

Homework 1: Type Safety, Substitution and the λ -calculus

15-814: Types and Programming Languages
TA: Bernardo Toninho (btoninho@cs.cmu.edu)

Out: 9/13/11
Due: 9/27/11

Welcome to 15-814's first homework assignment! The goal of this assignment is to develop and practice the basic techniques necessary for the study of programming languages.

Please submit your work as a PDF file to btoninho@cs.cmu.edu and include the phrase "15-814 Homework" in the subject line of your email.

1 A Simple Expression Language

In this question, we will consider the following language of expressions e and types τ :

$$\begin{aligned}\tau &::= \text{num} \mid \text{bool} \\ e &::= \text{num}[n] \mid \text{true} \mid \text{false} \mid e_1 + e_2 \mid e_1 == e_2\end{aligned}$$

The types `num` and `bool` are the type of natural numbers and booleans, respectively. The language has the following constructs: `num[n]` is a representation for the natural number n ; `true` and `false` represent the boolean values; $e_1 + e_2$ denotes addition of natural numbers; $e_1 == e_2$ denotes the boolean equality test of natural numbers.

The type system for this language is given below:

Definition 1 (Type System)

$$\begin{array}{c} \frac{}{\text{num}[n] : \text{num}} \text{ (Tnum)} \quad \frac{}{\text{true} : \text{bool}} \text{ (Ttrue)} \quad \frac{}{\text{false} : \text{bool}} \text{ (Tfalse)} \\ \frac{e_1 : \text{num} \quad e_2 : \text{num}}{e_1 + e_2 : \text{num}} \text{ (T+)} \quad \frac{e_1 : \text{num} \quad e_2 : \text{num}}{e_1 == e_2 : \text{bool}} \text{ (T==)} \end{array}$$

The dynamic semantics for the language are defined by two judgments: $e \text{ val}$ denotes that the expression e is a value; $e \mapsto e'$ states that expression e transitions to expression e' . The value judgment is inductively defined by the following rule:

$$\frac{}{\text{num}[n] \text{ val}} \quad \frac{}{\text{true} \text{ val}} \quad \frac{}{\text{false} \text{ val}}$$

The transition judgment $e \mapsto e'$ is defined inductively below.

Definition 2 (Small-step dynamic semantics)

$$\begin{array}{c} \frac{e_1 \mapsto e'_1}{e_1 + e_2 \mapsto e'_1 + e_2} \text{ (+step}_1\text{)} \quad \frac{e_1 \text{ val} \quad e_2 \mapsto e'_2}{e_1 + e_2 \mapsto e_1 + e'_2} \text{ (+step}_2\text{)} \quad \frac{n_1 + n_2 = m}{\text{num}[n_1] + \text{num}[n_2] \mapsto \text{num}[m]} \text{ (+step}_3\text{)} \\ \frac{e_1 \mapsto e'_1}{e_1 == e_2 \mapsto e'_1 == e_2} \text{ (== step}_1\text{)} \quad \frac{e_1 \text{ val} \quad e_2 \mapsto e'_2}{e_1 == e_2 \mapsto e_1 == e'_2} \text{ (== step}_2\text{)} \quad \frac{n_1 = n_2}{\text{num}[n_1] == \text{num}[n_2] \mapsto \text{true}} \text{ (== step}_3\text{)} \\ \frac{n_1 \neq n_2}{\text{num}[n_1] == \text{num}[n_2] \mapsto \text{false}} \text{ (== step}_4\text{)} \end{array}$$

Task 1 (Type Preservation) Prove type preservation for the language presented above, that is, show that if $e : \tau$ and $e \mapsto e'$ then $e' : \tau$.

(Hint) Proceed by induction on the dynamic semantics.

Task 2 (Progress) Show that if $e : \tau$ then either $e \text{ val}$ or $e \mapsto e'$.

(Hint) Use induction on typing.

