510: Programming Languages

Higher-Order Abstract Syntax

David Walker

Fall, 2002

Levels of Abstract Syntax

First-order abstract syntax (FOAS).

- Representation of syntax as terms (labelled trees).

- Expose "deep structure" of a phrase.

- Ignore "surface structure" of a phrase.

Levels of Abstract Syntax

Higher-Order abstract syntax (HOAS).

- Expose **binding structure** of a phrase.
- Renaming of bound variables.
- Substitution.

Binding and Scope in English

Pronouns and **demonstratives** are analogous to bound variables.

- He, she, it, this, that refer to a noun introduced elsewhere.

- Linguistic conventions establish scope and binding of pronouns and demonstratives.

Natural languages have only a small, fixed number of variables! Confusion is possible.

Binding and Scope in Arithmetic

Add to our language of arithmetic expressions the ability to

- **Bind** a variable to an expression in a given **scope**.
- **Refer** to (the value of) that expression.

There is an unlimited supply of variables. There will be no possibility for confusion.

The Let Expression

The expression `let` x `be` e `in` e' binds a **new** variable for us in a given context.

- The variable x is **bound** in its **body** e';
- The **scope** of x is the body e' of the `let` expression.

Bound Variables

What do we mean by a **new** variable?

- Every `let` binds a variable that is uniquely associated with that `let` expression.

- There can be **no confusion** about which variable is bound by which `let` expression.

Bound Variables

How is this achieved?

- Variables are referred to by name.
- It is possible to re-use variable names.
- It is unreasonable to preclude re-use of names.

Re-Use of Bound Names

Examples:

- "Parallel" scopes:

```
(let x be 3 in x+x) * (let x be 4 in x+x+x)
```

- "Nested" scopes:

```
let x be 10 in (let x be 11 in x+x)+x
```

Scope Resolution

Where is a variable bound?

Lexical scope rule: a variable is bound by the **nearest enclosing binding**.

- Proceed upwards through the abstract syntax tree.

- Find nearest enclosing `let` that binds that variable.

Renaming Bound Variables

We'll make use of **identification up to re-naming**, or α **conversion**.

- The **name** of a bound variable does not matter.

- Choose a different name to avoid ambiguity.

Names of Bound Variables

But watch out:

- `let x be 10 in (let x be 11 in x+x)+x` is the **same** as
  `let y be 10 in (let x be 11 in x+x)+y`.

- but is **different** from `let y be 10 in (let x be 11 in y+y)+y`.

When renaming we must avoid clashes with other variables in the same scope.

Higher-Order Abstract Syntax

Key idea: the hoas of a phrase is an **equivalence class**.

- **Equate** all foas terms that differ only in the names of bound variables.

- **Choose representatives** so as to satisfy any non-occurrence conditions.

Higher-Order Abstract Syntax

We choose representatives to ensure that bound variables are “fresh”.

Thus if `x` is “in use” we prefer

```
let y be 1+1 in y+y end
```

as a representative of the equivalence class that also contains

```
let x be 1+1 in x+x end
```

Higher-Order Abstract Syntax

For example, we prefer

```
let x be 10 in (let y be 11 in y+y)+x
```

to

```
let x be 10 in (let x be 11 in x+x)+x
```

since it avoids re-using `x`.

Free Variables

The **free** variables of an expression are those that are not bound within it.

$$
\begin{aligned}
\mathsf{FV}(\mathtt{var}(x)) &= \{\, x \,\} \\
\mathsf{FV}(\mathtt{number}(n)) &= \emptyset \\
\mathsf{FV}(\mathtt{plus}(e_1, e_2)) &= \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\
\mathsf{FV}(\mathtt{times}(e_1, e_2)) &= \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\
\mathsf{FV}(\mathtt{let}(x, e_1, e_2)) &= \mathsf{FV}(e_1) \cup (\mathsf{FV}(e_2) \setminus \{\, x \,\})
\end{aligned}
$$

