Concepts of Programming Languages

Syntax

Lecturer: Gabriele Keller
Tutor: Liam O’Connor
University of New South Wales
School of Computer Sciences & Engineering
Sydney, Australia

COMP 3161/9161 Week 3
Overview

• So far
  ‣ judgements and inference rules
  ‣ rule induction
  ‣ grammars specified using inference rules

• This week
  ‣ relations and inference rules
  ‣ first-order abstract syntax
  ‣ higher-order abstract syntax
  ‣ substitution
Judgements revisited

- A judgement states that a certain property holds for a specific object (which corresponds to a set membership)

- More generally, judgements express a relationship between a number of objects ($n$-ary relations)

- Examples:
  - $4$ divides $16$ (binary relationship)
  - ail is a substring of mail (binary)
  - $3$ plus $5$ equals $8$ (ternary)

- A $n$-ary relation implicitly defines sets of $n$-tuples

  - divides: {$(2,0)$, $(2,2)$, $(2,4)$, ..., $(3,0)$, $(3,3)$, $(3,6)$, ..., $(4,0)$, $(4,4)$, $(4,8)$, ...}
  - substring: {$(“”$,”mail”), $(“m”$,”mail”), $(“ma”$,”mail”), $(“ai”$,”mail”), ...}
  - plus_equal: {$(0,0,0)$, $(0,1,1)$, $(0,2,2)$, ..., $(1,2,3)$, $(2,2,4)$, $(3,2,5)$, .....}
Relations

Definition: A binary relation $R$ is

- **symmetric**, iff for all $a, b$, $a R b$ implies $b R a$
- **reflexive**, iff for all $a$, $a R a$ holds
- **transitive**, iff for all $a, b, c$, $a R b$ and $b R c$ implies $a R c$

Definition:

A relation which is symmetric, reflexive, and transitive is called equivalence relation.
Concrete Syntax

- The inference rules for SExpr defined the concrete syntax of a simple language, including precedence and associativity.

- The concrete syntax of a language is designed with the human user in mind.

- Not adequate for internal representation during compilation.
Concrete vs abstract syntax

• Example:
  ‣ $1 + 2 * 3$
  ‣ $1 + (2 * 3)$
  ‣ $(1) + ((2) * (3))$
  ‣ what is the problem?

• Concrete syntax contains too much information
  ‣ these expressions all have different derivations, but semantically, they represent the same arithmetic expression

• After parsing, we’re just interested in three cases: an expression is either
  ‣ an addition
  ‣ a multiplication or
  ‣ a number
Concrete vs abstract syntax

- we use terms of the form
  
  \((\text{Operator } \text{arg}_1 \text{ arg}_2 \ldots)\)

  to represent parsed programs unambiguously; e.g.,

  \(\text{Plus (Num 1) (Times (Num 2) (Num 3))}\)

- we define the abstract syntax of arithmetic expressions as follows:

  
  \[\begin{align*}
  t_1 \text{ expr} & \quad t_2 \text{ expr} \\
  (\text{Times } t_1 t_2) \text{ expr} & \quad (\text{Plus } t_1 t_2) \text{ expr} \\
  \quad & \quad i \in \text{Int} \\
  (\text{Num } i) \text{ expr}
  \end{align*}\]
Concrete vs abstract syntax

• Parsers

  ‣ check if the program (sequence of tokens) is derivable from the rules of the **concrete syntax**

  ‣ turn the derivation into an **abstract syntax tree**

• Transformation rules

  ‣ we formalise this with inference rules as a binary relation $\leftrightarrow$:

  We write

  $$e \text{ SExpr } \leftrightarrow t \text{ expr}$$

  iff the (concrete syntax) expression $e$ corresponds to the (abstract syntax) expression $t$.

Usually, many different concrete expressions correspond to a single abstract expression
Concrete vs abstract syntax

• Example:

★ \(1 + 2 \times 3\) \(\text{SExpr} \leftrightarrow \text{Plus (Num 1) (Times (Num 2) (Num 3))) expr}\)

★ \(1 + (2 \times 3)\) \(\text{SExpr} \leftrightarrow \text{Plus (Num 1) (Times (Num 2) (Num 3))) expr}\)

★ \((1) + ((2) \times (3))\) \(\text{SExpr} \leftrightarrow \text{Plus (Num 1) (Times (Num 2) (Num 3))) expr}\)
Concrete vs abstract syntax

- Formal definition: we define a parsing relation $\leftrightarrow$ formally as an extension of the structural rules of the concrete syntax.

