Universiteit Utrecht

[Faculty of Science
Information and Computing Sciences]

Compiler Construction

WWW: `http://www.cs.uu.nl/wiki/Cco`

Edition 2011/2012

Agenda

7. Abstraction

Abstraction

§7

So far, the programming languages we have considered have been very simple: they did not provide much more than some constants and some primitive operations over these.

For a programming language to be useful, it should at least provide some mechanism to abstract away from constants and to express computations in terms of the obtained abstractions.

Hence, we will now study the syntax, semantics, and implementation of some of these mechanisms.

7.1 Local definitions

Variables

We assume a countable infinite set of variables $\mathbf{Var} = \{\cdots, x, y, z, \cdots\}$.

$x \in \mathbf{Var}$ variables

Infinite: so we can also pick a new ("fresh") variable.

Countable: so we don't have too many of them.

☞ We use x both as a metavariable (ranging over the complete set $\mathbf{Var}$) and as an object variable (being an element of the set $\mathbf{Var}$).

Local definitions

We first study a simple language containing a local-definition construct as found in most functional programming languages:

n	$\in$	$\mathbf{Num}$	numerals
t	$\in$	$\mathbf{Tm}$	terms

$$t ::= n \mid x \mid \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} \mid t_1 + t_2$$

Of course, the language can easily be extended with other constructs such as boolean constants, conditionals, and additional binary operators.

Local definitions: example

As a simple example, consider the program

$$\begin{array}{l} \mathbf{let} \\ \quad x = 2 \\ \mathbf{in} \\ \quad x + x \\ \mathbf{ni} \end{array}$$

evaluating to

$$4$$

Unbound variables

As defined, our term language contains some perculiar anomalies.

For example:

$$\begin{array}{l}\textbf{let}\\ \quad x = 2\\ \textbf{in}\\ \quad x + y\\ \textbf{ni}\end{array}$$

The variable y is not defined anywhere: we say that y is unbound.

Binders

The local-definition construct

$$\textbf{let } x = t_1 \textbf{ in } t_2 \textbf{ ni}$$

is a binder for x: we say that x is bound in t_2, i.e., occurrences of x in t_2 either refer to the definition $x = t_1$ or to a nested definition for x in t_2.

If a variable is not bound in a term, we say it occurs free.

☞ x is not bound in t_1 (unless by some outer local definition), i.e., the local definitions we consider are, in contrast to those in, for example, Haskell, *nonrecursive*.

Free variables

Let us now formally define the free variables of a term by means of a metafunction *fv* : $\mathbf{Tm} \to \mathcal{P}(\mathbf{Var})$.

$$\begin{array}{ll}\mathit{fv}(n) & = \{\,\}\\ \mathit{fv}(x) & = \{\,x\,\}\\ \mathit{fv}(\textbf{let } x = t_1 \textbf{ in } t_2 \textbf{ ni}) & = \mathit{fv}(t_1) \cup \mathit{fv}(t_2)\backslash\{\,x\,\}\\ \mathit{fv}(t_1 + t_2) & = \mathit{fv}(t_1) \cup \mathit{fv}(t_2)\end{array}$$

☞ $\mathcal{P}(\mathbf{Var})$ denotes the *power set* of $\mathbf{Var}$, i.e., the set containing all subsets of $\mathbf{Var}$.

Alpha-equivalence

The terms

$$\textbf{let } x = 2 \textbf{ in } x + x \textbf{ ni}$$

and

$$\textbf{let } y = 2 \textbf{ in } y + y \textbf{ ni}$$

essentially denote the same program.

When two programs only differ in the names of their bound variables, we say that they are alpha-equivalent or “equal up to alpha-conversion”.

☞ Alpha-conversion is the process of consistently renaming bound variables without changing the semantics of a term.

Shadowing

The terms

$\textbf{let } x = 2 \textbf{ in let } y = 3 \textbf{ in } x + y \textbf{ ni ni}$

and

$\textbf{let } x = 2 \textbf{ in let } x = 3 \textbf{ in } x + x \textbf{ ni ni}$

are not alpha-equivalent, for renaming the bound variable y in the first term to x changes the semantics of the program.

In the second term the inner binding of x shadows the outer binding.

Beta-substitution

Beta-substitution is the process of replacing all free occurences of a variable in a given term by another term, *without binding any of the free variables in the substitute, performing alpha-conversion when necessary*.

Notation: $[x \mapsto t_0]t$. ("Substitute t_0 for x in t.")

