Recitation 2:
Abstract Syntax and De Bruijn Indices

15-312: Principles of Programming Languages

Wednesday, January 22, 2014

1 Abstract Syntax

Chapter 1 of PFPL has a rigorous treatment of abstract syntax in its most general form to support the requirements of most programming languages. In this recitation we will look at simpler abstract syntax trees and abstract binding trees similar to what we have asked you to implement in Assignment 1. We will also look at an alternative representation for bound varibles called deBruijn indices

Abstract syntax/binding trees are tree representations of programs. As we have seen so far in the course, such a tree representation allows us to do structural induction over programs, which in turn lets us prove properties about our programming language.

1.1 AST

Given a set X of variables, a set O of operators and an arity function ar on operators, we can inductively define abstract syntax trees in the following way

- If $x \in X$ then x is an AST.
- If $o \in O$ is an operator such that $\mathsf{ar}(o) = n$, then if a_1, ..., a_n are AST's then $o(a_1, ..., a_n)$ is an AST.

The arity function returns a number for each operator which is the number of arguments it take. In particular, if $\mathsf{ar}(o) = 0$, then $o()$ is an AST. In $\mathcal{L}(\texttt{natstr})$, operators like plus have an arity of 2 and atomic operators like strings and numbers have an arity of 0.

Note: In PFPL, the arity of an operator for an AST is defined as a tuple of "sorts". At this point in the class, the languages we are looking at only have one sort, so the arity of an operator for an AST can be adequately described by a single integer specifying the size of the tuple. Later on in the class, languages with multiple sorts will be introduced and the concept itself will be better explained.

1.2 ABT

Abstract Binding Trees enrich AST's with the means to introduce new variable parameters called bindings. The arity of each operator needs to be generalized to consist of a finite sequence of numbers which we will call valences. Valences indicate how many variables are introduced at a particular point. For example, if we have an operator whose expressions look like, $\mathsf{op}(x.y.e_1; x.e_2)$ where two variables are bound to the first argument and one variable is bound to its second argument, then $\mathsf{ar}(\mathsf{op}) = (2, 1)$. We generalize the definition of AST's to ABT's as follows.

Given a set X of variables, a set O of operators and an arity function ar on operators, we can inductively define abstract binding trees in the following way

- If $x \in X$ then x is an ABT.
- If $o \in O$ is an operator such that $\mathsf{ar}(o) = (v_1, ..., v_n)$, then if for each $1 \leq i \leq n$, x_i is a sequence of variables of length v_i and a_i is an ABT, then, $o(\vec{x_1}.a_1, ..., \vec{x_n}.a_n)$ is an ABT.

1.3 Substitution

Now that we have a definition for ABT's we can define substitution of an ABT b for a variable x in some ABT a. This is denoted by $[b/x]a$. This is partially defined by the following conditions.

- $[b/x]x = b$
- $[b/x]y = y$ if $x \neq y$
- $[b/x]o(\vec{x_1}.a_1, ...\vec{x_n}.a_n) = o(\vec{x_1}.[b/x]a_1, ..., \vec{[x_n]}.[b/x]a_n)$ if $\vec{x_i} \notin b$ and $x \notin \vec{x_i}$ for all $1 \leq i \leq n$.

In this definition $\vec{x_i} \notin b$ means that no variable in $\vec{x_i}$ occurs free in b. The removal of these restrictions would create additional unwanted binding of free variables. This unwanted binding is called capture. These restrictions result in substitution being undefined for certain terms. However, if we are allowed to freely rename bound variables, substitution is well-defined on an α-equivalence class of terms.

Task Let us see how substitution works in $\mathcal{L}(\texttt{natstr})$. Write out the result of the substitution $[b/x]a$ for the following a and b. You are allowed to make α-conversions to avoid capture.

- $a = \mathsf{let}(y; x.x)$ and $b = z * 2$
- $a = \mathsf{let}(5; y.x)$ and $b = y + 5$
- $a = (x + 3)$ and $b = "abc"$

2 De Bruijn Indices

So far, we have discussed everything "up to alpha-renaming of bound variables". Which is saying that we rename bound variables at will. However, we need a single representation for a term while implementing ABTs. One of the inconveniences of using a straightforward representation of ABTs is that α-equivalent terms can have multiple representations, so implementing `aequiv` and other helper functions becomes tricky. We will look at a more sophisticated representation, called *locally nameless form*, which avoids this problem, so that each bound variable is represented by a single number, called a *de Bruijn index*. The representation for a term then becomes a single data value.

