Syntax

Johannes Åman Pohjola
UNSW

Term 3 2022

Concrete Syntax

Arithmetic Expressions

$$\frac{i \in \mathbb{Z}}{i \ \mathsf{Atom}} \qquad \frac{a \ \mathsf{SExp}}{(a) \ \mathsf{Atom}} \qquad \frac{e \ \mathsf{Atom}}{e \ \mathsf{PExp}} \qquad \frac{e \ \mathsf{PExp}}{e \ \mathsf{SExp}}$$

$$\frac{a \ \mathsf{Atom} \qquad b \ \mathsf{PExp}}{a \times b \ \mathsf{PExp}} \qquad \frac{a \ \mathsf{PExp} \qquad b \ \mathsf{SExp}}{a + b \ \mathsf{SExp}}$$

All the syntax we have seen so far is *concrete syntax*. Concrete syntax is described by judgements on strings.

Abstract Syntax

Working with concrete syntax directly is *unsuitable* for both compiler implementation and proofs. Consider:

- $3 + (4 \times 5)$
- $3 + 4 \times 5$
- $(3 + (4 \times 5))$

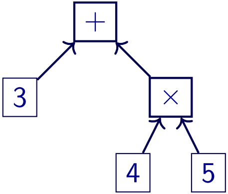
TIMTOWTDI[1] makes life harder for us. Different derivations represent the same semantic program. We would like a representation of programs that is as simple as possible, removing any extraneous information. Such a representation is called *abstract syntax*.

[1] "There is more than one way to do it".

Abstract Syntax

Typically, the *abstract syntax* of a program is represented as a tree rather than as a string.

$$(3 + (4 \times 5)) \quad \longleftrightarrow$$



Writing trees in our inference rules would become unwieldy. We shall define a term language in which to express trees.

Terms

Definition

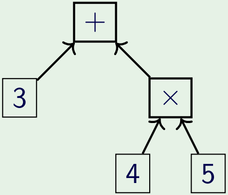
In this course, a *term* is a structure that can either be a symbol, like `Plus` or `Times` or 3; or a compound, which consists of an symbol followed by one or more argument subterms, all in parentheses.

$$t ::= \texttt{Symbol} \mid (\texttt{Symbol}\ t_1\ t_2\ \ldots)$$

These particular terms are also known as *s-expressions*. Terms can equivalently be thought of a subset of Haskell where the only kinds of expressions allowed are literals and data constructors.

Term Examples

Example



`(Plus (Num 3) (Times (Num 4) (Num 5)))`

Armed with an appropriate Haskell `data` declaration, this can be implemented straightforwardly:

```
data Exp  =  Plus Exp Exp
          |  Times Exp Exp
          |  Num Int
```

Concrete to Abstract

Concrete Syntax

$$\frac{i \in \mathbb{Z}}{i\ \mathsf{Atom}} \qquad \frac{a\ \mathsf{SExp}}{(a)\ \mathsf{Atom}} \qquad \frac{e\ \mathsf{Atom}}{e\ \mathsf{PExp}} \qquad \frac{e\ \mathsf{PExp}}{e\ \mathsf{SExp}}$$

$$\frac{a\ \mathsf{Atom} \qquad b\ \mathsf{PExp}}{a \times b\ \mathsf{PExp}} \qquad \frac{a\ \mathsf{PExp} \qquad b\ \mathsf{SExp}}{a + b\ \mathsf{SExp}}$$

Abstract Syntax

$$\frac{i \in \mathbb{Z}}{(\texttt{Num}\ i)\ \mathsf{AST}} \qquad \frac{a\ \mathsf{AST} \qquad b\ \mathsf{AST}}{(\texttt{Plus}\ a\ b)\ \mathsf{AST}} \qquad \frac{a\ \mathsf{AST} \qquad b\ \mathsf{AST}}{(\texttt{Times}\ a\ b)\ \mathsf{AST}}$$

Now we have to specify a *relation* to connect the two!

Relations

Up until now, most judgements we have used have been *unary* — corresponding to a set of satisfying objects.
A judgement can also express a relationship between two objects (a *binary* judgement) or a number of objects (an *n-ary* judgement).

Example (Relations)

- 4 *divides* 16 (binary)
- `mail` *is an anagram of* `liam` (binary)
- 3 *plus* 5 *equals* 8 (ternary)

n-ary judgements where $n \geq 2$ are sometimes called *relations*, and correspond to an n-tuple of satisfying objects.