For example:

$[x \mapsto 2](x + y) = 2 + y$

Beta-substitution: first attempt

Let us try to formally define beta-substitution as a metaoperation $[\cdot \mapsto \cdot]\cdot : \mathbf{Var} \to \mathbf{Tm} \to \mathbf{Tm} \to \mathbf{Tm}$.

$$
\begin{array}{lll}
[x \mapsto t_0]n & = n & \\
[x \mapsto t_0]x_0 & = t_0 & \text{if } x = x_0 \\
[x \mapsto t_0]x_0 & = x_0 & \text{if } x \neq x_0 \\
[x \mapsto t_0](\textbf{let } x_0 = t_1 \textbf{ in } t_2 \textbf{ ni}) = & & \\
\qquad \textbf{let } x_0 = [x \mapsto t_0]t_1 \textbf{ in } [x \mapsto t_0]t_2 \textbf{ ni} & & \\
[x \mapsto t_0](t_1 + t_2) & = [x \mapsto t_0]t_1 + [x \mapsto t_0]t_2 &
\end{array}
$$

Indeed:

$[x \mapsto 2](x + y) = 2 + y$

But also:

$[x \mapsto 2](x + \textbf{let } x = 3 \textbf{ in } x \textbf{ ni}) = 2 + \textbf{let } x = 3 \textbf{ in } 2 \textbf{ ni}$

☞ Beta-substitution now erroneously replaces *bound* occurrences of variables as well.

Beta-substitution: second attempt

$$
\begin{array}{lll}
[x \mapsto t_0]n & = n & \\
[x \mapsto t_0]x_0 & = t_0 & \text{if } x = x_0 \\
[x \mapsto t_0]x_0 & = x_0 & \text{if } x \neq x_0 \\
[x \mapsto t_0](\textbf{let } x_0 = t_1 \textbf{ in } t_2 \textbf{ ni}) = & & \\
\qquad \textbf{let } x_0 = [x \mapsto t_0]t_1 \textbf{ in } [x \mapsto t_0]t_2 \textbf{ ni} & & \text{if } x \neq x_0 \\
[x \mapsto t_0](\textbf{let } x_0 = t_1 \textbf{ in } t_2 \textbf{ ni}) = & & \\
\qquad \textbf{let } x_0 = [x \mapsto t_0]t_1 \textbf{ in } t_2 \textbf{ ni} & & \text{if } x = x_0 \\
[x \mapsto t_0](t_1 + t_2) & = [x \mapsto t_0]t_1 + [x \mapsto t_0]t_2 &
\end{array}
$$

Indeed:

$[x \mapsto 2](x + \textbf{let } x = 3 \textbf{ in } x \textbf{ ni}) = 2 + \textbf{let } x = 3 \textbf{ in } x \textbf{ ni}$

But also:

$[x \mapsto y](x + \textbf{let } y = 3 \textbf{ in } x + y \textbf{ ni}) = y + \textbf{let } y = 3 \textbf{ in } y + y \textbf{ ni}$

☞ Beta-substitution can erroneously *capture* free variables of the substitute.

Beta-substitution: final solution

$$
\begin{array}{lll}
[x \mapsto t_0]n & = n & \\
[x \mapsto t_0]x_0 & = t_0 & \text{if } x = x_0 \\
[x \mapsto t_0]x_0 & = x_0 & \text{if } x \neq x_0 \\
[x \mapsto t_0](\mathbf{let}\ x_0 = t_1\ \mathbf{in}\ t_2\ \mathbf{ni}) = & & \\
\quad \mathbf{let}\ x_0 = [x \mapsto t_0]t_1\ \mathbf{in}\ [x \mapsto t_0]t_2\ \mathbf{ni} & & \\
 & & \text{if } x \neq x_0 \text{ and } x_0 \notin \mathit{fv}(t_0) \\
[x \mapsto t_0](\mathbf{let}\ x_0 = t_1\ \mathbf{in}\ t_2\ \mathbf{ni}) = & & \\
\quad \textit{let } x_0' \textit{ be fresh} & & \\
\quad \textit{in } \mathbf{let}\ x_0' = [x \mapsto t_0]t_1\ \mathbf{in}\ [x \mapsto t_0][x_0 \mapsto x_0']t_2\ \mathbf{ni} & & \\
 & & \text{if } x \neq x_0 \text{ and } x_0 \in \mathit{fv}(t_0) \\
[x \mapsto t_0](\mathbf{let}\ x_0 = t_1\ \mathbf{in}\ t_2\ \mathbf{ni}) = & & \\
\quad \mathbf{let}\ x_0 = [x \mapsto t_0]t_1\ \mathbf{in}\ t_2\ \mathbf{ni} & & \text{if } x = x_0 \\
[x \mapsto t_0](t_1 + t_2) & = [x \mapsto t_0]t_1 + [x \mapsto t_0]t_2 &
\end{array}
$$

