

Type Theory (15-814) Fall 2006

Homework 1: Inductive definitions and α -equivalence

William Lovas (wlovas@cs)

Out: Friday, September 22, 2005

Due: Friday, October 6, 2005 (before 1:30 pm)

Welcome to Type Systems! Your homeworks will generally consist of a theory component and a programming component: in the theory component, you will explore formalisms and proof techniques described in class, and in the programming component you will find practical applications for these ideas in implementation.

Your solution should be submitted electronically as three files: `solution.pdf`, `syntax.sml`, and `normalize.sml`. Submit these files by copying them to the directory

`/afs/cs.cmu.edu/academic/class/15814/handin/<userid>/hw1`

Good luck!

1 Theory: Inductive definitions

Your first homework covers the basics of inductive definitions and explores the wide range of applicability of inductive reasoning.

1.1 α -equivalence

Consider the α -equivalence relation over ABTs, defined by the following rules:

$$\begin{array}{c}
 \frac{}{x =_{\alpha} x \text{ abt}^0} \text{ (\alpha-VAR)} \qquad \frac{a_1 =_{\alpha} b_1 \text{ abt}^{n_1} \quad \dots \quad a_k =_{\alpha} b_k \text{ abt}^{n_k} \quad (o \text{ has valence } (n_1, \dots, n_k))}{o(a_1, \dots, a_k) =_{\alpha} o(b_1, \dots, b_k) \text{ abt}^0} \text{ (\alpha-OP)} \\
 \\
 \frac{a =_{\alpha} b \text{ abt}^n}{x.a =_{\alpha} x.b \text{ abt}^{n+1}} \text{ (\alpha-BIND-SAME)} \qquad \frac{x \# y \quad y \# a \text{ abt}^n \quad [x \leftrightarrow y]a =_{\alpha} b \text{ abt}^n}{x.a =_{\alpha} y.b \text{ abt}^{n+1}} \text{ (\alpha-BIND-DIFF)} \\
 \\
 \frac{}{a =_{\alpha} a \text{ abt}^n} \text{ (\alpha-REFL)} \qquad \frac{b =_{\alpha} a \text{ abt}^n}{a =_{\alpha} b \text{ abt}^n} \text{ (\alpha-SYMM)} \qquad \frac{a =_{\alpha} b \text{ abt}^n \quad b =_{\alpha} c \text{ abt}^n}{a =_{\alpha} c \text{ abt}^n} \text{ (\alpha-TRANS)}
 \end{array}$$

Exercise 1. Show by providing derivations that the following rules are derivable:

$$\frac{x \# y \quad y \# a \text{ abt}^n}{x.a =_{\alpha} y.[x \leftrightarrow y]a \text{ abt}^{n+1}} \text{ (\alpha-CONV)}$$

$$\frac{y \# z \quad x \# z \quad z \# b \text{ abt}^n \quad z \# a \text{ abt}^n \quad [z \leftrightarrow y]b =_{\alpha} [z \leftrightarrow x]a \text{ abt}^n}{x.a =_{\alpha} y.b \text{ abt}^{n+1}} \text{ (\alpha-BIND)}$$

Hint: You may find it useful to use the α -CONV rule in your derivation of the α -BIND rule.

In fact, the rules explicitly making $=_\alpha$ an equivalence relation, labeled α -REFL, α -SYMM, α -TRANS above, are admissible: $=_\alpha$ is an equivalence relation even without them.

Exercise 2. Prove that rule α -SYMM is admissible under rules α -VAR, α -OP, α -BIND-SAME, and α -BIND-DIFF.

Hint: You may find it useful to prove the following lemmas first:

Lemma 1 (swapping preserves apartness). If $z \# a \text{ abt}^n$, then $[x \leftrightarrow y]z \# [x \leftrightarrow y]a \text{ abt}^n$.

Lemma 2 (swapping preserves α -equivalence). . If $a =_\alpha b \text{ abt}^n$, then $[x \leftrightarrow y]a =_\alpha [x \leftrightarrow y]b \text{ abt}^n$.

Lemma 3 (swapping is self-inverse). If $[x \leftrightarrow y]a = a'$, then $[x \leftrightarrow y]a' = a$.

You may assume the following lemmas *without proof*:

Lemma 4 (apartness respects α -equivalence). If $x \# a \text{ abt}^n$ and $a =_\alpha b \text{ abt}^n$, then $x \# b \text{ abt}^n$.