2 Variables and Binding

Consider that we wish to extend the language of Problem 1 with a form of declaration. One way of doing so is to add to the language a let-binding construct:

$$e ::= \text{let}(e_1, x.e_2) \mid x \mid \dots$$

The let binder allows us to bind the expression e_1 to the variable x in e_2 . To be able to account for the new constructs in our type system, we must generalize the typing judgment to a hypothetical judgment, written $\Gamma \vdash e : \tau$. A context Γ is a set of hypotheses of the form $x : \tau$, registering the fact that variable x stands for an expression of type τ . The meaning of $\Gamma \vdash e : \tau$ is that expression e has type τ , under the assumptions recorded in Γ . The new rules for our type system are (all remaining rules stay the same, with the straightforward addition of the context Γ to the premises and conclusions of every rule):

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \text{ (var)} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \text{let}(e_1, x.e_2) : \tau_2} \text{ (Tlet)}$$

The **var** rule captures the nature of a hypothetical judgment: we are always allowed to use an assumption. The rule for the let allows us to type such an expression with a type τ_2 , provided e_1 has some type τ_1 and provided that e_2 , under the assumption that x stands for an expression of type τ_1 , has type τ_2 . In the rules presented above, we assume that a context never has more than one variable with the same name.

Informally, we expect that the dynamics of a let declaration $\text{let}(e_1, x.e_2)$ is to plug in e_2 all occurrences of x with e_1 . To make this notion precise, we must therefore define a substitution relation $[e_1/x]e_2 = e_3$, denoting that e_3 is the result of substituting all occurrences of x in e_2 by e_1 . To give a precise account of substitution in the presence of binding, we require some additional machinery. In particular, we need a way of determining if a variable occurs free in a given expression. This is necessary because we do not want substitution to substitute bound occurrences of a variable (e.g. $[e/x]\text{let}(e_1, x.x) \neq \text{let}(e_1, x.e)$) since that would break the binding structure of the expression. We begin by first defining a renaming operation on expressions $[x \leftrightarrow y]e$, that replaces all occurrences of y in e with x and all occurrences of x in e with y , including binding occurrences:

$$\frac{}{[x \leftrightarrow y]y = x} \quad \frac{}{[x \leftrightarrow y]x = y} \quad \frac{z \neq y \quad z \neq x}{[x \leftrightarrow y]z = z}$$

$$\frac{}{[x \leftrightarrow y]\text{num}[n] = \text{num}[n]} \quad \frac{}{[x \leftrightarrow y]\text{true} = \text{true}} \quad \frac{}{[x \leftrightarrow y]\text{false} = \text{false}}$$

$$\frac{[x \leftrightarrow y]e_1 = e'_1 \quad [x \leftrightarrow y]e_2 = e'_2}{[x \leftrightarrow y]e_1 + e_2 = e'_1 + e'_2} \quad \frac{[x \leftrightarrow y]e_1 = e'_1 \quad [x \leftrightarrow y]e_2 = e'_2}{[x \leftrightarrow y](e_1 == e_2) = (e'_1 == e'_2)}$$

$$\frac{[x \leftrightarrow y]e_1 = e'_1 \quad [x \leftrightarrow y]z = z' \quad [x \leftrightarrow y]e_2 = e'_2}{[x \leftrightarrow y]\text{let}(e_1, z.e_2) = \text{let}(e'_1, z'.e'_2)}$$

Renaming is important because it allows to adequately define what it means for a variable to be free in an expression with binding, written $x \in e$:

Definition 3 (Free Variables)

$$\begin{aligned}
x \in y & \triangleq x = y \\
x \in e_1 + e_2 & \triangleq x \in e_1 \text{ or } x \in e_2 \\
x \in e_1 == e_2 & \triangleq x \in e_1 \text{ or } x \in e_2 \\
x \in \text{let}(e_1, y.e_2) & \triangleq x \in e_1 \text{ or, for all fresh } z, x \in [z \leftrightarrow y]e_2
\end{aligned}$$

We state that a variable is fresh if it is unused in any expression (a more precise account of freshness is given in Prof. Harper's book, Chapter 1.3). Thus, a variable x is free in a let-binding if it is free in the declared sub-expression or if for all possible fresh renamings of the variable bound in the let, x remains free in e_2 .