Indeed:

$$[x \mapsto y](x + \mathbf{let}\ y = 3\ \mathbf{in}\ x + y\ \mathbf{ni}) = y + \mathbf{let}\ z = 3\ \mathbf{in}\ y + z\ \mathbf{ni}$$

- ☞ A fresh variable is just a variable that differs from all other variables in a program. In particular we have: $x_0' \notin \mathit{fv}(t_0)$.
- ☞ In principle, we could do without the first clause for local definitions and drop the check for $x_0 \notin \mathit{fv}(t_0)$.

Natural semantics

With beta-substitution in place, we can now define a natural (i.e., big-step) semantics for our language.

The values in our language are just the numerals:

$$v \in \mathbf{Val} \quad \text{values}$$

$$v ::= n$$

As always, our natural semantics is presented as a natural deduction system for deriving judgements of the form $t \Downarrow v$.

- ☞ We have $\mathbf{Val} = \mathbf{Num}$ and $\mathbf{Val} \subset \mathbf{Tm}$.

Evaluation rules

$$\frac{}{n \Downarrow n}\ [\textit{e-num}]$$

$$\frac{t_1 \Downarrow v_1 \qquad [x \mapsto v_1]t_2 \Downarrow v}{\mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} \Downarrow v}\ [\textit{e-let}]$$

$$\frac{t_1 \Downarrow n_1 \qquad t_2 \Downarrow n_2}{t_1 + t_2 \Downarrow n_1 \underline{+} n_2}\ [\textit{e-add}]$$

- ☞ There is no rule for variables: evaluation is undefined for programs containing unbound variables.
- ☞ Writing $[x \mapsto v_1]t_2$ we make essential use of the fact that $\mathbf{Val} \subset \mathbf{Tm}$.

Explicit substitutions

Although evaluation by means of beta-substitution is by far the most popular form of describing the semantics of languages like the one we are considering, beta-substitution is rather clumsy to implement directly.

To bridge the gap between theory and practice somewhat, we now look into an alternative way of defining the semantics of our language: one that is easier to map to an implementation in an interpreter.

The key idea is to make substitutions explicit by having them as values of a map-like data structure.

Environments

An environment denotes a (finite) mapping from variables to values:

$\eta \quad \in \quad \mathbf{Env} \cong \mathbf{Var} \to_{\text{fin}} \mathbf{Val} \qquad$ environments

$\eta \quad ::= \quad [\,] \mid \eta_1[x \mapsto v]$

☞ A finite mapping is much like an association list.

Domain and codomain of an environment

The domain of an environment is just the set of all variables that appear as keys in the mapping:

$$\begin{aligned}
&\mathit{dom} : \mathbf{Env} \to \mathcal{P}(\mathbf{Var})\\
&\mathit{dom}([\,]) = \{\,\}\\
&\mathit{dom}(\eta_1[x \mapsto v]) = \mathit{dom}(\eta_1) \cup \{\,x\,\}
\end{aligned}$$

Similarly, the codomain (or range) of an environment is defined by

$$\begin{aligned}
&\mathit{cod} : \mathbf{Env} \to \mathcal{P}(\mathbf{Val})\\
&\mathit{cod}([\,]) = \{\,\}\\
&\mathit{cod}(\eta_1[x \mapsto v]) = \mathit{cod}(\eta_1) \cup \{\,v\,\}
\end{aligned}$$

Looking up bindings in an environment

We write $\eta(x)$ for the value associated with the *rightmost* binding for x in η:

$$\begin{aligned}
&\cdot(\cdot) : \mathbf{Env} \to \mathbf{Var} \rightharpoonup \mathbf{Val}\\
&(\eta_1[x_0 \mapsto v])(x) = v \qquad \text{if } x = x_0\\
&(\eta_1[x_0 \mapsto v])(x) = \eta_1(x) \quad \text{if } x \neq x_0
\end{aligned}$$

☞ $\eta(\cdot)$ is a *partial* function over variables.