Parsing Relation

$$
\begin{tikzcd}
3 + (4 \times 5) \arrow[dr, leftrightarrow] & \\
3 + 4 \times 5 \arrow[r, leftrightarrow] & \texttt{(Plus (Num 3) (Times (Num 4) (Num 5)))} \\
(3 + (4 \times 5)) \arrow[ur, leftrightarrow] &
\end{tikzcd}
$$

$$\frac{i \in \mathbb{Z}}{i\ \text{Atom} \longleftrightarrow (\texttt{Num}\ i)\ \text{AST}}$$

$$\frac{a\ \text{Atom} \longleftrightarrow a'\ \text{AST} \qquad b\ \text{PExp} \longleftrightarrow b'\ \text{AST}}{a \times b\ \text{PExp} \longleftrightarrow (\texttt{Times}\ a'\ b')\ \text{AST}}$$

$$\frac{a\ \text{PExp} \longleftrightarrow a'\ \text{AST} \qquad b\ \text{SExp} \longleftrightarrow b'\ \text{AST}}{a + b\ \text{SExp} \longleftrightarrow (\texttt{Plus}\ a'\ b')\ \text{AST}}$$

$$\frac{e\ \text{SExp} \longleftrightarrow a'\ \text{AST}}{(e)\ \text{Atom} \longleftrightarrow a'\ \text{AST}} \qquad \frac{e\ \text{Atom} \longleftrightarrow a\ \text{AST}}{e\ \text{PExp} \longleftrightarrow a\ \text{AST}} \qquad \frac{e\ \text{PExp} \longleftrightarrow a\ \text{AST}}{e\ \text{SExp} \longleftrightarrow a\ \text{AST}}$$

Relations as Algorithms

The *parsing relation* $\longleftrightarrow$ is an extension of our existing concrete syntax rules. Therefore it is unambiguous, just as those rules are. Furthermore, the abstract syntax can be unambiguously determined solely by looking at the left hand side of $\longleftrightarrow$.

An Algorithm

To determine the term corresponding to a particular string:

1. Derive the left hand side of the $\longleftrightarrow$ (the concrete syntax) bottom-up until reaching axioms.
2. Fill in the right hand side of the $\longleftrightarrow$ (the abstract syntax) top-down, starting at the axioms.

This process is called *parsing*.

Example

Rules

$$\frac{i \in \mathbb{Z}}{i\ \mathsf{A} \longleftrightarrow (\texttt{Num}\ i)} \qquad \frac{a\ \mathsf{S} \longleftrightarrow a'}{(a)\ \mathsf{A} \longleftrightarrow a'} \qquad \frac{e\ \mathsf{A} \longleftrightarrow a}{e\ \mathsf{P} \longleftrightarrow a} \qquad \frac{e\ \mathsf{P} \longleftrightarrow a}{e\ \mathsf{S} \longleftrightarrow a}$$

$$\frac{a\ \mathsf{A} \longleftrightarrow a' \qquad b\ \mathsf{P} \longleftrightarrow b'}{a \times b\ \mathsf{P} \longleftrightarrow (\texttt{Times}\ a'\ b')} \qquad \frac{a\ \mathsf{P} \longleftrightarrow a' \qquad b\ \mathsf{S} \longleftrightarrow b'}{a + b\ \mathsf{S} \longleftrightarrow (\texttt{Plus}\ a'\ b')}$$

$$\frac{\dfrac{\overline{1\ \mathsf{A} \longleftrightarrow (\texttt{Num}\ 1)\ \mathsf{AST}}}{1\ \mathsf{P} \longleftrightarrow (\texttt{Num}\ 1)\ \mathsf{AST}} \qquad \dfrac{\dfrac{\overline{2\ \mathsf{A} \longleftrightarrow (\texttt{Num}\ 2)\ \mathsf{AST}} \qquad \dfrac{\overline{3\ \mathsf{A} \longleftrightarrow (\texttt{Num}\ 3)\ \mathsf{AST}}}{3\ \mathsf{P} \longleftrightarrow (\texttt{Num}\ 3)\ \mathsf{AST}}}{2 \times 3\ \mathsf{P} \longleftrightarrow (\texttt{Times}\ (\texttt{Num}\ 2)\ (\texttt{Num}\ 3))\ \mathsf{AST}}}{2 \times 3\ \mathsf{S} \longleftrightarrow (\texttt{Times}\ (\texttt{Num}\ 2)\ (\texttt{Num}\ 3))\ \mathsf{AST}}}{1 + 2 \times 3\ \mathsf{S} \longleftrightarrow (\texttt{Plus}\ (\texttt{Num}\ 1)\ (\texttt{Times}\ (\texttt{Num}\ 2)\ (\texttt{Num}\ 3)))\ \mathsf{AST}}$$

The Inverse

What about the inverse operation to parsing?

