Utrecht University

Concepts of Programming Language Design

Syntax

Gabriele Keller
Tom Smeding

Overview

- So far
    - revision of inference rules, natural (rule) induction
    - simple languages specified using inference rules
    - proofs on inductively defined languages

- This week:
    - concrete syntax of a language
    - first-order & higher-order abstract syntax,
    - static and dynamic semantics

Utrecht University

Concrete Syntax

- the inference rules for $\boldsymbol{SExpr}$ 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

$$\frac{e_1\ \boldsymbol{SExpr} \quad e_2\ \boldsymbol{PExpr}}{e_1\ \texttt{+}\ e_2\ \boldsymbol{SExpr}}$$

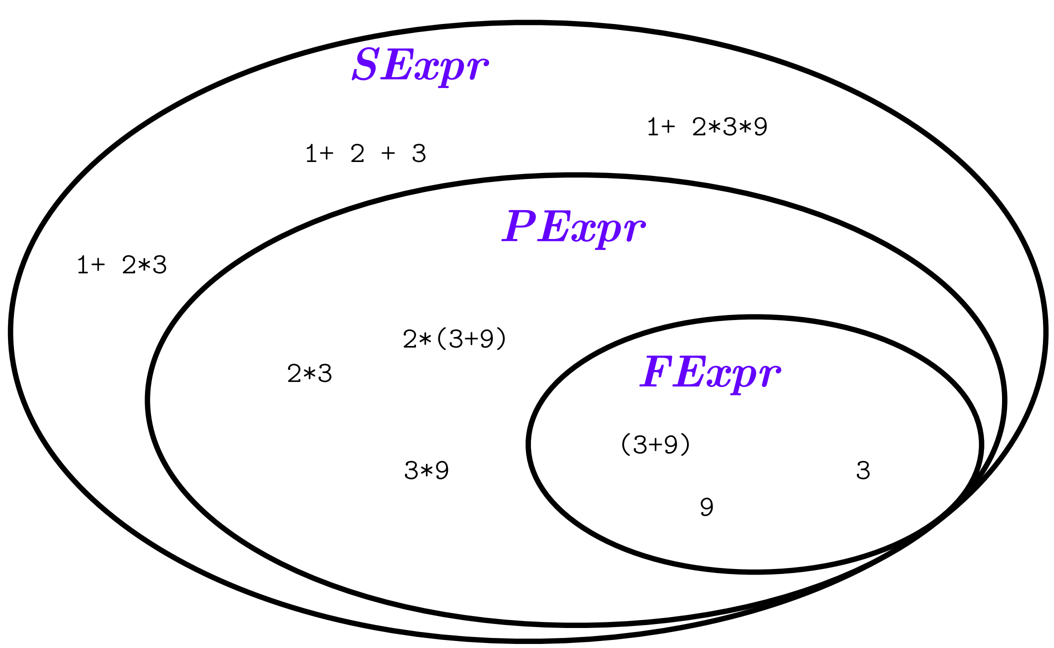
$$\frac{e\ \boldsymbol{PExpr}}{e\ \boldsymbol{SExpr}}$$

$$\frac{e_1\ \boldsymbol{PExpr} \quad e_2\ \boldsymbol{FExpr}}{e_1\ \texttt{*}\ e_2\ \boldsymbol{PExpr}}$$

$$\frac{e\ \boldsymbol{FExpr}}{e\ \boldsymbol{PExpr}}$$

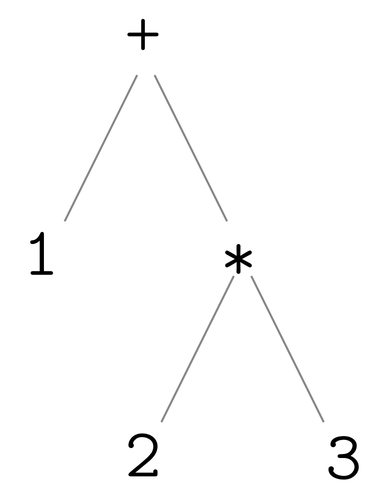
$$\frac{e\ \boldsymbol{SExpr}}{\texttt{(}e\texttt{)}\ \boldsymbol{FExpr}}$$