Lemma 5 (permutation of swapping). If $[x \leftrightarrow y]z = z'$ and $[x \leftrightarrow y]w = w'$, then $[x \leftrightarrow y][w \leftrightarrow z]a = [w' \leftrightarrow z'] [x \leftrightarrow y]a$.

1.2 Parsing

The machinery of inductive definitions can be used to give rules to define traditional BNF grammars. Recall the following grammar for arithmetic expressions and definition of abstract syntax trees, given in class as inference rules:

$$\begin{array}{ccc}
 \frac{n \text{ nat}}{n F} & \frac{s = (\wedge s_1 \wedge) \quad s_1 E}{s F} & \frac{n \text{ nat}}{\text{num}[n] \text{ ast}} \\
 \\
 \frac{s F}{s T} & \frac{s = s_1 \wedge * \wedge s_2 \quad s_1 F \quad s_2 T}{s T} & \frac{a_1 \text{ ast} \quad a_2 \text{ ast}}{\text{times}(a_1, a_2) \text{ ast}} \\
 \\
 \frac{s T}{s E} & \frac{s = s_1 \wedge + \wedge s_2 \quad s_1 T \quad s_2 E}{s E} & \frac{a_1 \text{ ast} \quad a_2 \text{ ast}}{\text{plus}(a_1, a_2) \text{ ast}}
 \end{array}$$

Recall also that we gave rules for parsing and pretty-printing between strings representing expressions and abstract syntax trees:

$$\begin{array}{ccc}
 \frac{n \text{ nat}}{n F \longleftrightarrow \text{num}[n] \text{ ast}} \text{ (PARSE-F-NAT)} & \frac{s = (\wedge s_1 \wedge) \quad s_1 E \longleftrightarrow a_1 \text{ ast}}{s F \longleftrightarrow a_1 \text{ ast}} \text{ (PARSE-F-PARENS)} & \\
 \\
 \frac{s F \longleftrightarrow a \text{ ast}}{s T \longleftrightarrow a \text{ ast}} \text{ (PARSE-T-F)} & \frac{s = s_1 \wedge * \wedge s_2 \quad s_1 F \longleftrightarrow a_1 \text{ ast} \quad s_2 T \longleftrightarrow a_2 \text{ ast}}{s T \longleftrightarrow \text{times}(a_1, a_2) \text{ ast}} \text{ (PARSE-T-TIMES)} & \\
 \\
 \frac{s T \longleftrightarrow a \text{ ast}}{s E \longleftrightarrow a \text{ ast}} \text{ (PARSE-E-T)} & \frac{s = s_1 \wedge + \wedge s_2 \quad s_1 T \longleftrightarrow a_1 \text{ ast} \quad s_2 E \longleftrightarrow a_2 \text{ ast}}{s E \longleftrightarrow \text{plus}(a_1, a_2) \text{ ast}} \text{ (PARSE-E-PLUS)} &
 \end{array}$$

Exercise 3. Prove by simultaneous rule induction the following *validity properties* of the parsing/pretty-printing judgements:

1. If $s F \longleftrightarrow a \text{ ast}$, then $s F$ and $a \text{ ast}$.
2. If $s T \longleftrightarrow a \text{ ast}$, then $s T$ and $a \text{ ast}$.
3. If $s E \longleftrightarrow a \text{ ast}$, then $s E$ and $a \text{ ast}$.

1.3 Substitution

Substitution was defined in class by the following rules:

$$\frac{}{[a/x]x = a} \text{ (SUBST-VAR-SAME)} \qquad \frac{x \# y}{[a/x]y = y} \text{ (SUBST-VAR-DIFF)}$$
$$\frac{[a/x]a_1b_1 \quad \dots \quad [a/x]a_kb_k}{[a/x]o(a_1, \dots, a_k) = o(b_1, \dots, b_k)} \text{ (SUBST-OP)} \qquad \frac{x \# y \quad y \# a \text{ abt}^n \quad [a/x]b = c}{[a/x]y. b = y. c} \text{ (SUBST-BIND)}$$

Exercise 4. Prove by structural induction over the ABT b that the substitution relation $[a/x]b = b'$ has mode $(\forall, \forall, \forall, \exists!)$ and thus defines a function. That is, prove that $\forall a \text{ abt}^0. \forall x \text{ name}. \forall b \text{ abt}^n. \exists! b' \text{ abt}^n. [a/x]b = b'$.