Natural semantics with environments

We now define a natural semantics for our language in terms of a natural deduction system for deriving judgements of the form

$\eta \vdash t \Downarrow v$

Read: "in environment η, the term t evaluates to the value v".

The idea is that η provides bindings for the free variables of t, i.e., that $\mathit{fv}(t) \subseteq \mathit{dom}(\eta)$.

Evaluation rules with environments

$$\frac{}{\eta \vdash n \Downarrow n}\ \text{[e-num]}$$

$$\frac{\eta(x) = v}{\eta \vdash x \Downarrow v}\ \text{[e-var]}$$

$$\frac{\eta \vdash t_1 \Downarrow v_1 \quad \eta[x \mapsto v_1] \vdash t_2 \Downarrow v}{\eta \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} \Downarrow v}\ \text{[e-let]}$$

$$\frac{\eta \vdash t_1 \Downarrow n_1 \quad \eta \vdash t_2 \Downarrow n_2}{\eta \vdash t_1 + t_2 \Downarrow n_1 \underline{+} n_2}\ \text{[e-add]}$$

- ☞ This time, we do have a rule for variables.
- ☞ In the rule for local definitions, substitution is made explicit as a new binding in the environment.

Evaluation: AG implementation


```
{type Num_ = Int    }
{type Var  = String}
data Tm | Num n :: {Num_} | Var x :: {Var}
        | Let x :: {Var} t1 :: Tm t2 :: Tm
        | Add t1 :: Tm t2 :: Tm
{type Val = Num_       }
{type Env = [(Var, Val)]}
attr Tm
  inh env ::  {Env}
  syn val ::  {Val}
sem Tm
  | Num lhs.val = @n
  | Var  lhs.val = case lookup @x @lhs.env of
                     Nothing -> error "unbound variable"
                     Just v  -> v
  | Let   t2.env  = (@x, @t1.val) : @lhs.env
  | Add  lhs.val = @t1.val + @t2.val
```


Type environments

Defining a static semantics for our langauge, we deal with variables in pretty much the same way as we did when defining a natural semantics with explicit substitutions.

First we define a language of types:

τ	$\in$	$\mathbf{Ty}$	types

$$\tau \ ::= \ \mathit{Nat}$$

Then, we introduce *type environments*, which provide an abstraction of environments, just like types provide an abstraction of values:

Γ	$\in$	$\mathbf{TyEnv} \cong \mathbf{Var} \to_{\mathsf{fin}} \mathbf{Ty}$	type environments

$$\Gamma \ ::= \ [\,] \mid \Gamma_1[x \mapsto \tau]$$

Notation for type environments

Similar to the notation used for environments, we write *dom*(Γ) and *cod*(Γ) for, respectively, the domain and codomain of a type environment.

Moreover, we write $\Gamma(x)$ to refer to the type associated with the rightmost binding for x in Γ.

Typing

Then, the typing relation can be given by a natural deduction system for deriving judgements of the form

$$\Gamma \vdash t : \tau$$

Read: “in type environment Γ, the term t can be assigned the type τ”.

The idea is that Γ provides types for the free variables of t, i.e., that $fv(t) \subseteq dom(\Gamma)$.

Typing rules

$$\frac{}{\Gamma \vdash n : \mathit{Nat}}\ [\textit{t-num}]$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ [\textit{t-var}]$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash t_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} : \tau}\ [\textit{t-let}]$$

$$\frac{\Gamma \vdash t_1 : \mathit{Nat} \quad \Gamma \vdash t_2 : \mathit{Nat}}{\Gamma \vdash t_1 + t_2 : \mathit{Nat}}\ [\textit{t-add}]$$

Typing: AG implementation


```
{data Ty     = Nat          }
{type TyEnv = [(Var, Ty)]}
attr Tm
  inh tyEnv :: {TyEnv}
  syn ty    :: {Ty}
sem Tm
  | Num lhs.ty    = Nat
  | Var lhs.ty    = case lookup @x @lhs.tyEnv of
                      Nothing → error "unbound variable"
                      Just τ   → τ
  | Let t₂.tyEnv = (@x, @t₁.ty) : @lhs.tyEnv
  | Add lhs.ty    = case (@t₁.ty, @t₂.ty) of
                      (Nat, Nat) → Nat
```


7.2 Mutuable state

Assignments

Pure functional languages do not allow the programmer to change the value of already computed values and, hence, do not provide assignments: variables retrieve their values when they are introduced and then remain *immutable*.

Most programming languages, however, do include a notion of assignment: they provide constructs that allow values bound to already introduced variables to be altered.