$$\frac{n \in \boldsymbol{Int}}{n\ \boldsymbol{FExpr}}$$



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

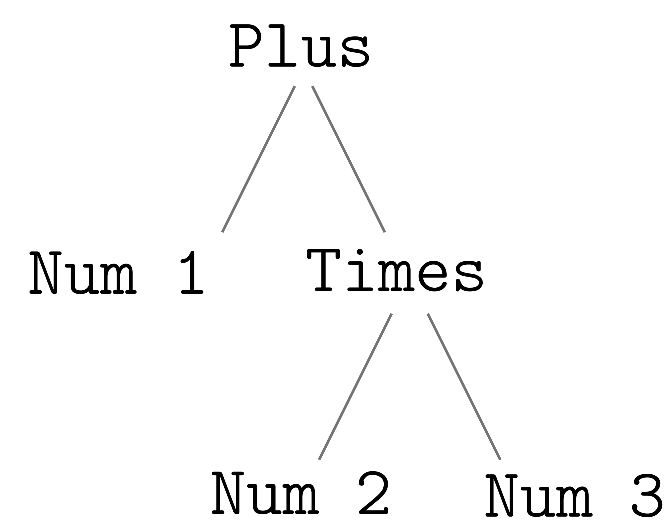
Concrete vs abstract syntax

- we use Haskell style terms of the form

  $$(\mathit{Operator}\ \mathit{arg}_1\ \ \mathit{arg}_2 \ldots.)$$

  to represent parsed programs unambiguously; e.g.,

  `(Plus (Num 1) (Times (Num 2) (Num 3)))`



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

$$\frac{i \in \boldsymbol{Int}}{(\texttt{Num}\ i\ )\ \boldsymbol{expr}}$$

$$\frac{t_1\ \boldsymbol{expr} \qquad t_2\ \boldsymbol{expr}}{(\texttt{Plus}\ t_1\ t_2)\ \ \boldsymbol{expr}}$$

$$\frac{t_1\ \boldsymbol{expr} \qquad t_2\ \boldsymbol{expr}}{(\texttt{Times}\ t_1\ t_2)\ \ \boldsymbol{expr}}$$

Concrete vs abstract syntax

- BNF:

$$\mathit{expr} ::= (\texttt{Num Int}) \mid (\texttt{Plus}\ \mathit{expr}\ \mathit{expr}) \mid (\texttt{Times}\ \mathit{expr}\ \mathit{expr})$$

- Abstract syntax terms of the form

  $(\mathit{Operator}\ \mathit{arg}_1\ \ \mathit{arg}_2 \ldots.)$

  can directly be translated into Haskell data types:

```haskell
data Expr
  = Num   Int
  | Plus  Expr Expr
  | Times Expr Expr
```

$$\frac{i \in \mathit{Int}}{(\texttt{Num}\ i\ )\ \mathit{expr}}$$

$$\frac{t_1\ \mathit{expr} \qquad t_2\ \mathit{expr}}{(\texttt{Plus}\ t_1\ t_2)\ \ \mathit{expr}}$$

$$\frac{t_1\ \mathit{expr} \qquad t_2\ \mathit{expr}}{(\texttt{Times}\ t_1\ t_2)\ \ \mathit{expr}}$$

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 (AST)
- Transformation rules
  - we formalise this with inference rules as a binary relation $\leftrightarrow$:

We write

$$e \; SExpr \leftrightarrow t \; expr$$

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

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

Utrecht University

Concrete vs abstract syntax

- Example:
  - `1 + 2 * 3` $\boldsymbol{SExpr}$ $\leftrightarrow$ `(Plus (Num 1) (Times (Num 2)(Num 3)))` $\boldsymbol{expr}$
  - `1 + (2 * 3)` $\boldsymbol{SExpr}$ $\leftrightarrow$ `(Plus (Num 1) (Times (Num 2)(Num 3)))` $\boldsymbol{expr}$
  - `(1) + ((2)*(3))` $\boldsymbol{SExpr}$ $\leftrightarrow$ `(Plus (Num 1) (Times (Num 2)(Num 3)))` $\boldsymbol{expr}$