\[
\begin{align*}
e_1 \text{ SExpr} & \leftrightarrow e_1' \text{ expr} & e_2 \text{ PExpr} & \leftrightarrow e_2' \text{ expr} \\
e_1 + e_2 \text{ SExpr} & \leftrightarrow (\text{Plus } e_1' \ e_2') \text{ expr} \\
e_1 \text{ PExpr} & \leftrightarrow e_1' \text{ expr} & e_2 \text{ FExpr} & \leftrightarrow e_2' \text{ expr} \\
e_1 * e_2 \text{ PExpr} & \leftrightarrow (\text{Times } e_1' \ e_2') \text{ expr} \\
i \text{ FExpr} & \leftrightarrow (\text{Num } i) \text{ expr} \\
i \in \text{ Int}
\end{align*}
\]
The translation relation ↔

• The binary syntax translation relation
  ‣ $e \leftrightarrow e'$

  can be viewed as translation function
  ‣ input is $e$
  ‣ output is $e'$
  ‣ derivations are unambiguously determined by $e$
    ‣ since the grammar of the concrete syntax was unambiguous
  ‣ $e'$ is unambiguously determined by the derivation
    ‣ for each concrete syntax term, there is only one rule we can apply at each step
The translation relation ↔

• Derive the abstract syntax as follows:

(1) **bottom up**, decompose the concrete expression \( e \) according to the left hand side of ↔

(2) **top down**, synthesise the abstract expression \( e' \) according to the right hand side of each ↔ from the rules used in the derivation.

• **Example:** derivation for \( 1 + 2 \times 3 \) (we abbreviate \( SExpr, PExpr, FExpr \) with \( S, P, F \) respectively, and \( expr \) with \( e \))

\[
\begin{align*}
1 \text{ Int} & \quad 2 \text{ Int} \\
\frac{1 \text{ F} \leftrightarrow (\text{Num } 1) \ e}{1 \text{ P} \leftrightarrow (\text{Num } 1) \ e} & \quad \frac{2 \text{ F} \leftrightarrow (\text{Num } 2) \ e}{2 \text{ P} \leftrightarrow (\text{Num } 2) \ e} \quad \frac{3 \text{ Int}}{3 \text{ F} \leftrightarrow (\text{Num } 3) \ e} \\
\frac{1 \text{ S} \leftrightarrow (\text{Num } 1) \ e}{2 \times 3 \text{ P} \leftrightarrow (\text{Times } (\text{Num } 2) \ (\text{Num } 3)) \ e} & \quad \frac{1 + 2 \times 3 \text{ S} \leftrightarrow \text{Plus } (\text{Num } 1)(\text{Times } (\text{Num } 2)(\text{Num } 3)) \ e}
\end{align*}
\]
The parsing problem

Given a sequence of tokens $s \ SExpr$, find $t$ such that

$$s \ SExpr \leftrightarrow t \ expr$$

Requirements

A parser should be

- **total** for all expressions that are correct according to the concrete syntax, that is
  
  - there must be a $t \ expr$ for every $s \ SExpr$

- **unambiguous**, that is for every $t_1$ and $t_2$ with

  - $s \ SExpr \leftrightarrow t_1 \ expr$ and $s \ SExpr \leftrightarrow t_2 \ expr$
  
  we have $t_1 = t_2$
Parsing and pretty printing

• The parsing problem

Given a sequence of tokens \( s \ SExpr \), find \( t \) such that

\[ s \ SExpr \leftrightarrow t \ expr \]

• What about the inverse?

- given \( t \ expr \), find \( s \ SExpr \)

• The inverse of parsing is unparsing

  ▶ unparsing is often ambiguous
  ▶ unparsing is often partial (not total)

• Pretty printing

  • unparsing together with appropriate formatting is called pretty printing

  • due to the ambiguity of unparsing, this will usually not reproduce the original program (but a semantically equivalent one)
Parsing and pretty printing

Example

Given the abstract syntax term

\[
\text{Times (Num 3) (Times (Num 4) (Num 5))}
\]

pretty printing may produce the string

“3 * 4 * 5” or “(3 * 4) * 5”

- it’s best to choose the most simple, readable representation
- but usually, this requires extra effort
Bindings

• Local variable bindings (let)

Let’s extend our simple expression language with one feature

> variables and variable bindings

> let v = e₁ in e₂

• Example:

```
let
    x = 3
in x + 1
```

```
let x = 3
in let y = x + 1
    in x + y
```