For example, in Java we can write:

```
int x = 2;
int y = 3;
int z = x + y;
x = z;          // redefinition of x
z = x + y;      // redefinition of z
```


Syntax

We now study the syntax and semantics of a small language featuring assignments.

n	$\in$	$\mathbf{Num}$	numerals
x	$\in$	$\mathbf{Var}$	variables
t	$\in$	$\mathbf{Tm}$	terms

$$t ::= n \mid x \mid t_1 + t_2 \mid x := t_1 \mid t_1; t_2$$

$x := t_1$ denotes the assigning the value of the term t_1 to the variable x.

$t_1; t_2$ denotes sequencing: first the term t_1 is evaluated, followed by the evaluation of t_2.

Semantics

Defining the semantics of assignment and sequencing, we have to decide on:

- how to deal with the side effect of an assignment;
- to what value an assignment evaluates (for instance, what is the result of $2 + (x := 3)$?); and
- to what value a sequence of terms evaluates.

With respect to the value of an assignment there seem to be at least two sensible approaches: **(1)** each assignment can evaluate to a fixed, trivial value (like $()$ in Haskell) or **(2)** an assignment $x := t_1$ evaluates to the value of t_1. (Here, we choose the latter.)

For the value of a sequence $t_1; t_2$ we simply pick the value of t_2, so that, for example, $x := 2; y := 3; x + y$ evaluates to 5.

Environments (first attempt)

As a first (failing) attempt to formally define the semantics of assignments and sequencing, we simply record the binding of values to variables in an environment and pass that environment downwards (i.e., as an inherited attribute) through the abstract-syntax tree—just like we did for local definitions.

v	$\in$	$\mathbf{Val} \cong \mathbf{Num}$	values
η	$\in$	$\mathbf{Env} \cong \mathbf{Var} \to_{\text{fin}} \mathbf{Val}$	environments

$$v ::= n$$
$$\eta ::= [\,] \mid \eta_1[x \mapsto v]$$

Natural semantics (first attempt)

$$\frac{}{\eta \vdash n \Downarrow n}\ \textit{[e-num]}$$

$$\frac{\eta(x) = v}{\eta \vdash x \Downarrow v}\ \textit{[e-var]}$$

$$\frac{\eta \vdash t_1 \Downarrow v}{\eta \vdash x := t_1 \Downarrow v}\ \textit{[e-assign]}$$

$$\frac{\eta \vdash t_2 \Downarrow v}{\eta \vdash t_1; t_2 \Downarrow v}\ \textit{[e-seq]}$$

$$\frac{\eta \vdash t_1 \Downarrow n_1 \quad \eta \vdash t_2 \Downarrow n_2}{\eta \vdash t_1 + t_2 \Downarrow n_1 \underline{+} n_2}\ \textit{[e-add]}$$

☞ But, of course, this will not work out: how do we store the side effect of an assignment $x := t_1$?

Chaining environments

Solution: we not only pass the environment downwards through the tree, we also pass it upwards (i.e., as a chained attribute).

The judgements of the natural semantics then take the form

$\langle \eta, t\rangle \Downarrow \langle \eta', v\rangle$	evaluation

That is, in an environment η a term t evaluates to an *updated environment* η' and a value v.

Natural semantics (chained environment)

$$\frac{}{\langle \eta, n\rangle \Downarrow \langle \eta, n\rangle}\ \textit{[e-num]}$$

$$\frac{\eta(x) = v}{\langle \eta, x\rangle \Downarrow \langle \eta, v\rangle}\ \textit{[e-var]}$$

$$\frac{\langle \eta, t_1\rangle \Downarrow \langle \eta'_1, v\rangle}{\langle \eta, x := t_1\rangle \Downarrow \langle \eta'_1[x \mapsto v], v\rangle}\ \textit{[e-assign]}$$

$$\frac{\langle \eta, t_1\rangle \Downarrow \langle \eta'', v_1\rangle \quad \langle \eta'', t_2\rangle \Downarrow \langle \eta', v\rangle}{\langle \eta, t_1; t_2\rangle \Downarrow \langle \eta', v\rangle}\ \textit{[e-seq]}$$

$$\frac{\langle \eta, t_1\rangle \Downarrow \langle \eta'', n_1\rangle \quad \langle \eta'', t_2\rangle \Downarrow \langle \eta', n_2\rangle}{\langle \eta, t_1 + t_2\rangle \Downarrow \langle \eta', n_1 \underline{+} n_2\rangle}\ \textit{[e-add]}$$