We can now finally define the substitution relation $[e_1/x]e_2 = e_3$, inductively on the structure of the expression e_2 as follows:

$$\begin{aligned}
& \frac{}{[e/x]x = e} \text{ (varsubst}_1\text{)} \quad \frac{x \neq y}{[e/x]y = y} \text{ (varsubst}_2\text{)} \quad \frac{}{[e/x]\text{num}[n] = \text{num}[n]} \text{ (numsubst)} \\
& \frac{}{[e/x]\text{true} = \text{true}} \text{ (truesubst)} \quad \frac{}{[e/x]\text{false} = \text{false}} \text{ (falsesubst)} \\
& \frac{[e/x]e_1 = e'_1 \quad [e/x]e_2 = e'_2}{[e/x]e_1 + e_2 = e'_1 + e'_2} \text{ (+subst)} \quad \frac{[e/x]e_1 = e'_1 \quad [e/x]e_2 = e'_2}{[e/x](e_1 == e_2) = (e'_1 == e'_2)} \text{ (== subst)} \\
& \frac{y \neq x \quad y \notin e \quad [e/x]e_1 = e'_1 \quad [e/x]e_2 = e'_2}{[e/x]\text{let}(e_1, y.e_2) = \text{let}(e'_1, y.e'_2)} \text{ (letsubst)}
\end{aligned}$$

Note that the side-conditions on the free variables in the rule for the let binder make substitution a partial function from expressions to expressions. In particular, substitution is not defined on expressions e_2 containing bound variables that clash either with x or e .

Task 3 Consider an alternative version of the letsubst rule where the side-condition $y \neq x$ is not present. What would be the consequence of such a change?

Task 4 Consider yet again an alternative version of letsubst where the side-condition $y \notin e$ is erased. Why would this be a problem?

To make substitution a total function from expressions to expressions, we need to define α -equivalence $e \equiv_\alpha e'$: an equivalence relation that equates identical expressions, modulo consistent renaming of their bound variables.

Definition 4 (α -equivalence)

$$\begin{aligned}
x & \equiv_\alpha x \\
\text{num}[n] & \equiv_\alpha \text{num}[n] \\
\text{true} & \equiv_\alpha \text{true} \\
\text{false} & \equiv_\alpha \text{false} \\
e_1 + e_2 & \equiv_\alpha e'_1 + e'_2 \iff e_1 \equiv_\alpha e'_1 \text{ and } e_2 \equiv_\alpha e'_2 \\
e_1 == e_2 & \equiv_\alpha e'_1 == e'_2 \iff e_1 \equiv_\alpha e'_1 \text{ and } e_2 \equiv_\alpha e'_2 \\
\text{let}(e_1, x.e_2) & \equiv_\alpha \text{let}(e'_1, y.e'_2) \iff e_1 \equiv_\alpha e'_1 \text{ and for all fresh } z, [z \leftrightarrow x]e_2 \equiv_\alpha [z \leftrightarrow y]e'_2
\end{aligned}$$

It can be shown that substitution as defined above is a total function on α -equivalence classes of expressions. We will from this point always consider such equivalence classes.

When we presented the typing rules using a hypothetical judgment, we mentioned that a context never has more than one variable with the same name. The rule for let could therefore not be applicable if a repeated variable were to occur in the expression. This is now a non-issue since the condition can always be verified by choosing an appropriate representative of the α -equivalence class of the expression.

We can finally define the let rule for the dynamics of our language:

$$\overline{\text{let}(e_1, x.e_2) \mapsto [e_1/x]e_2}$$

To show the new cases in the proof of type preservation for our language, we require an auxiliary theorem about substitution and typing. This theorem states that substitution is well-behaved with respect to typing.

Task 5 Show that if $\Gamma \vdash e : \tau_1$ and $\Gamma, x : \tau_1 \vdash e' : \tau$, then $\Gamma \vdash [e/x]e' : \tau$.

(Hint) Proceed by induction on the typing derivation of the second assumption. You are allowed to assume weakening (if $\Gamma \vdash e : \tau$ then $\Gamma, x : \tau_1 \vdash e : \tau$, for any type τ_1).

Task 6 Show the new cases in the proofs for type preservation and progress.