2 Programming: α -equivalence and β -normalization

For your first programming component, you'll implement a full β -normalizer for the λ -calculus. Along the way, you'll implement an incredibly useful abstraction for variable binding that you'll use throughout the semester.

2.1 Abstract types for abstract syntax

On paper and at the blackboard, we reason about terms with binding structure up to α -equivalence. We have an induction principle that generalizes ordinary structural induction by explicating that when we come to a binder, we can assume inductively that it could bind *any* variable, in particular one that does not interfere with the rest of our proof.

Similarly, we'd like to be able to write recursive programs over terms with binding structure up to α -equivalence: we'd like to not have to worry about the names of bound variables. We can achieve this by implementing our term structure as an abstract type endowed with projection functions to and from a concrete datatype representation. When we project an abstract representation to a concrete one, we α -vary its bound variable to a fresh one so that the rest of the program can proceed without having to worry about it. (See Figure 1.)

This interface is an instance of a common functional programming paradigm where one has both an abstract representation and a concrete representation along with functions to transform between them while maintaining some invariants — our invariant is simply that bound variables never interfere with the rest of the program. Such an interface is colloquially referred to as a “wizard”, as in “pay no attention to the code behind the curtain”.

Program 1. Fill in the `term_out` function in the `Syntax` structure. Your function should take care to maintain the invariant that any bound variable presented to the rest of the program has a fresh name. You will want to make use of the functions in the `Syntax.Name` structure to generate fresh names — see `name-sig.sml` for details.

Part of the beauty of programming with abstract types is that it allows one to change the underlying implementation of an abstraction without affecting any code that uses it. Pierce describes a nameless representation of terms due to Nicolaas de Bruijn, where a variable is denoted not by a name but by the number of abstractors between its occurrence and its binding site. For example, the term $\lambda x. x (\lambda y. x y)$ would be represented by the nameless term $\lambda. 0 (\lambda. 1 0)$.

Program 2 (optional). Experiment with an internal implementation of λ -terms that uses the nameless representation. Observe that as long as the abstraction is implemented correctly, no calling code outside the abstraction can tell the difference!

```

signature SYNTAX =
  sig
    structure Name :
      sig
        type name
        ...
      end

    type term

    datatype term_ =
      TmVar of Name.name
    | TmAbs of Name.name * term
    | TmApp of term * term

    val term_out : term -> term_
    val term_in  : term_ -> term
    ...
  end

```

Figure 1: Abridged SYNTAX signature with “wizard” interface. The internal type `term` is hidden from view, but the `term_out` function allows a client to obtain a concrete representation obeying certain invariants.

2.2 β -normalization

Using the interface you implemented for manipulating λ -terms modulo α -conversion, you will implement a full β -normalizer for the λ -calculus. Your normalizer should produce a λ -term that is β -equivalent to its input and contains no β -redexes.

One strategy for implementing such a normalizer might follow rules such as the following, writing $t \Downarrow u$ for “term t has β -normal form u ”:

$$\begin{array}{c}
 \frac{}{x \Downarrow x} \\
 \\
 \frac{t_1 \Downarrow \lambda x. u_1 \quad t_2 \Downarrow u_2 \quad [u_2/x]u_1 \Downarrow u}{t_1 t_2 \Downarrow u} \qquad \frac{t_1 \Downarrow u_1 \quad t_2 \Downarrow u_2 \quad (u_1 \neq \lambda x. u'_1)}{t_1 t_2 \Downarrow u_1 u_2} \\
 \\
 \frac{t_1 \Downarrow u_1}{\lambda x. t_1 \Downarrow \lambda x. u_1}
 \end{array}$$

Program 3. Implement the functions `subst` and `normalize` in the structure `Normalize`. `subst` should perform capture-avoiding substitution and `normalize` should take a term t to its normal form u (up to α -equivalence, of course), where $t \Downarrow u$.

Hint: If you’ve programmed your wizard interface correctly, these functions should be very simple!

2.3 Test cases

There are a few limited test cases in the files `example.txt` and `example.sml`. You are encouraged to devise your own test cases and submit them with your assignment. Not only will this help you to understand the material better, but it might also improve your grade. Although you won’t be graded on the submitted tests themselves, your solution will be tested against a subset of submitted test cases, including your own. Thus, it is in your interest to submit test cases your solution handles correctly.