AG implementation


```
{type Num_ = Int    }
{type Var  = String}
data Tm | Num n :: {Num_} | Var x :: {Var}
        | Assign x :: {Var} t1 :: Tm | Seq t1 :: Tm t2 :: Tm
        | Add t1 :: Tm t2 :: Tm
{type Val = Num_        }
{type Env = [(Var, Val)]}
attr Tm inh env :: {Env}
        syn env :: {Env}
        syn val :: {Val }
sem Tm
  | Num    lhs.val = @n
  | Var    lhs.val = case lookup @x @lhs.env of
                       Nothing → error "unbound variable"
                       Just v  → v
  | Assign lhs.env = (@x, @t1.val) : @lhs.env
  | Add    lhs.val = @t1.val + @t2.val
```

☞ The copy rule naturally takes care of all "trivial" propagation.

Type system

Type checking a language with assignments it seems natural to follow the structure of the operational semantics, i.e., to thread a type environment through the syntax tree.

τ	$\in$	$\mathbf{Ty}$	types
Γ	$\in$	$\mathbf{TyEnv} \cong \mathbf{Var} \to_{\text{fin}} \mathbf{Ty}$	type environments

$$\begin{array}{lcl} \tau & ::= & \mathit{Nat} \\ \Gamma & ::= & [\,] \mid \Gamma_1[x \mapsto \tau] \end{array}$$

The judgements of the typing relation then take the form

$\Gamma \vdash t : \tau \rightsquigarrow \Gamma'$	typing

Here, Γ' is an *updated type environment*.

Typing rules

$$\frac{}{\Gamma \vdash n : \mathit{Nat} \rightsquigarrow \Gamma}\ [\textit{t-num}]$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash t : \tau \rightsquigarrow \Gamma}\ [\textit{t-var}]$$

$$\frac{\Gamma \vdash t_1 : \tau \rightsquigarrow \Gamma_1'}{\Gamma \vdash x := t_1 : \tau \rightsquigarrow \Gamma_1'[x \mapsto \tau]}\ [\textit{t-assign}]$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \rightsquigarrow \Gamma'' \quad \Gamma'' \vdash t_2 : \tau \rightsquigarrow \Gamma'}{\Gamma \vdash t_1; t_2 : \tau \rightsquigarrow \Gamma'}\ [\textit{t-seq}]$$

$$\frac{\Gamma \vdash t_1 : \mathit{Nat} \rightsquigarrow \Gamma'' \quad \Gamma'' \vdash t_2 : \mathit{Nat} \rightsquigarrow \Gamma'}{\Gamma \vdash t_1 + t_2 : \mathit{Nat} \rightsquigarrow \Gamma'}\ [\textit{t-add}]$$

Problem with typing

Still, there is a problem with defining the typing relation in this way.

To show this, let us first add booleans to our language:

$$\begin{array}{lcl} t & ::= & \cdots \mid \mathtt{false} \mid \mathtt{true} \mid \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ \mathbf{fi} \\ v & ::= & \cdots \mid \mathtt{false} \mid \mathtt{true} \\ \tau & ::= & \cdots \mid \mathit{Bool} \end{array}$$

Evaluating with booleans

$$\frac{}{\langle \eta; \mathtt{false} \rangle \Downarrow \langle \eta; \mathtt{false} \rangle}\ [\textit{e-false}]$$

$$\frac{}{\langle \eta; \mathtt{true} \rangle \Downarrow \langle \eta; \mathtt{true} \rangle}\ [\textit{e-true}]$$

$$\frac{\langle \eta, t_1 \rangle \Downarrow \langle \eta'', \mathtt{true} \rangle \quad \langle \eta'', t_2 \rangle \Downarrow \langle \eta', v \rangle}{\langle \eta, \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ \mathbf{fi} \rangle \Downarrow \langle \eta', v \rangle}\ [\textit{e-if-true}]$$

$$\frac{\langle \eta, t_1 \rangle \Downarrow \langle \eta'', \mathtt{false} \rangle \quad \langle \eta'', t_3 \rangle \Downarrow \langle \eta', v \rangle}{\langle \eta, \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ \mathbf{fi} \rangle \Downarrow \langle \eta', v \rangle}\ [\textit{e-if-false}]$$