Unparsing

Unparsing, also called *pretty-printing*, is the process of starting with the term on the right hand side of $\longleftrightarrow$ and attempting to synthesise a string on the left.

Problem

There are many concrete strings for a given abstract syntax term. The algorithm is *non-deterministic*.

While it is desirable to have:

$$parse \circ unparse = id$$

It is not usually true that:

$$unparse \circ parse = id$$

Example

$3 + (4 \times 5)$

$3 + 4 \times 5$ $\longleftrightarrow$ `(Plus (Num 3) (Times (Num 4) (Num 5)))`

$(3 + (4 \times 5))$

Going from right to left requires some formatting guesswork to produce readable code.

Algorithms to do this can get quite involved!

Let's implement a parser for arithmetic. to coding

Adding Let

Let us extend our arithmetic expression language with variables, including a `let` construct to give them values.

Concrete Syntax

$$\frac{x\ \text{Ident}}{x\ \text{Atom}} \qquad \frac{x\ \text{Ident} \qquad e_1\ \text{SExp} \qquad e_2\ \text{SExp}}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}\ \text{Atom}}$$

Example

$\texttt{let}\ x = 3\ \texttt{in}$
$\quad x + 4$
$\texttt{end}$

$\texttt{let}\ x = 3\ \texttt{in}$
$\quad \texttt{let}\ y = 4\ \texttt{in}\ x + y\ \texttt{end}$
$\texttt{end}$

Scope

binding occurrence of x

scope of x

```
let x = 5 in
  let y = 2 in
    x + y
  end
end
```

usage occurrence of x

The process of finding the binding occurrence of each used variable is called *scope resolution*. Usually this is done statically. If no binding can be found, an *out of scope* error is raised.

Shadowing

What does this program evaluate to?

```
let x = 5 in
  let x = 2 in
    x + x
  end
+ x end
```

x is *shadowed* here

This program results in 9.

α-equivalence

What is the difference between these two programs?

let $x = 5$ in	let $a = 5$ in
let $x = 2$ in	let $y = 2$ in
$x + x$	$y + y$
end	end
end	end

They are semantically identical, but differ in the choice of bound variable names. Such expressions are called *α-equivalent*.

We write $e_1 \equiv_\alpha e_2$ if e_1 is α-equivalent to e_2. The relation $\equiv_\alpha$ is an *equivalence relation*. That is, it is *reflexive*, *transitive* and *symmetric*.
The process of consistently renaming variables that preserves α-equivalence is called *α-renaming*.

Substitution

A variable x is *free* in an expression e if x occurs in e but is not bound in e.

Example (Free Variables)

The variable x is free in $x + 1$, but not in $\texttt{let}\ x = 3\ \texttt{in}\ x + 1\ \texttt{end}$.

A *substitution*, written $e[x := t]$ (or $e[^t/_x]$ in some other courses), is the replacement of all free occurrences of x in e with the term t.

Example (Simple Substitution)

$(5 \times x + 7)[x := y \times 4]$ is the same as $(5 \times (y \times 4) + 7)$.

Problems with substitution

Consider these two α-equivalent expressions.

$$\texttt{let}\ y = 5\ \texttt{in}\ y \times x + 7\ \texttt{end}$$

and

$$\texttt{let}\ z = 5\ \texttt{in}\ z \times x + 7\ \texttt{end}$$

What happens if you apply the substitution $[x := y \times 3]$ to both expressions? You get two non-α-equivalent expressions!

$$\texttt{let}\ y = 5\ \texttt{in}\ y \times (y \times 3) + 7\ \texttt{end}$$

and

$$\texttt{let}\ z = 5\ \texttt{in}\ z \times (y \times 3) + 7\ \texttt{end}$$

This problem is called *capture*.

Variable Capture

Capture can occur for a substitution $e[x := t]$ when a bound variable in e clashes with a free variable occuring in t.