3 The λ and combinator calculi

Consider the following presentation of the λ -calculus:

$$M, N ::= x \mid M N \mid \lambda x.M$$

We define β -reduction $M \rightarrow_\beta N$ through the following rules (note that β -reduction is not defined in the same way as β -equivalence from class):

$$\frac{M \rightarrow_\beta M'}{M N \rightarrow_\beta M' N} \quad \frac{N \rightarrow_\beta N'}{M N \rightarrow_\beta M N'} \quad \frac{}{(\lambda x.M) N \rightarrow_\beta [N/x]M}$$

The first two rules allow for reductions in an application. The last rule substitutes N for x in the body M of a λ -abstraction (substitution is defined to be capture-avoiding, following the developments of the previous exercise).

Surprisingly, it turns out that we can define a “variant” of the λ -calculus where we replace the λ abstraction with a set of combinators in which there is no notion of a bound variable. A reason why we would want to do this is that computation with combinators, since it does not involve capture-avoiding substitution, is much simpler and therefore easier to implement in practice.

We begin by defining the terms of our combinator calculus, which consist of variables, three combinators called **I**, **K** and **S** (for historical reasons), and combinator application:

$$E ::= x \mid \mathbf{I} \mid \mathbf{K} \mid \mathbf{S} \mid E_1 E_2$$

We can then define combinator reduction, written $E_1 \rightarrow_{\mathbf{SK}} E_2$ as follows (X, Y, Z stand for combinator terms):

$$\overline{\mathbf{I} X \rightarrow_{\mathbf{SK}} X} \quad \overline{\mathbf{K} X Y \rightarrow_{\mathbf{SK}} X} \quad \overline{\mathbf{S} X Y Z \rightarrow_{\mathbf{SK}} (X Z)(Y Z)}$$

$$\frac{E_1 \rightarrow_{\mathbf{SK}} E'_1}{E_1 E_2 \rightarrow_{\mathbf{SK}} E'_1 E_2} \quad \frac{E_2 \rightarrow_{\mathbf{SK}} E'_2}{E_1 E_2 \rightarrow_{\mathbf{SK}} E_1 E'_2}$$

Our goal is to define a way to translate (or compile) a λ -term to a term in our combinator calculus, which by construction does not have a notion of binding. Therefore, the challenge is to conceive a way of simulating binding in the combinator calculus. Traditionally, this is done using something called *bracket abstraction*, written $[x]E$. E stands for a combinator term, and

x is a variable that is potentially free in E . Bracket abstraction is defined inductively over the combinator term E as follows:

$$\begin{aligned} [x]x &= \mathbf{I} \\ [x]y &= \mathbf{K}y && \text{for } y \neq x \\ [x]E &= \mathbf{K}E && \text{for } E = \mathbf{I}, \mathbf{K} \text{ or } \mathbf{S} \\ [x]E_1 E_2 &= \mathbf{S}([x]E_1)([x]E_2) \end{aligned}$$

We can now define our compilation function $(\cdot)^*$ that takes a λ -term and compiles it to a combinator term (making use of bracket abstraction):

$$\begin{aligned} x^* &= x \\ (MN)^* &= M^* N^* \\ (\lambda x.M)^* &= [x]M^* \end{aligned}$$

We must now check that our compilation is correct, in the sense that the combinator term E that results from the translation of a λ -term M can simulate the reduction steps that can be performed by M .

We begin by first proving the correctness of bracket abstraction, that is, the application of a bracket abstraction reduces to a substitution.

Task 7 Show that, for any E and E' , $([x]E) E' \rightarrow_{\mathbf{SK}}^* [E'/x]E$, where $\rightarrow_{\mathbf{SK}}^*$ is the reflexive transitive closure of $\rightarrow_{\mathbf{SK}}$.

(Hint) Proceed by induction on the structure of E .

We can now prove our main result. You may assume that $([N/x]M)^* = [N^*/x]M^*$.

Task 8 Show that if $M \rightarrow_{\beta} N$, then $M^* \rightarrow_{\mathbf{SK}}^* N^*$.

Bonus Task 1 It turns out that the \mathbf{I} combinator is actually redundant, in that its reduction behavior can be mimicked by application, \mathbf{S} and \mathbf{K} . Produce a combinator term \mathbf{T} that only makes use of \mathbf{S} , \mathbf{K} and application that can simulate the behavior of \mathbf{I} , in the following way:

$$\mathbf{T} X \rightarrow_{\mathbf{SK}}^* X$$