Typing booleans

$$\frac{}{\Gamma \vdash \texttt{false} : \mathit{Bool} \rightsquigarrow \Gamma}\ [\textit{t-false}]$$

$$\frac{}{\Gamma \vdash \texttt{true} : \mathit{Bool} \rightsquigarrow \Gamma}\ [\textit{t-true}]$$

$$\frac{\Gamma \vdash t_1 : \mathit{Bool} \rightsquigarrow \Gamma'' \quad \Gamma'' \vdash t_2 : \tau \rightsquigarrow \Gamma' \quad \Gamma'' \vdash t_3 : \tau \rightsquigarrow \Gamma'}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \textbf{ fi} : \tau \rightsquigarrow \Gamma'}\ [\textit{t-if}]$$

☞ Requiring that both branches of a conditional result in the same updated environment seems quite harsh: but what's the alternative?

A closer look at the problem

Assume that t_1 is a term of type Bool. Under the proposed static semantics, the following program is ill-typed:

$$\textbf{if } t_1 \textbf{ then } x := 2 \textbf{ else } x := \texttt{false} \textbf{ fi};$$
$$x + 3$$

That seems reasonable. But what about the following (ill-typed) program?

$$\textbf{if } t_1 \textbf{ then } x := 2 \textbf{ else } z := 5; x := z + 7 \textbf{ fi};$$
$$x + 3$$

☞ Here, z is only used locally within the else-branch, but, of course, still shows up in the updated type environment.

A closer look at the problem (cont'd)

Things are even worse. Consider:

$$\textbf{if } t_1 \textbf{ then } x := 2 \textbf{ else } x := 5; x := 7 \textbf{ fi};$$
$$x + 3$$

☞ Here, all assignments are at least intuitively well-typed, but still, the branches of the conditional result in different updated type environments (one containing one additional binding for x and one containing two) and so the program is considered ill-typed by our static semantics.

Controlling scope

As a solution, we will make the notion of scope explicit in our language and no longer allow type environments to be updated: while its value can be changed, the type of a variable remains unchanged after the variable has been introduced.

To control the scope of variables, we introduce local definitions in our language:

$$t \quad ::= \quad \cdots \mid \textbf{let } x = t_1 \textbf{ in } t_2 \textbf{ ni}$$

☞ In most imperative languages, scope is not made as explicit as is here, but still plays an important rôle in the typing of programs.

Local definitions: examples

The three previous example programs can now be written as:

$\textbf{let } x = 0$ *-- x must be "initialised" first*
$\textbf{in } \textbf{ if } t_1 \textbf{ then } x := 2 \textbf{ else } x := \texttt{false} \textbf{ fi}; x + 3$
$\textbf{ni}$

$\textbf{let } x = 0$ *-- x must be "initialised" first*
$\textbf{in } \textbf{ if } t_1 \textbf{ then } x := 2 \textbf{ else let } z = 5 \textbf{ in } x := z + z \textbf{ ni fi}; x + 3$
$\textbf{ni}$

$\textbf{let } x = 0$ *-- x must be "initialised" first*
$\textbf{in } \textbf{ if } t_1 \textbf{ then } x := 2 \textbf{ else } x := 5; x := 7 \textbf{ fi}; x + 3$
$\textbf{ni}$

☞ Our goal is to only have the first program considered ill-typed.

Evaluation with local definitions

Adding a rule for local definitions we have to be careful to deal with scoping issues correctly:

$$\frac{\langle \eta, t_1 \rangle \Downarrow \langle \eta'', v_1 \rangle \quad \langle \eta''[x \mapsto v_1], t_2 \rangle \Downarrow \langle \eta', v \rangle}{\langle \eta, \textbf{let } x = t_1 \textbf{ in } t_2 \textbf{ ni} \rangle \Downarrow \langle \eta' \backslash x, v \rangle} \text{[\textit{e-let}]}$$

Here, $\eta' \backslash x$ denotes the environment that is obtained by removing the *rightmost* binding for x from η'.

Removing this binding is necessary, because outside the local definition, occurrences of the variable x refer to other bindings.

Similarly, we have to adjust the rule for assignments, so that the environment is really updated, rather than just extended:

$$\frac{\langle \eta, t_1 \rangle \Downarrow \langle \eta'_1, v \rangle}{\langle \eta, x := t_1 \rangle \Downarrow \langle (\eta'_1 \backslash x)[x \mapsto v], v \rangle} \text{[\textit{e-assign}]}$$

Adjusting the type system

Now we redefine the type system so that type environments are no longer updated.