Fortunately

It is always possible to avoid capture.

- α-rename the offending bound variable to an unused name, or
- If you have access to the free variable's definition, renaming the free variable, or
- Use a different abstract syntax representation that makes capture impossible (More on this later).

Abstract Syntax for Variables

We shall extend our AST and parsing relation to include a definition for `let` and variables.

Let Syntax

$$\frac{x\ \mathsf{Ident}}{x\ \mathsf{Atom} \longleftrightarrow (\mathtt{Var}\ x)\ \mathsf{AST}}$$

$$\frac{x\ \mathsf{Ident} \qquad e_1\ \mathsf{SExp} \longleftrightarrow a_1\ \mathsf{AST} \qquad e_2\ \mathsf{SExp} \longleftrightarrow a_2\ \mathsf{AST}}{\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2\ \mathtt{end}\ \mathsf{Atom} \longleftrightarrow (\mathtt{Let}\ x\ a_1\ a_2)\ \mathsf{AST}}$$

First Order Abstract Syntax

Consider the following two pieces of abstract syntax:

```
(Let "x" (Num 5) (Plus (Num 4) (Var "x")))
```

```
(Let "y" (Num 5) (Plus (Num 4) (Var "y")))
```

This demonstrates some problems with our abstract syntax approach.

1. Substitution capture is a problem.
2. α-equivalent expressions are not equal. Determining if an expression is α-equivalent requires us to search for a consistent α-renaming of variables.
3. No distinction is made between binding and usage occurrences of variables. This means that we must define substitution by hand on each type of expression we introduce.
4. Scoping errors cannot be easily detected — malformed syntax is easy to write.

de Bruijn Indices

One popular approach to address the first issue is *de Bruijn indices*.

Key Idea

1. Remove all identifiers from binding expressions like `Let`.
2. Replace the identifier in a `Var` with a number indicating how many binders we must skip in order to find the binder for that variable.

```
(Let "a" (Num 5)
  (Let "y" (Num 2)
    (Plus (Var "a") (Var "y"))))
```

```
(Let (Num 5)
  (Let (Num 2)
    (Plus (Var 1) (Var 0))))
```

Debruijnification

Algorithm

Given a piece of *first order abstract syntax* with explicit variable names, we can convert to de Bruijn indices by keeping a *stack* of variable names, pushing onto the stack at each `Let` and popping after the variable goes out of scope. When a usage occurrence is encountered, replace the variable name with its first position in the stack (starting at the top of the stack).

This approach naturally handles shadowing. It's also possible, but harder, to have de Bruijn indices going in the other direction (from the bottom of the stack, upwards).

de Bruijn Substitution

Substitution is now capture avoiding by definition.

$$
\begin{array}{lcl}
(\texttt{Num}\ i)[n := t] & = & (\texttt{Num}\ i) \\
(\texttt{Plus}\ a\ b)[n := t] & = & (\texttt{Plus}\ a[n := t]\ b[n := t]) \\
(\texttt{Times}\ a\ b)[n := t] & = & (\texttt{Times}\ a[n := t]\ b[n := t]) \\
(\texttt{Var}\ m)[n := t] & = & \begin{cases} t & \text{if } n = m \\ (\texttt{Var}\ (m-1)) & \text{if } m > n \\ (\texttt{Var}\ m) & \text{otherwise} \end{cases} \\
(\texttt{Let}\ e_1\ e_2)[n := t] & = & (\texttt{Let}\ e_1[n := t]\ e_2[n+1 := t_{\uparrow 0}])
\end{array}
$$

Where $e_{\uparrow n}$ is an *up-shifting* operation defined as follows:

$$
\begin{array}{lcl}
(\texttt{Num}\ i)_{\uparrow n} & = & (\texttt{Num}\ i) \\
(\texttt{Plus}\ a\ b)_{\uparrow n} & = & (\texttt{Plus}\ a_{\uparrow n}\ b_{\uparrow n}) \\
(\texttt{Times}\ a\ b)_{\uparrow n} & = & (\texttt{Times}\ a_{\uparrow n}\ b_{\uparrow n}) \\
(\texttt{Var}\ m)_{\uparrow n} & = & \begin{cases} (\texttt{Var}\ (m+1)) & \text{if } m \geq n \\ (\texttt{Var}\ m) & \text{otherwise} \end{cases} \\
(\texttt{Let}\ e_1\ e_2)_{\uparrow n} & = & (\texttt{Let}\ e_{1\uparrow n}\ e_{2\uparrow n+1})
\end{array}
$$

Examining de Bruijn indices

How do de Bruijn indices stack up against explicit names?