Some of the nice features which de Bruijn indices offer are:.

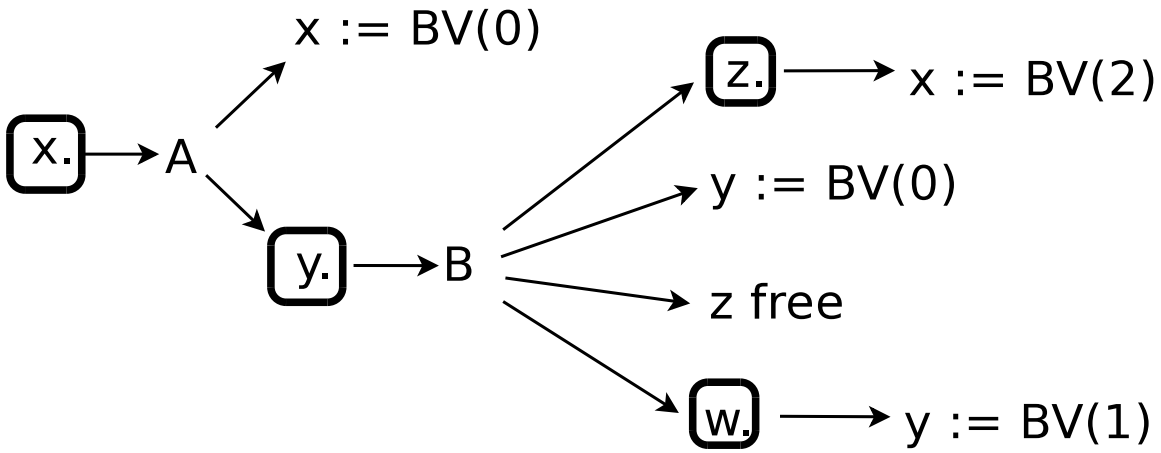
- α-equivalence can be tested by structure traversal.
- No renaming required for bound variables.
- As bound variables are separate from free variables, capture is impossible

- It is stable under substitution. i.e. de Bruijn indices don't change when you substitute one term in another

The basic idea is to observe that variables in an abstract binding tree serve two roles. First, they can appear free in a ABT – that is, they can be a *name* for a hypothesis in some hypothetical context. Second, they can be a bound variable, in which case the variable occurences *refer back* to the location of the binding abstraction. In the second case, the names

In a locally nameless representation, we distinguish these two roles in our data structure, in order to make α-equivalence implementable as structural equality on terms. The trick in this representation is to exploit a structural invariant of binding trees. First, binding trees are *trees* – they have no cycles in them. Second, in any abstraction $x.e$, the only occurences of x are within e. As a result of these two facts, we have a *unique* path from an abstractor to each occurence of the variable it binds. Furthermore, since we're only interested in the binder sites, we can compress this path to a single number, which tells us how many binders we have to hop over before we reach the one we're interested in.

Consider the following diagram of the fragment of a binding tree $x.A(x, y.B(z.x, y, z, w.y))$, in which A and B are operator names.



We've put a box around every abstraction, and labelled each bound variable with its bound variable number. We can calculate the bound variable number by looking at each path from an abstraction to its use sites, and count the number of abstractions crossed along the way:

Path	Variable #
$\boxed{x.} \to A \to x$	0
$\boxed{x.} \to A \to \boxed{y.} \to B \to \boxed{z.} \to x$	2
$\boxed{y.} \to B \to y$	0
$\boxed{y.} \to B \to \boxed{w.} \to y$	1

An important fact to notice about these paths is that even for the same binder, each occurrence of its bound variable can have a *different* bound variable number, depending on the number of abstractions we crossed over to reach that variable occurrence.

Task For the term $\mathsf{let}(z; x.z + \mathsf{let}(x; y.x + y))$, replace each bound variable with its de Bruijn index and denote each binding site with the character $-$