Hence, typing judgements again read

$\Gamma \vdash t : \tau$ typing

Typing rules

The rules for constants and variables are, as always, straightforward:

$$\frac{}{\Gamma \vdash n : \mathit{Nat}} \text{[\textit{t-num}]}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \mathit{Bool}} \text{[\textit{t-false}]}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \mathit{Bool}} \text{[\textit{t-true}]}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash t : \tau} \text{[\textit{t-var}]}$$

Typing rules (cont'd)

The rule for local definitions is as we have seen it before:

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma[x \mapsto \tau_1] \vdash t_2 : \tau}{\Gamma \vdash \mathbf{let}\ x = t_1\ \mathbf{in}\ t_2\ \mathbf{ni} : \tau}\ [\textit{t-let}]$$

When typing assignments, we have to make sure that the type of the new value is consistent with the type of the variable:

$$\frac{\Gamma(x) = \tau \quad \Gamma \vdash t_1 : \tau}{\Gamma \vdash x := t_1 : \tau}\ [\textit{t-assign}]$$

Typing sequences is simple:

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1; t_2 : \tau}\ [\textit{t-seq}]$$

Typing rules (cont'd)

Finally, we have the following rules for conditionals and additions:

$$\frac{\Gamma \vdash t_1 : \mathit{Bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3\ \mathbf{fi} : \tau}\ [\textit{t-if}]$$

$$\frac{\Gamma \vdash t_1 : \mathit{Nat} \quad \Gamma \vdash t_2 : \mathit{Nat}}{\Gamma \vdash t_1 + t_2 : \mathit{Nat}}\ [\textit{t-add}]$$

Variation: local declarations

Instead of always requiring an initial value, we could, additionally, introduce a construct that does not define a variable locally but instead only declares its type:

$$t \quad ::= \quad \cdots \mid \mathbf{let}\ x : \tau\ \mathbf{in}\ t_1\ \mathbf{ni}$$

For example, assuming that t_1 has type Bool:

$$\begin{array}{l}\mathbf{let}\ x : \mathit{Nat} \\ \mathbf{in}\ \ \mathbf{if}\ t_1\ \mathbf{then}\ x := 2\ \mathbf{else}\ x := 5\ \mathbf{fi}; x + 3\ \mathbf{ni}\end{array}$$

☞ Such a construct requires type expressions to be part of the language's concrete syntax.

Evaluating and typing local declarations

When evaluating a program, types play no rôle:

$$\frac{\langle \eta, t_1 \rangle \Downarrow \langle \eta', v \rangle}{\langle \eta, \mathbf{let}\ x : \tau\ \mathbf{in}\ t_1\ \mathbf{ni} \rangle \Downarrow \langle \eta', v \rangle}\ [\textit{e-let-ty}]$$

Typing local declarations is straightforward:

$$\frac{\Gamma[x \mapsto \tau_1] \vdash t_1 : \tau}{\Gamma \vdash \mathbf{let}\ x : \tau_1\ \mathbf{in}\ t_1\ \mathbf{ni} : \tau}\ [\textit{t-let-ty}]$$

Uninitialised variables

With local declarations, we have introduced yet another way for evaluation to fail, i.e., by means of accessing an uninitialised variable.

For example:

$$\mathbf{let}\ x : \mathit{Nat}\ \mathbf{in}\ x + x\ \mathbf{ni}$$

This program is well-typed, but will fail at run-time.

Exercise: implement, as an attribute grammar, a static check that prevents uninitialised variables from being accessed. (Hint: most interesting is how you deal with conditionals. Furthermore, you have to take care of shadowing properly.)

Variation: local declarations without types

Yet another variation: we can add local declarations that do *not* even mention the type of the variable.

$$t \quad ::= \quad \cdots \mid \mathbf{let}\ x\ \mathbf{in}\ t_1\ \mathbf{ni}$$

$$\frac{\langle \eta, t_1 \rangle \Downarrow \langle \eta', v \rangle}{\langle \eta, \mathbf{let}\ x\ \mathbf{in}\ t_1\ \mathbf{ni} \rangle \Downarrow \langle \eta', v \rangle}\ [\textit{e-let}']$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash t_1 : \tau}{\Gamma \vdash \mathbf{let}\ x\ \mathbf{in}\ t_1\ \mathbf{ni} : \tau}\ [\textit{t-let}']$$

☞ Curiously, when typing such a definition, we have to “guess” an appropriate type τ_1 for x. Effectively, we then have to *infer* types, rather than just check types.

8. Simple functions