1. Substitution capture solved.
2. α-equivalent expressions are now equal.
3. We still must define substitution machinery by hand for each type of expression.
4. It is still possible to make malformed syntax – indices that overflow the stack, for example.

Two out of four isn't bad, but can we do better by changing the term language?

Higher Order Terms

We shall change our term language to include built-in notions of variables and binding.

$$\begin{array}{lcll} t & ::= & \texttt{Symbol} & \textit{(symbols)} \\ & | & x & \textit{(variables)} \\ & | & t_1 \; t_2 & \textit{(application)} \\ & | & x.\; t & \textit{(binding or abstraction)} \end{array}$$

As in Haskell, we shall say that application is left-associative, so

$$(\texttt{Plus}\ (\texttt{Num}\ 3)\ (\texttt{Num}\ 4)) = ((\texttt{Plus}\ (\texttt{Num}\ 3))\ (\texttt{Num}\ 4))$$

Now the binding and usage occurrences of variables are distinguished from regular symbols in our term language. Let's see what this lets us do...

Representing Let

$$\frac{a_1 \ \mathsf{AST} \qquad a_2 \ \mathsf{AST}}{(\mathtt{Let}\ a_1\ (x.\ a_2))\ \mathsf{AST}}$$

We no longer need a rule for variables, because they're baked into the structure of terms.

How would we represent this AST in Haskell?

$$\begin{array}{lcl} \text{data } AST & = & \text{Num } Int \\ & \mid & \text{Plus } AST\ AST \\ & \mid & \text{Times } AST\ AST \\ & \mid & \text{Let } AST\ ???(AST \to AST) \end{array}$$

So $\mathtt{let}\ x = 3\ \mathtt{in}\ x + 2\ \mathtt{end}$ becomes, in Haskell:

$$(\mathtt{Let}\ (\mathtt{Num}\ 3)\ (\lambda x \to \mathtt{Plus}\ x\ (\mathtt{Num}\ 2))$$

Substitution

We can now define substitution across all terms in the meta-logic:

$$
\begin{array}{lcl}
\texttt{Symbol}[x := e] & = & \texttt{Symbol} \\
y[x := e] & = & \begin{cases} e & \text{if } y = x \\ y & \text{otherwise} \end{cases} \\
(t_1\ t_2)[x := e] & = & t_1[x := e]\ t_2[x := e] \\
(y.\ t)[x := e] & = & \begin{cases} (y.\ t) & \text{if } x = y \\ (y.\ t[x := e]) & \text{if } y \notin \mathrm{FV}(e) \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}
$$

Where $\mathrm{FV}(\cdot)$ is the set of all free variables in a term:

$$
\begin{array}{lcl}
\mathrm{FV}(\texttt{Symbol}) & = & \emptyset \\
\mathrm{FV}(x) & = & \{x\} \\
\mathrm{FV}(t_1\ t_2) & = & \mathrm{FV}(t_1) \cup \mathrm{FV}(t_2) \\
\mathrm{FV}(x.\ t) & = & \mathrm{FV}(t)\ \setminus \{x\}
\end{array}
$$

Cheating Outrageously

Substitution capture is still a problem in HOAS. But it is not our problem. Because substitution is defined in the meta-language, it's the job of the implementors of the meta-language (if any) to deal with capture issues.

- When Haskell is our meta-language, it's the job of the GHC developers.
- When we are doing proofs in our meta-logic, there is no implementation, so we can just say that we assume α-equivalent terms to be equal, and therefore assume that variables are always renamed to avoid capture.

So, we have solved the problem by making it someone else's problem. **Outrageous cheating!**

Evaluating All Approaches

	HOAS		FOAS	
	Proofs	Haskell	Strings	de Bruijn
Capture	Cheat	Cheat	Problem	Solved
α-equivalence	Cheat	Cheat	Problem	Solved
Generic subst.	Solved	Solved	Problem	Problem
Malformed syntax	Cheat	Cheat	Problem	Problem

- In embedded languages and in pen and paper proofs, HOAS is very common.
- In conventional language implementations and machine-checked formalisations, de Bruijn indices are more popular.
- In your assignments, strings will be used