Typing of simPL

YSC3208: Programming Language Design & Implementation

Răzvan Voicu

Week 4, Jan 30 – Feb 3, 2017

Substitution

Goal: For function application, replace all free occurrences of the formal parameters in the function body by the actual arguments.

```
(fun {int -> int} x -> x * x end 4)
```

Replace every free occurrence of `x` in `x * x` by the actual parameter 4, resulting in

```
4 * 4
```

Substitution

Define the substitution relation

$$\cdot\,[\cdot\!\leftarrow\!\cdot] \rightsquigarrow \cdot : \text{simPL} \times V \times \text{simPL} \times \text{simPL}$$

such that $\texttt{x * x}[\texttt{x} \leftarrow \texttt{4}] \rightsquigarrow \texttt{4 * 4}$ holds.

Definition of Substitution

$$\frac{}{v\,[v \leftarrow E_1] \rightsquigarrow E_1} \quad \text{for any variable } v$$

$$\frac{}{x\,[v \leftarrow E_1] \rightsquigarrow x} \quad \text{for any variable } x \neq v$$

Definition of Substitution (cont'd)

$$\frac{E_1\,[v\leftarrow E] \rightsquigarrow E_1' \qquad E_2\,[v\leftarrow E] \rightsquigarrow E_2'}{(E_1\ E_2)\,[v\leftarrow E] \rightsquigarrow (E_1'\ E_2')}$$

Definition of Substitution (cont'd)

$$\frac{}{\texttt{fun}\ \{\cdot\}\ v\texttt{->}E\ \texttt{end}\,[v{\leftarrow}E_1] \rightsquigarrow \texttt{fun}\ \{\cdot\}\ v\ \texttt{->}\ E\ \texttt{end}}$$

Note that the above rule help avoids name clashes.

$$\frac{E\,[v{\leftarrow}E_1] \rightsquigarrow E' \qquad x \neq v \qquad E_1 \bowtie X_1 \qquad x \notin X_1}{\texttt{fun}\ \{\cdot\}\ x\texttt{->}E\ \texttt{end}\,[v{\leftarrow}E_1] \rightsquigarrow \texttt{fun}\ \{\cdot\}\ x\ \texttt{->}\ E'\ \texttt{end}}$$

Definition of Substitution (cont'd)

$$\frac{E_1 \bowtie X_1 \qquad x \in X_1 \qquad E \bowtie X \qquad E\,[x \leftarrow z] \leadsto E' \qquad E'\,[v \leftarrow E_1] \leadsto E'' \qquad x \neq v}{\texttt{fun}\ \{\cdot\}\ x\texttt{->}E\ \texttt{end}\,[v \leftarrow E_1] \leadsto \texttt{fun}\ \{\cdot\}\ z\ \texttt{->}\ E''\ \texttt{end}}$$

where we choose z such that $z \notin X_1 \cup X$. The renaming of x to a fresh z is to avoid a free x variable in E_1 being accidentally captured as a bound variable.

Examples

- Avoiding name clash:

  `fun {int -> int} factor -> factor * 4 * y end`

  `[factor`$\leftarrow$ `x + 1]` $\rightsquigarrow$

  `fun {int -> int} factor -> factor * 4 * y end`

- No name clash below:

  `fun {int -> int} factor -> factor * 4 * y end`

  `[y`$\leftarrow$ `x + 1]` $\rightsquigarrow$

  `fun {int -> int} factor -> factor * 4 * (x + 1) end`

Examples

- Avoiding name capture with a fresh naming:

  `fun {int -> int} factor -> factor * 4 * y end`

  $[$`y`$\leftarrow$ `factor + 1`$] \rightsquigarrow$

  ```
  fun {int -> int} newfactor ->
     newfactor * 4 * (factor + 1) end
  end
  ```

Substitution in OCaml

Substitution need to avoid name clashes and also perform renaming to avoid name capture.

```
let apply_subs
  (fnc:id list->id list-> id list * (id * id) list)
  (rename_op:(id*id) list->sPL_expr->sPL_expr)
  (ss:(id*sPL_expr)list)
  (e:sPL_expr) : sPL_expr =
```

Substitution in OCaml (cont.)