Concrete vs abstract syntax

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

Concrete vs abstract syntax

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

$$\frac{e_1\ \mathit{SExpr} \leftrightarrow e_1'\ \mathit{expr} \qquad e_2\ \mathit{PExpr} \leftrightarrow e_2'\ \mathit{expr}}{e_1 + e_2\ \mathit{SExpr} \leftrightarrow \texttt{(Plus}\ e_1'\ e_2'\texttt{)}\ \mathit{expr}}$$

$$\frac{e\ \mathit{PExpr} \leftrightarrow e'\ \mathit{expr}}{e\ \mathit{SExpr} \leftrightarrow e'\ \mathit{expr}}$$

$$\frac{e_1\ \mathit{PExpr} \leftrightarrow e_1'\ \mathit{expr} \qquad e_2\ \mathit{FExpr} \leftrightarrow e_2'\ \mathit{expr}}{e_1 * e_2\ \mathit{PExpr} \leftrightarrow \texttt{(Times}\ e_1'\ e_2'\texttt{)}\ \mathit{expr}}$$

$$\frac{e\ \mathit{FExpr} \leftrightarrow e'\ \mathit{expr}}{e\ \mathit{PExpr} \leftrightarrow e'\ \mathit{expr}}$$

$$i \in \mathit{Int} \qquad \frac{}{i\ \mathit{FExpr} \leftrightarrow \texttt{(Num}\ i\texttt{)}\ \mathit{expr}}$$

$$\frac{e\ \mathit{SExpr} \leftrightarrow e'\ \mathit{expr}}{\texttt{(}e\texttt{)}\ \mathit{FExpr} \leftrightarrow e'\ \mathit{expr}}$$

The translation relation $\leftrightarrow$

- 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 valid concrete syntax term, there is only one rule we can apply at each step of the full proof

The translation relation $\leftrightarrow$

- Derive the abstract syntax as follows:

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

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

- Example: derivation for `1 + 2 * 3`

$$\frac{e_1\ \mathit{SExpr} \leftrightarrow e_1' \qquad e_2\ \mathit{PExpr} \leftrightarrow e_2'}{e_1 + e_2\ \mathit{SExpr} \leftrightarrow (\texttt{Plus}\ e_1'\ e_2')}$$

$$\frac{e\ \mathit{PExpr} \leftrightarrow e'}{e\ \mathit{SExpr} \leftrightarrow e'}$$

$$\frac{e_1\ \mathit{PExpr} \leftrightarrow e_1' \qquad e_2\ \mathit{FExpr} \leftrightarrow e_2'}{e_1 * e_2\ \mathit{PExpr} \leftrightarrow (\texttt{Times}\ e_1'\ e_2')}$$

$$\frac{e\ \mathit{FExpr} \leftrightarrow e'}{e\ \mathit{PExpr} \leftrightarrow e'}$$

$$i \in \mathit{Int} \qquad \frac{}{i\ \mathit{FExpr} \leftrightarrow (\texttt{Num}\ i)}$$

$$\frac{e\ \mathit{SExpr} \leftrightarrow e'}{(e)\ \mathit{FExpr} \leftrightarrow e'}$$

$$\frac{\dfrac{\dfrac{\dfrac{}{1\ \mathit{FExpr} \leftrightarrow (\texttt{Num 1})}}{1\ \mathit{PExpr} \leftrightarrow (\texttt{Num 1})}}{1\ \mathit{SExpr} \leftrightarrow (\texttt{Num 1})} \qquad \dfrac{\dfrac{\dfrac{}{2\ \mathit{FExpr} \leftrightarrow (\texttt{Num 2})}}{2\ \mathit{PExpr} \leftrightarrow (\texttt{Num 2})} \qquad \dfrac{}{3\ \mathit{FExpr} \leftrightarrow (\texttt{Num 3})}}{2*3\ \mathit{PExpr} \leftrightarrow (\texttt{Times (Num 2) (Num 3)})}}{1 + 2*3\ \mathit{SExpr} \leftrightarrow (\texttt{Plus (Num 1) (Times (Num 2) (Num 3))})}$$

Concrete vs abstract syntax

- Do the rules actually specify a deterministic algorithm?

