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 \( \text{let } x \text{ be } e \text{ in } e' \) 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.

Re-Use of Bound Names

Examples:

- "Parallel" scopes:
  \((\text{let } x \text{ be } 3 \text{ in } x+x) \ast (\text{let } x \text{ be } 4 \text{ in } x+x+x)\)

- "Nested" scopes:
  \(\text{let } x \text{ be } 10 \text{ in } (\text{let } x \text{ be } 11 \text{ in } x+x)+x\)

Renaming Bound Variables

We'll make use of identification up to renaming, or a conversion.

- The name of a bound variable does not matter.
- Choose a different name to avoid ambiguity.

Names of Bound Variables

But watch out:

- \(\text{let } x \text{ be } 10 \text{ in } (\text{let } x \text{ be } 11 \text{ in } x+x)+x\) is the same as \(\text{let } y \text{ be } 10 \text{ in } (\text{let } x \text{ be } 11 \text{ in } x+x)+y\).
- but is different from \(\text{let } y \text{ be } 10 \text{ in } (\text{let } x \text{ be } 11 \text{ in } y+y)+y\).

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

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.

Names of Bound Variables

But watch out:

- \(\text{let } x \text{ be } 10 \text{ in } (\text{let } x \text{ be } 11 \text{ in } x+x)+x\) is the same as \(\text{let } y \text{ be } 10 \text{ in } (\text{let } x \text{ be } 11 \text{ in } x+x)+y\).
- but is different from \(\text{let } y \text{ be } 10 \text{ in } (\text{let } x \text{ be } 11 \text{ in } y+y)+y\).

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

For example, we prefer

\[\text{let } x \text{ be 10 in } (\text{let } y \text{ be 11 in } y+y)+x\]

to

\[\text{let } x \text{ be 10 in } (\text{let } x \text{ be 11 in } x+x)+x\]

since it avoids re-using \(x\).

Variable Renaming

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

\[\begin{align*}
[y/x]\text{var}(z) &= \begin{cases} \text{var}(y) & (z = x) \\ \text{var}(z) & (z \neq x) \end{cases} \\
[y/x]\text{num}[n] &= \text{num}[n] \\
[y/x]\text{plus}(c_1, c_2) &= \text{plus}([y/x]c_1, [y/x]c_2) \\
[y/x]\text{times}(c_1, c_2) &= \text{times}([y/x]c_1, [y/x]c_2) \\
[y/x]\text{let}(z, c_1, c_2) &= \begin{cases} \text{let}(z, [y/x]c_1, c_2) & (z = x) \\ \text{let}(z, [y/x]c_1, [y/x]c_2) & (z \neq x, z \neq y) \end{cases}
\end{align*}\]

Free Variables

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

\[\begin{align*}
\text{FV}(\text{var}(x)) &= \{x\} \\
\text{FV}(\text{num}[n]) &= \emptyset \\
\text{FV}(\text{plus}(c_1, c_2)) &= \text{FV}(c_1) \cup \text{FV}(c_2) \\
\text{FV}(\text{times}(c_1, c_2)) &= \text{FV}(c_1) \cup \text{FV}(c_2) \\
\text{FV}(\text{let}(x, c_1, c_2)) &= \text{FV}(c_1) \cup (\text{FV}(c_2) \setminus \{x\})
\end{align*}\]

\(\alpha\) Conversion

The relation \(c_1 \equiv c_2\), called \(\alpha\)-conversion, is the smallest congruence containing the axiom:

\[\text{let}(x, c_1, c_2) \equiv \text{let}(y, c_1, [y/x]c_2)\]

where \(y\) is neither free nor bound in \(c_2\).
\[ \alpha \text{ Conversion} \]

Saying “smallest congruence” is an implied inductive definition!

- **Equivalence** relation:

   \[ e \equiv e' \quad e_1 \equiv e_2 \quad e_2 \equiv e_3 \quad e_1 \equiv e_3 \]

- **Compatibility** rules, such as:

\[
\frac{e_1 \equiv e'_1 \quad e_2 \equiv e'_2}{\text{plus}(e_1, e_2) \equiv \text{plus}(e'_1, e'_2)}
\]

\[ \text{Substitution} \]

We extend variable renaming to substitution: \([e/x]e'\).

Suppose that \(e\) is \(2\cdot y + 1\) and that \(e'\) is \(\text{let } z\ be\ 5\ \text{in}\ \(2\cdot y + 1) + z\). Inner binder captures \(y\) in \(e'\).

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

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

\[ \text{Substitution} \]

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

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

\[ \text{Higher-Order Abstract Syntax} \]

HOAS is the quotient of FOAS by \(\alpha\)-conversion:

\[ \text{HOAS} = \{ [e] \mid e \in \text{FOAS} \} \]

where \([e]\) is the \(\alpha\)-equivalence class of \(e\):

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

\[ \text{deBruijn Indices} \]

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 \(i = \)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 2 + 1
```

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 1 + 2 )+ 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”.