Some OCaml code on substitution

```
let rec aux ss e =
  match e with
    | BoolConst _ | IntConst _ -> e
    | Var i -> subs_var i ss
    | UnaryPrimApp (op,arg)
       -> UnaryPrimApp (op,aux ss arg)
    | BinaryPrimApp (op,arg1,arg2)
      -> BinaryPrimApp (op,aux ss arg1,aux ss arg2)
    | Cond (e1,e2,e3)
      -> Cond (aux ss e1,aux ss e2,aux ss e3)
    ...
```

Contraction of Function Application

$$\frac{E\,[x{\leftarrow}v] \rightsquigarrow E'}{(\texttt{fun}\ \{\,\cdot\,\}\ x\ \texttt{->}\ E\ \texttt{end}\quad v)\ >_{\text{simPL}}\ E'}\ [\text{CallFun}]$$

Contraction of Recursive Function Application

$$\frac{E\,[f \leftarrow \texttt{recfun}\ \{\cdot\}\ f\ x\ \texttt{->}\ E\ \texttt{end}] \rightsquigarrow E' \qquad E'\,[x \leftarrow v] \rightsquigarrow E''}{(\texttt{recfun}\ f\ x\ \texttt{->}\ E\ \texttt{end}\quad v) >_{\mathrm{simPL}} E''}[\mathrm{RF}]$$

One-Step Evaluation

$$\frac{E >_{\text{simPL}} E'}{E \mapsto_{\text{simPL}} E'}[\text{Contraction}]$$

$$\frac{E \mapsto_{\text{simPL}} E'}{p_1[E] \mapsto_{\text{simPL}} p_1[E']}[\text{OpArg}_1]$$

One-Step Evaluation (cont'd)

$$\frac{E_1 \mapsto_{\text{simPL}} E_1'}{p_2[E_1, E_2] \mapsto_{\text{simPL}} p_2[E_1', E_2]} [\text{OpArg}_2]$$

$$\frac{E_2 \mapsto_{\text{simPL}} E_2'}{p_2[v_1, E_2] \mapsto_{\text{simPL}} p_2[v_1, E_2']} [\text{OpArg}_3]$$

One-Step Evaluation (cont'd)

$$\frac{E \mapsto_{\mathrm{simPL}} E'}{\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end} \mapsto_{\mathrm{simPL}} \texttt{if } E' \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end}}$$

One-Step Evaluation (cont'd)

$$\frac{E \mapsto_{\text{simPL}} E'}{(E\ E_1 \ldots E_n) \mapsto_{\text{simPL}} (E'\ E_1 \ldots E_n)}\,[\text{AppFun}]$$

One-Step Evaluation (cont'd)

$$\frac{E_i \mapsto_{\mathrm{simPL}} E_i'}{(v\ \ v_1 \ldots v_{i-1}\ \ E_i \ldots E_n) \mapsto_{\mathrm{simPL}} (v\ \ v_1 \ldots v_{i-1}\ \ E_i' \ldots E_n)}\,[\mathrm{AppArg}]$$

Evaluation of simPL Programs

As for ePL, evaluation of simPL is defined by the evalution relation $\mapsto^*_{\mathrm{simPL}}$, the reflexive transitive closure of $\mapsto_{\mathrm{simPL}}$.

1. Dynamic Semantics of simPL (cont'd)

2. Typing of simPL
   - Type Environments
   - Typing Relation for simPL
   - Type Safety of simPL

Example

Is `x + 3` well-typed?

Type Environments

We need a type environment to tell us the types of each variable.

A *Type environment*, denoted by Γ, keeps track of the type of identifiers appearing in the expression.

$\Gamma(x)$ returns the type that is known by environment Γ for the identifier x.

Environment Extension

If $\Gamma[x \leftarrow t]\Gamma'$, then Γ' behaves like Γ, except that the type of x is t.

Example

Let $\Gamma = \emptyset$.

$\emptyset[\texttt{AboutPi} \leftarrow \texttt{int}]\Gamma'$

$\Gamma'(\texttt{AboutPi}) = \texttt{int}$

$\Gamma'[\texttt{Square} \leftarrow \texttt{int->int}]\Gamma''$

$dom(\Gamma'') = \{\texttt{AboutPi}, \texttt{Square}\}$

Type Environment in OCaml

```
module Environ =
struct
  type 'b et = (id * 'b) list

  let empty_env : 'b et = []

  let get_val (env:'b et) (v:id) : 'b option =
    try
      Some (snd (List.find (fun (i,_) -> i=v) env))
    with _ -> None
    :
```

Type Environment in OCaml (cont.)

```
  let add_env (env:'b et) (v:id) (e:'b)
    : 'b et = (v,e)::env

  let extend_env (env:'b et) (ls:(id*'b) list)
    : 'b et = ls@env
end;;
```

Instantiating the type for environment:

type env_type = sPL_type Environ.et

Typing Relation

The set of well-typed expressions is defined by the ternary *typing relation*, written $\Gamma \vdash E : t$, where Γ is a type environment such that $E \bowtie X$ and $X \subseteq dom(\Gamma)$.