Variable Renaming

Define $[y/x]e$ to be e with all free occ's of x renamed to y:

$$[y/x]\texttt{var}(z) = \begin{cases} \texttt{var}(y) & (z = x) \\ \texttt{var}(z) & (z \neq x) \end{cases}$$

$$[y/x]\texttt{number}(n) = \texttt{number}(n)$$

$$[y/x]\texttt{plus}(e_1, e_2) = \texttt{plus}([y/x]e_1, [y/x]e_2)$$

$$[y/x]\texttt{times}(e_1, e_2) = \texttt{times}([y/x]e_1, [y/x]e_2)$$

$$[y/x]\texttt{let}(z, e_1, e_2) = \begin{cases} \texttt{let}(z, [y/x]e_1, e_2) & (z = x) \\ \texttt{let}(z, [y/x]e_1, [y/x]e_2) & (z \neq x, z \neq y) \end{cases}$$

α Conversion

The relation $e_1 \equiv e_2$, called α-**conversion**, is the smallest congruence containing the axiom:

$$\overline{\texttt{let}(x, e_1, e_2) \equiv \texttt{let}(y, e_1, [y/x]e_2)}$$

where y is neither free nor bound in e_2.

α Conversion

Saying "smallest congruence" is an implied inductive definition!

- **Equivalence** relation:

$$\frac{}{e \equiv e} \qquad \frac{e_1 \equiv e_2}{e_2 \equiv e_1} \qquad \frac{e_1 \equiv e_2 \quad e_2 \equiv e_3}{e_1 \equiv e_3}$$

- **Compatibility** rules, such as:

$$\frac{e_1 \equiv e_1' \qquad e_2 \equiv e_2'}{\texttt{plus}(e_1, e_2) \equiv \texttt{plus}(e_1', e_2')}$$

Higher-Order Abstract Syntax

HOAS is the **quotient** of FOAS by α-conversion:

$$HOAS = \{\, [e] \mid e \in FOAS \,\}$$

where $[e]$ is the α-**equivalence class** of e:

$$[e] = \{\, e' \mid e \equiv e' \,\}.$$

Substitution and Capture

We extend variable renaming to substitution: $[e/x]e'$.

Suppose that e is `2*y+1` and that e' is `let y be 5 in x+y`.

What should be $[e/x]e'$?

- **Wrong**: `let y be 5 in (2*y+1)+y`. Inner binder **captures** `y` in e'.

- **Right**: `let z be 5 in (2*y+1)+z`. Rename inner binder to avoid capture.

Substitution

$$
\begin{aligned}
[e/x]\mathtt{var}(z) &= \begin{cases} e & (z = x) \\ \mathtt{var}(z) & (z \neq x) \end{cases} \\
[e/x]\mathtt{number}(n) &= n \\
[e/x](\mathtt{plus}(e_1', e_2')) &= \mathtt{plus}([e/x]e_1', [e/x]e_2') \\
[e/x](\mathtt{times}(e_1', e_2')) &= \mathtt{times}([e/x]e_1', [e'/x]e_2') \\
[e/x]\mathtt{let}(y, e_1', e_2') &= \mathtt{let}(y, [e/x]e_1', [e/x]e_2') \ (y \neq x,
\end{aligned}
$$

Substitution

The condition on variables in the last clause of substitution can always be achieved, up to α-conversion!

Fact: Substitution is well-defined: if $e \equiv e'$ and $e_1 \equiv e'_1$, then $[e/x]e_1 \equiv [e'/x]e'_1$.

deBruijn Indices

Another method for managing variables is to avoid names entirely!

deBruijn indices

- Variables do not have names.
- Reference is via an **index** that determines the binder.
- Index $i = i$th enclosing binder.

deBruijn Indices

Example:

```
let x be 10 in let y be 11 in x+y
```

is represented as

let 10 in let 11 in $\boxed{2}$+$\boxed{1}$

deBruijn Indices

The index of a variable varies according to context!

```
let x be 10 in (let y be 11 in y+x)+x
```

is represented by

let 10 in (let 11 in $\boxed{1}$+$\boxed{2}$)+$\boxed{1}$

The **same** variable can have **many** indices!

deBruijn Indices

deBruijn indices work well for **implementations**

- No need to worry about renaming or capture.

- Canonical representative of an equivalence class.

But they are terrible for **people**!

- We prefer to use names for variables.

Summary

Abstract Syntax

- Ignore issues of textual presentation.

- Concentrate on hierarchical and binding structure.

- Ensure that bound variables are "new".