true and false such that $\vdash \mathsf{true} : \mathsf{bool}$ and $\vdash \mathsf{false} : \mathsf{bool}$. Also give a closed term if of type $\forall\alpha.\mathsf{bool} \to \alpha \to \alpha \to \alpha$ such that $(\mathsf{if}\ \sigma\ \mathsf{true}\ x\ y)$ reduces to x and $(\mathsf{if}\ \sigma\ \mathsf{false}\ x\ y)$ reduces to y. (Hint: look at the encoding of booleans in the untyped λ-calculus.)

(b) Just as we can encode product types using just polymorphism and functions, we can do the same thing for sums. Show how to encode the sum type $\tau_1 + \tau_2$ using this language. Give translations for polymorphic versions of the operators inl, inr, and case.

(c) We know that the untyped λ-calculus has divergent terms such as $((\lambda x . x\ x)\ (\lambda x . x\ x))$. It is a theorem (beyond the scope of this course) that programs written in the polymorphic λ-calculus studied here are strongly normalizing, and so we have lost some expressiveness by moving to a typed setting. Nevertheless, it is natural to wonder how "close" to the untyped λ-calculus we can get. In this problem, we will see that any primitive recursive function from the natural numbers to the natural numbers can be computed in the polymorphic λ-calculus.

The type nat of natural numbers represented as Church numerals can be easily defined. With explicit parameterization over types, the expressions for $\mathsf{0}$ and succ are as follows:

$$\mathsf{0} \triangleq \Lambda\alpha . \lambda f : \alpha \to \alpha . \lambda x : \alpha . x$$
$$\mathsf{succ} \triangleq \lambda n : \mathsf{nat} . \Lambda\alpha . \lambda f : \alpha \to \alpha . \lambda x : \alpha . f\ (n\ \alpha\ f\ x)$$

We wish to define a recursion operator, R, of the type $\mathsf{nat} \to (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \to \mathsf{nat} \to \mathsf{nat}$ with the property that $(\mathsf{R}\ n\ f\ \mathsf{0})$ reduces to n, and $(\mathsf{R}\ n\ f\ (\mathsf{succ}\ m))$ reduces to $(f\ (R\ n\ f\ m)\ (\mathsf{succ}\ m))$.

As an example of how to use the R operator, assume that the terms $\mathsf{1} : \mathsf{nat}$ and $\mathsf{times} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ are defined appropriately. Then the factorial function can be encoded as: $\lambda x : \mathsf{nat} . (\mathsf{R}\ \mathsf{1}\ \mathsf{times}\ x)$.

Give a closed, polymorphic λ-calculus term for R as described above. (Hint: You may find it helpful to create an auxiliary function $\mathsf{step} : (\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}) \to (\mathsf{nat} * \mathsf{nat}) \to (\mathsf{nat} * \mathsf{nat})$ that takes a function f and maps the pair $(\mathsf{R}\ n\ f\ m, m)$ to the pair $(\mathsf{R}\ n\ f\ (\mathsf{succ}\ m), \mathsf{succ}\ m)$. Feel free to use the terms pair and pi_i.)

5. Debriefing

(a) How many hours did you spend on this assignment?

(b) Would you rate it as easy, moderate, or difficult?

(c) Did everyone in your study group participate?

(d) How deeply do you feel you understand the material it covers (0%–100%)?

(e) If you have any other comments, we would like to hear them! Please write them here or send email to `jnfoster@cs.cornell.edu`