"The expression E has type t, under the assumption that its free identifiers have the types given by Γ."

Examples

- $\Gamma' \vdash \texttt{AboutPi} * 2 : \texttt{int}$
- $\Gamma'' \vdash \texttt{fun}\{\texttt{int} \texttt{->} \texttt{int}\}\ \texttt{x->AboutPi * (Square 2) end}$
  $: \texttt{int} \texttt{->} \texttt{int}$

Examples

Let $\Gamma = \emptyset$.

$\emptyset[\texttt{AboutPi} \leftarrow \texttt{int}]\Gamma'$

$\Gamma' \vdash \texttt{fun \{int->int\} x->AboutPi * (Square 2) end} : \texttt{int->int}$

does not hold, because `Square` occurs free in the expression, but the type environment Γ' to the left of the $\vdash$ symbol is not defined for `Square`.

Definition of Typing Relation

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \textbf{[VarT]}$$

If $\Gamma(x)$ is not defined, then this rule is not applicable. In this case, we say that there is no type for x derivable from the assumptions Γ.

Definition of Typing Relation - Constants

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \textbf{[NumT]} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ \textbf{[TrueT]}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ \textbf{[FalseT]}$$

Definition of Typing Relation : Primitives

For each primitive operation p that takes n arguments of types $t_1, ..., t_n$ and returns a value of type t, we have exactly one rule of the following form.

$$\frac{\Gamma \vdash E_1 : t_1 \qquad \cdots \qquad \Gamma \vdash E_n : t_n}{\Gamma \vdash p[E_1, \ldots, E_n] : t} \quad \textbf{[PrimT]}$$

Definition of Typing Relation (cont'd)

p	t_1	t_2	t
`+`	`int`	`int`	`int`
`-`	`int`	`int`	`int`
`*`	`int`	`int`	`int`
`/`	`int`	`int`	`int`
`~`	`int`		`int`
`\`	`bool`		`bool`
`&`	`bool`	`bool`	`bool`
`|`	`bool`	`bool`	`bool`
`=`	`int`	`int`	`bool`
`<`	`int`	`int`	`bool`
`>`	`int`	`int`	`bool`

Definition of Typing Relation : Conditional

$$\frac{\Gamma \vdash E : \texttt{bool} \qquad \Gamma \vdash E_1 : t \qquad \Gamma \vdash E_2 : t}{\Gamma \vdash \texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2 \texttt{ end} : t}$$

Definition of Typing Relation : Function

$$\frac{\Gamma_1[x_1 \leftarrow t_1]\Gamma_2 \ \cdots \Gamma_n[x_n \leftarrow t_n]\Gamma_{n+1} \qquad \Gamma_{n+1} \vdash E : t}{\Gamma_1 \vdash \texttt{fun}\ \{t_1\texttt{->}\cdots\texttt{->}t_n\texttt{->}t\}\ x_1\ \ldots x_n\texttt{->}E\ \texttt{end} : t_1\texttt{->}\cdots\texttt{->}t_n\texttt{->}t}$$

Definition of Typing Relation : Recursive Function

$$\frac{\Gamma[f \leftarrow t_1\texttt{->}\cdots\texttt{->}t_n\texttt{->}t]\Gamma_1 \quad \Gamma_1[x_1 \leftarrow t_1]\Gamma_2 \ \cdots \Gamma_n[x_n \leftarrow t_n]\Gamma_{n+1} \quad \Gamma_{n+1} \vdash E : t}{\Gamma \vdash \texttt{recfun}\ f\ \{t_1\texttt{->}\cdots\texttt{->}t_n\texttt{->}t\}\ x_1 \ldots x_n \texttt{->} E\ \texttt{end} : t_1\texttt{->}\cdots t_n\texttt{->}t}$$

Definition of Typing Relation : General Application

$$\frac{\Gamma \vdash E : t_1\texttt{->} \cdots \texttt{->}t_n\texttt{->}t \qquad \Gamma \vdash E_1 : t_1 \quad \cdots \quad \Gamma \vdash E_n : t_n}{\Gamma \vdash (E\ E_1 \cdots E_n) : t}$$