• Concrete syntax (adding two new rules):

```
<table>
<thead>
<tr>
<th>id</th>
<th>Ident</th>
<th>e₁</th>
<th>SExpr</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>FExpr</td>
<td>e₂</td>
<td>SExpr</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>let id = e₁ in e₂ FExpr</td>
</tr>
</tbody>
</table>
```
• First order abstract syntax:

\[
\begin{align*}
&\text{Num} \ i \ \text{expr} \\
&\text{Times} \ t_1 \ t_2 \ \text{expr} \\
&\text{Var} \ id \ \text{expr} \\
&\text{Let} \ id \ t_1 \ t_2 \ \text{expr}
\end{align*}
\]
**Bindings**

- **Scope**
  - `let x = e_1 in e_2` introduces -or binds- the variable `x` for use within its scope `e_2`
  - we call the occurrence of `x` in the left-hand side of the binding its **binding occurrence** (or defining occurrence)
  - occurrences of `x` in `e_2` are **usage occurrences**
  - finding the binding occurrence of a variable is called **scope resolution**

- **Two types of scope resolution**
  - **static scoping**: scoping resolution happens at compile time
  - **dynamic scoping**: resolution happens at run time (discussed later in the course)
Bindings

Example:

```plaintext
let
  x = y
in let y = 2
  in x
```

**Out of scope variable:** the first occurrence of `y` is *out of scope*
Bindings

Example:

```
let
  x = 5
in let x = 3
  in x + x
```

**Shadowing:** the inner binding of \( x \) is shadowing the outer binding
\textbf{static scoping}

```c
const int b = 5;
int foo()
{
    int a = b + 5;
    return a;
}

int bar()
{
    int b = 2;
    return foo();
}

int main()
{
    foo(); // returns 10
    bar(); // returns 10
    return 0;
}
```

\textbf{dynamic scoping:}

```c
const int b = 5;
int foo()
{
    int a = b + 5;
    return a;
}

int bar()
{
    int b = 2;
    return foo();
}

int main()
{
    foo(); // returns 10
    bar(); // returns 7
    return 0;
}
```

example from: 

http://msujaws.wordpress.com
Example:

what is the difference between these two expressions?

\[
\begin{align*}
\text{let} & \quad x = 3 \\
in & \quad x + 1
\end{align*}
\quad \text{versus} \quad \\
\begin{align*}
\text{let} & \quad y = 3 \\
in & \quad y + 1
\end{align*}
\]

α-equivalence:

- they only differ in the choice of the bound variable names
- we call them \( \alpha \)-equivalent
- we call the process of consistently changing variable names \( \alpha \)-renaming
- the terminology is due to a conversion rule of the \( \lambda \)-calculus
- we write \( e_1 \equiv_\alpha e_2 \) if two expressions are \( \alpha \)-equivalent
- the relation \( \equiv_\alpha \) is an equivalence relation
Substitution

• Free variables
  ▸ a free variable is one without a binding occurrence
    ▸ let x = 1 in x + y \(\text{y is free in this expression}\)

• Substitution: replacing all occurrences of a free variable \(x\) in an expression \(e\) by another expression \(e'\) is called substitution

• Example: substituting \(x\) with \(2 \times y\) in
  ▸ \(5 \times x + 7\) yields
    ▸ \(5 \times (2 \times y) + 7\)
Substitution

• We have to be careful when applying substitution:
  
  \[
  \text{let } y = 5 \text{ in } y \times x + 7 \quad \text{\textit{\(\alpha\)-equivalent}}
  \]
  
  \[
  \text{let } z = 5 \text{ in } z \times x + 7
  \]
  
  ▷ substitute \(x\) by \(2 \times y\) in both

  \[
  \text{let } y = 5 \text{ in } y \times (2 \times y) + 7
  \quad \text{\textit{\(\alpha\)-equivalent}}
  \]
  
  \[
  \text{let } z = 5 \text{ in } z \times (2 \times y) + 7
  \]
  
  ▷ the free variable \(y\) of \(2 \times y\) is \textit{captured} in the first expression

not \(\alpha\)-equivalent anymore!
Substitution

- **Capture-free substitution**: to substitute $e'$ for $x$ in $e$, we require the free variables in $e'$ to be different from the variables in $e$.