$$\frac{e_1\ \mathit{SExpr} \leftrightarrow e_1'\ \mathit{expr} \qquad e_2\ \mathit{PExpr} \leftrightarrow e_2'\ \mathit{expr}}{e_1 + e_2\ \mathit{SExpr} \leftrightarrow \texttt{(Plus}\ e_1'\ e_2'\texttt{)}\ \mathit{expr}}$$

$$\frac{e\ \mathit{PExpr} \leftrightarrow e'\ \mathit{expr}}{e\ \mathit{SExpr} \leftrightarrow e'\ \mathit{expr}}$$

$$\frac{e_1\ \mathit{PExpr} \leftrightarrow e_1'\ \mathit{expr} \qquad e_2\ \mathit{FExpr} \leftrightarrow e_2'\ \mathit{expr}}{e_1 * e_2\ \mathit{PExpr} \leftrightarrow \texttt{(Times}\ e_1'\ e_2'\texttt{)}\ \mathit{expr}}$$

$$\frac{e\ \mathit{FExpr} \leftrightarrow e'\ \mathit{expr}}{e\ \mathit{PExpr} \leftrightarrow e'\ \mathit{expr}}$$

$$\frac{i \in \mathit{Int}}{i\ \mathit{FExpr} \leftrightarrow \texttt{(Num}\ i\texttt{)}\ \mathit{expr}}$$

$$\frac{e\ \mathit{SExpr} \leftrightarrow e'\ \mathit{expr}}{\texttt{(}e\texttt{)}\ \mathit{FExpr} \leftrightarrow e'\ \mathit{expr}}$$

Parsing and inference rules

- The parsing problem

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

  $$s\ \mathit{SExpr} \leftrightarrow t\ \mathit{expr}$$

- 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
  - $s\ \mathit{SExpr} \leftrightarrow t_1\ \mathit{expr}$ and $s\ \mathit{SExpr} \leftrightarrow t_2\ \mathit{expr}$

    we have $t_1 = t_2$

Utrecht University

Parsing and pretty printing

- The parsing problem

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

  $$s\ \boldsymbol{SExpr} \leftrightarrow t\ \boldsymbol{expr}$$

- What about the inverse?
  - given t $\boldsymbol{expr}$, find s $\boldsymbol{SExpr}$
- 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

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

pretty printing may produce the string

`3 * 4 * 5` *or* `(3 * 4) * 5`

- it's best to chose the most simple, readable representation
- but usually, this requires extra effort

Bindings

- Local variable bindings (let)

  Let's extend our simple expression language with

  - variables and variable bindings
  - `let` v `=` e_1 `in` e_2 `end`

- Example:

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

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

- Concrete syntax (adding two new rules):

$$\frac{id\ \ \mathit{Ident}}{id\ \ \mathit{FExpr}} \qquad \frac{e_1\ \ \mathit{SExpr} \qquad e_2\ \ \mathit{SExpr}}{\texttt{let}\ id = e_1\ \texttt{in}\ e_2\ \texttt{end}\ \ \ \mathit{FExpr}}$$

Bindings

The `end` keyword is necessary for nested `let`-expressions:

```
let
  x = 3
in 2 * let y = 5 in y + x
```

we'll leave it out when not needed to disambiguate

Bindings

- First-order abstract syntax:

$$\frac{i \in \mathit{Int}}{(\texttt{Num}\ i\ )\ \mathit{expr}}$$

$$\frac{\mathit{id}\ \ \mathit{Ident}}{(\texttt{Var}\ \ \mathit{id})\ \ \mathit{expr}}$$

$$\frac{t_1\ \mathit{expr} \qquad t_2\ \mathit{expr}}{(\texttt{Plus}\ t_1\ t_2)\ \ \mathit{expr}}$$

$$\frac{\mathit{id}\ \ \mathit{Ident} \qquad t_1\ \mathit{expr} \qquad t_2\ \mathit{expr}}{(\texttt{Let}\ \ \mathit{id}\ \ t_1\ \ t_2)\ \ \mathit{expr}}$$

$$\frac{t_1\ \mathit{expr} \qquad t_2\ \mathit{expr}}{(\texttt{Times}\ t_1\ \ t_2)\ \ \mathit{expr}}$$