Definition of Typing Relation : Let Construct

It will be good to have type checking (or inference) done for syntactic abbreviations too, as this can give better error messages.

$$\frac{\Gamma \vdash E_1 : t_1 \qquad \Gamma[x \leftarrow t_1] \vdash E : t}{\Gamma \vdash \texttt{let}\ \{t_1\}\ x\ =\ E_1\ \texttt{in}\ \{t\}\ E\ \texttt{end} : t}$$

Well-Typedness

An expression E is well-typed, if there is a type t such that $E : t$.

Example Proof

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash 2 : \mathtt{int}} \qquad \dfrac{}{\emptyset \vdash 3 : \mathtt{int}}}{\emptyset \vdash \mathtt{2*3} : \mathtt{int}} \qquad \dfrac{}{\emptyset \vdash 7 : \mathtt{int}}}{\emptyset \vdash \mathtt{2*3>7} : \mathtt{bool}}$$

Example Proof

$$
\dfrac{
\dfrac{
\emptyset[\texttt{x} \leftarrow \texttt{int}]\Gamma \qquad
\dfrac{
\dfrac{}{\Gamma \vdash \texttt{x} : \texttt{int}} \qquad \dfrac{}{\Gamma \vdash \texttt{1} : \texttt{int}}
}{\Gamma \vdash \texttt{x+1} : \texttt{int}}
}{\emptyset \vdash \texttt{fun \{int->int\} x->x+1 end} : \texttt{int->int}}
\qquad
\dfrac{}{\emptyset \vdash \texttt{2} : \texttt{int}}
}{\emptyset \vdash \texttt{(fun \{int->int\} x->x+1 end 2)} : \texttt{int}}
$$

Unique Type

Lemma

For every expression E and every type assignment Γ, there exists at most one type t such that $\Gamma \vdash E : t$.

More Properties of Typing Relation

Lemma

Typing is not affected by "junk" in the type assignment. If $\Gamma \vdash E : t$, and $\Gamma \subset \Gamma'$, then $\Gamma' \vdash E : t$.

Lemma

Substituting an identifier by an expression of the same type does not affect typing. If $\Gamma[x \leftarrow t']\Gamma'$, $\Gamma' \vdash E : t$, and $\Gamma \vdash E' : t'$, then $\Gamma \vdash E'' : t$, where $E[x \leftarrow E']E''$.

Type Safety

Type safety is a property of a given language with a given static and dynamic semantics. It says that if a program of the language is well-typed, certain problems are guaranteed not to occur at runtime.

What do we consider as "problems"?

Components of Type Safety

Progress. Well-typed expressions are values or can be further evaluated.

Preservation. Well-typed expressions do not change their type during evaluation.

Definition of Type Safety

A programming language with a given typing relation $\cdots \vdash \cdots : \cdots$ and one-step evaluation $\mapsto$ is called type-safe, if the following two conditions hold:

1. **Preservation.** If E is a well-typed program with respect to $\cdots \vdash \cdots : \cdots$ and $E \mapsto E'$, then E' is also a well-typed program with respect to $\vdash$.
2. **Progress.** If E is a well-typed program, then either E is a value or there exists a program E' such that $E \mapsto E'$.

Preservation in simPL

If for a simPL expression E and some type t holds $E : t$ and if $E \mapsto_{\mathrm{simPL}} E'$, then $E' : t$.

Progress in simPL

Let simPL' be simPL without division.
If for a simPL' expression E holds $E : t$ for some type t, then either E is a value, or there exists an expression E' such that $E \mapsto_{\mathrm{simPL'}} E'$.

Divide by zero may cause program to get stuck, but this is due to violation of *safety precondition* of division rather than due to type error problem. Type system is unable to handle errors due to incorrect program logic.

Is perfect typing possible?

The type safety of simPL' ensures that evaluation of a well-typed simPL' expression does not get stuck due to a wrong type.
Can we say the reverse by claiming that any expression for which the dynamic semantics produces a value is well-typed?

Stepping back

Summary so far

- Typing allows us to focus on well-typed programs
- Well-typed programs "behave well" (progress, preservation)

Outlook

We will focus on well-typed programs and develop a semantics that eliminates many of the efficiency and engineering issues encountered with dynamic semantics.