- We can always arrange for a substitution to be capture free:
  - use $\alpha$-renaming of $e'$ (the expression replacing the variable)
  - change all variable names that occur in $e$ and $e'$
  - or use fresh variable names
Higher-order abstract syntax

- A problem with (first-order) abstract syntax

- Defining and usage occurrence of variables are treated the same
  - abstract syntax doesn’t differentiate between binding and using occurrence of a variable
  - it’s difficult to identify $\alpha$-equivalent expressions
  - variables are just terms, like numbers
Higher-order abstract syntax

- **Higher-order abstract syntax** has variables and abstraction as special constructs.

- A term of the form $x.t$ is called an abstraction.

- Structure of a higher-order term: a higher-order term can have one of four forms:

  1. a constant (e.g., int, string)
  2. a variable $x$
  3. (Operator $t_1 ... t_n$)
     - Num 4
     - Plus $x$ (Num 4)
  4. $x.t$ (i.e., the variable $x$ is bound in term $t$)
     - $x$. Plus $x$ (Num 1)
     - $x.y$. Plus $x$ $y$
Higher-order abstract syntax

- Higher-order abstract syntax for let-expressions

**first-order**

- \( \text{id} \) \( \text{Ident} \)
- \( \text{var(id)} \) \( \text{expr} \) \( t_1 \) \( \text{expr} \) \( t_2 \) \( \text{expr} \)
- \( (\text{Var id}) \) \( \text{expr} \)
- \( (\text{Let id } t_1 t_2) \) \( \text{expr} \)

**higher-order**

- \( \text{id} \) \( \text{Ident} \)
- \( t_1 \) \( \text{expr} \)
- \( t_2 \) \( \text{expr} \)
- \( (\text{Let } t_1 \text{id.t}_2) \) \( \text{expr} \)

- Mapping of concrete to higher-order syntax

- \( e_1 \) \( \text{SExpr} \leftrightarrow t_1 \) \( \text{expr} \)
- \( e_2 \) \( \text{SExpr} \leftrightarrow t_2 \) \( \text{expr} \)
- \( \text{let id = e}_1 \text{ in e}_2 \text{ end SExpr} \leftrightarrow (\text{Let } t_1 \text{id.t}_2) \) \( \text{expr} \)
- \( \text{id} \) \( \text{Ident} \)
- \( \text{id} \) \( \text{FExpr} \leftrightarrow \text{id} \) \( \text{expr} \)

- Example:

\[
\text{let } x = 5 \text{ in } x+y \text{ SExpr} \leftrightarrow (\text{Let (Num 5) (x.Plus x y)}) \text{ expr}
\]
Substitution

**Definition: A notation for substitution**

We write

\[ t[x:=t'] \]

to denote a term \( t \) where all the occurrences of \( x \) have been replaced by the term \( t' \).

- **Example:**

\[
(\text{Plus } x \ y) \ [x :=(\text{Num } 1)] = (\text{Plus } (\text{Num } 1) \ y)
\]

**Definition: Renaming**

If we replace a variable in the binding and the body of an abstraction, it is called **renaming**, and the resulting term is \( \alpha \)-equivalent to the original term:

\[
x.t \equiv_\alpha y.t \ [x := y]
\]

if \( y \) doesn't occur free in \( t \) (or \( y \not\in \text{FV}(t) \))
Substitution

• A inductive definition of $FV(t)$:

\[
\begin{align*}
FV(x) & = \{x\} \\
FV(Op \ t_1 \ldots \ t_n) & = FV(t_1) \cup \ldots \cup FV(t_n) \\
FV(x.t) & = FV(t) \setminus \{x\}
\end{align*}
\]

• Substituting one variable by another:

\[
\begin{align*}
x[x:=y] & = y \\
z[x:=y] & = z, \text{ if } z \neq x \\
(Op \ t_1 \ldots t_n)[x:=y] & = Op \ (t_1[x:=y]) \ldots (t_n[x:=y]) \\
x.t[x:=y] & = x.t \\
z.t[x:=y] & = z \cdot (t[x:=y]), \text{ if } x \neq z, y \neq z \\
y.t[x:=y] & = \text{undefined, if } x \neq y
\end{align*}
\]
Substitution

• Substituting a variable by a term u:

\[ x [x:=u] = u \]
\[ z [x:=u] = z, \text{ if } z \neq x \]
\[ (\text{Op } t_1 \ldots t_n) [x:=u] = \text{Op } (t_1 [x:=u]) \ldots (t_n [x:=u]) \]
\[ x.t [x:=u] = x.t \]
\[ z.t [x:=u] = z. (t [x:=u]), \text{ if } x \neq z, z \notin \text{FV}(u) \]
\[ y.t [x:=u] = \text{undefined, if } y \in \text{FV}(u) \]