Bindings

- Scope
  - `let` x `=` e_1 `in` e_2 `end` 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 (or lexical) scoping***: scoping resolution happens at compile time
  - ***dynamic scoping***: resolution happens at run time

Bindings

Example:

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

scope of y

scope of 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

Scope

- Where the scope starts differs in different languages:

In C:

```
void f () {
 …
 int x = 5;
 int y = x;
 … }
```

scope of x

JavaScript:

```
function showMsg () {
  console.log(msg);
  …
  var msg = ‘‘hi’’;
  …
}
```

scope of msg

In Haskell:

```
let
  y = x
  x = 5
  …
in …
```

scope of x

```
…
where
  y = x
  x = 5
  …
```

Utrecht University

Scope

- Scope resolution
  - resolving which usage occurrence of a variable belongs to which defining occurrence
- Static scope
  - scope is determined according to where in the program (text) the variable is used
- Dynamic scope
  - scope is determined according when during the program execution a variable is used

Dynamic vs static scoping

```
int x = 10;

// Called by g()
int f()
{
   return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
   int x = 20;
   return f();
}

main()
{
  printf(f());
  printf(g());
}
```

example from Vineet Joshi, https://www.geeksforgeeks.org/static-and-dynamic-scoping/

Bindings

Example:

what is the difference between these two expressions?

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

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

α-equivalence:

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

Substitution

- Free variables
  - a free variable is one without a binding occurrence

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

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 + y` in

```
5 * x + 7
```

yields

```
5 * (2 + y) + 7
```

Substitution

- We have to be careful when applying substitution:
  - `let y = 5 in y * x + 7`
  - `let z = 5 in z * x + 7`

α-equivalent

- substitute `x` by `2 * y` in both
  - `let y = 5 in y * (2 * y) + 7`
  - `let z = 5 in z * (2 * y) + 7`

not α-equivalent anymore!

- the free variable `y` of `2 * y` is captured in the first expression

Substitution

- ***Capture-free substitution***: to substitute e' for x in e we require the free variables in e' to be different from the bound variables in e
- We a can always arrange for a substitution to be capture free
  - α-rename e

Higher-order abstract syntax

- Limitations of (first-order) abstract syntax

$$\frac{i \in \mathit{Int}}{(\texttt{Num}\ i)\ \mathit{expr}} \qquad \frac{\mathit{id}\ \mathit{Ident}}{(\texttt{Var}\ \mathit{id})\ \mathit{expr}}$$

$$\frac{t_1\ \mathit{expr} \quad t_2\ \mathit{expr}}{(\texttt{Plus}\ t_1\ t_2)\ \mathit{expr}} \qquad \frac{\mathit{id}\ \mathit{Ident} \quad t_1\ \mathit{expr} \quad t_2\ \mathit{expr}}{(\texttt{Let}\ \mathit{id}\ t_1\ t_2)\ \mathit{expr}}$$

$$\frac{t_1\ \mathit{expr} \quad t_2\ \mathit{expr}}{(\texttt{Times}\ t_1\ t_2)\ \mathit{expr}}$$

- Defining and usage occurrence of variables are treated the same
  - abstract syntax doesn't differentiate between ***binding*** and ***using*** occurrence of a variable, scope isn't clear from the syntax (see `Let`)
  - it's difficult to identify α-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 $(x.e)$ is called ***an abstraction***
  - the higher-order variable x is bound in the term e (i.e., the scope x of is e)
- variables and abstraction are a built-in feature of higher-order syntax

Higher-order abstract syntax

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

  (1) a constant (e.g., int, string)

  (2) a variable x

  (3) ($Operator\ e_1 \ \ldots\ e_n$)

  - (Num 4)
  - (Plus x (Num 4))

  (4) an abstraction (x. e)

  - (x. Plus x (Num 1))
  - (x.(y. Plus x y)

Higher-order abstract syntax

- Higher-order abstract syntax for let-expressions

first-order

$$\frac{id\ \ Ident}{(\texttt{Var}\ \ id)\ \ expr} \qquad \frac{id\ \ Ident \qquad t_1\ expr \qquad t_2\ expr}{(\texttt{Let}\ \ id\ \ t_1\ \ t_2)\ \ expr}$$

higher-order

$$\frac{}{id\ \ expr} \qquad \frac{t_1\ expr \qquad t_2\ expr}{(\texttt{Let}\ t_1\ \ (id.t_2))\ \ expr}$$

- Mapping of concrete to higher-order syntax

$$\frac{e_1\ \ SExpr \leftrightarrow t_1\ \ expr \qquad e_2\ \ SExpr \leftrightarrow t_2\ \ expr}{\texttt{let}\ \ id = e_1\ \texttt{in}\ e_2\ \texttt{end}\ \ SExpr \ \leftrightarrow\ (\texttt{Let}\ \ t_1\ (id.t_2))\ expr}$$

$$\frac{id\ Ident}{id\ FExpr \leftrightarrow id\ expr}$$

- Example:

`let x = 5 in x+y` $SExpr \leftrightarrow$ `(Let (Num 5) (`x`.Plus` x y`))` $expr$

Higher-order abstract syntax

- Some observations:

```
let  z = x          let  x = x
in   z              in   x
```

- First-order abstract syntax

```
(Let ‘‘z’’ (Var ‘‘x’’) (Var ‘‘z’’))          (Let ‘‘x’’ (Var ‘‘x’’) (Var ‘‘x’’))
```

these terms may or may not be α-equivalent:
depends on what the Let-term actually means

- Higher-order abstract syntax

(Let x ($z.z$))   (Let x ($x.x$))

..

these terms are definitely α-equivalent:
not relevant what the Let-term actually means

Substitution

Definition: *A notation for substitution*

We write

$$t[x := t']$$

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

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

$$(\texttt{Let}\ x\ (x.x))\ [x := (\texttt{Num 1})] = (\texttt{Let (Num 1)}\ (x.x))$$

Substitution

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 α-equivalent to the original term:

$$(x.t) \equiv_\alpha (y.(t\,[x := y]))$$

if y doesn't occur free in t (i.e., or $y \notin FV\,(t)$)

$(x.$ `(Plus` x `(Num 1))` $\equiv_\alpha$ $(y.$ `(Plus` y `(Num 1))`

Utrecht University

Substitution

- A inductive definition of FV(t):

$$FV\ (x) = \{x\}$$

$$FV\ ((Op\ t_1\ \dots\ t_n)) = FV(t_1) \cup \dots . \cup FV(t_n)$$

$$FV\ ((x.t)) = FV\ (t) \setminus \{x\}$$

- Again, we can use one definition of free variables for any language represented in higher-order abstract syntax
- With first-order, we have to define how to calculate the set of free variables for every concrete language we are looking at

Utrecht University

Substitution

- Substituting a free variable by another free variable:

$$x[x := y] \qquad = \; y$$

Utrecht University

Substitution

- Substituting a free variable by another free variable:

$$
\begin{aligned}
x[x := y] &= y \\
z[x := y] &= z, \;\; \mathit{if}\; x \neq z \\
(Op\; t_1 \ldots t_n)[x := y] &= (Op\; t_1[x := y] \;\ldots\; t_n[x := y]) \\
(x.t)\,[x := y] &= (x.t) \\
(z.t)\,[x := y] &= (z.t[x := y]) \;\; \mathit{if}\; x \neq z,\; y \neq z, \\
(y.t)\,[x := y] &= \mathit{undefined}\; \mathit{if}\; x \neq y
\end{aligned}
$$

Substitution

- Substituting a variable by a term u:

$$x[x := u] \qquad = \; u$$

Substitution

- Substituting a variable by a term u:

$$
\begin{array}{lll}
x[x := u] & = & u \\
z[x := u] & = & z, \quad \mathit{if} \ x \neq z \\
(Op\ t_1 \ldots t_n)[x := u] & = & (Op\ t_1[x := u] \ldots\ t_n[x := u]) \\
(x.t)[x := u] & = & (x.t) \\
(z.t)[x := u] & = & z.t[x := u], \ \mathit{if} \ x \neq z, \ z \notin FV(u). \\
(y.t)[x := u] & = & \mathit{undefined}, \ \mathit{if} \ y \in FV(u)
\end{array}
$$