Compiler

Static typing

Jyun-Ao Lin

iFIRST & CSIE, NTUT
jalin@ntut.edu.tw

Credits

A large part of this course is based on the Compilation Course of J.-C. Filliâtre at ENS Ulm.

Typing[1]

+

[1] source: `https://zh.wiktionary.org/`

Typing[2]

/

[2]source: `https://zh.wiktionary.org/`

Type checking

If we write

```
"5" + 37
```

do we get

- a compile-time error? (OCaml, Rust, Go)
- a runtime error? (Python, Julia)
- the integer 42? (Visual Basic, PHP)
- the string "537"? (Java, Scala, Kotlin)
- a pointer? (C, C++)
- something else?

and what about

```
37 / "5"
```

??????

Typing

If we now add two arbitrary expressions

```
e1 + e2
```

how can we decide whether this is legal and which operation to perform?

The answer is typing, a program analysis that binds types to each sub-expression, to rule out inconsistent programs

When?

Some languages are dynamically typed: types are bound to values and are used at runtime

examples: Lisp, PHP, Python, Julia

Other languages are statically typed: types are bound to expressions and are used at compile time

examples: C, C++, Java, OCaml, Rust, Go

Remark

A language may use both static and dynamic typing

Overview of the course

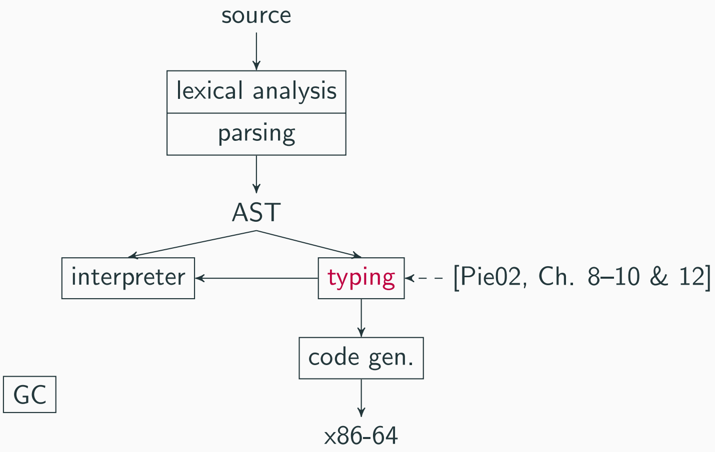

Table of contents

Static typing

Slogan (Milner, 1978)

well-typed programs do not go wrong

Goal of typing

- type checking must be decidable
- type checking must reject programs whose evaluation would fail; this is type safety
- type checking must not reject too many non-absurd programs; the type system must be expressive

Several solutions

1. any sub-expression is annotated with a type

```
fun (x: int) -> let (y: int) = (+ :) (((x: int),(1: int)): int * int )
```

   type checking is easy but this is unmanageable for the programmer

2. only annotate variable declarations (C, C++, Java, etc.)

```
fun (x : int) -> let (y : int) = +(x,1) in y
```

3. only annotate function parameters (C++ 11, Java 10)

```
fun (x : int) -> let y = +(x,1) in y
```

4. no annotation at all ⇒ type inference (OCaml, Haskell, etc.)

```
fun x -> x + 1
```

Expected properties

A type checking algorithm must have properties of

- correctness: if the algorithm answers “yes” then the program is effectively well-typed
- completeness: if the program is well-typed, then the algorithm must answer “yes”

and possibly of

- principality: the type calculated for an expression is the most general possible

Typing for mini-ML

Consider mini-ML typing

1. monomorphic typing
2. polymorphic typing, type inference

mini-ML

Recall the abstract syntax of mini-ML

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
 & | & c & \text{constant } (1, 2, \ldots, \texttt{true}, \ldots) \\
 & | & op & \text{primitive operator } (+, \times, \texttt{fst}, \ldots) \\
 & | & \texttt{fun } x \rightarrow e & \text{function} \\
 & | & e\ e & \text{application} \\
 & | & (e, e) & \text{pair} \\
 & | & \texttt{let } x = e \texttt{ in } e & \text{local let}
\end{array}
$$

Monomorphic typing of mini-ML

We introduce a simple typing, with the following abstract syntax

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \dots & \text{basic types} \\
 & \mid & \tau \rightarrow \tau & \text{type of a function} \\
 & \mid & \tau \times \tau & \text{type of a pair}
\end{array}
$$

Typing judgment

The type of a variable is given by a typing environment Γ
(a function from variables to types)

The typing judgment is written as

$$\Gamma \vdash e : \tau$$

and reads "in typing environment Γ, expression e has type τ"
The environment Γ associates a type $\Gamma(x)$ for each free variable x in e

We use inference rules to define $\Gamma \vdash e : \tau$

Rules of typing

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \text{ ETC.} \qquad \frac{}{\Gamma \vdash + : \texttt{int} \times \texttt{int} \rightarrow \texttt{int}} \text{ ETC.}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

$\Gamma + x : \tau$ is the environment Γ' defined by $\Gamma'(y) = \begin{cases} \tau & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$

Example

$$
\dfrac{
\dfrac{
\dfrac{\vdots \qquad \dfrac{\vdots}{x : \texttt{int} \vdash (x, 1) : \texttt{int} \times \texttt{int}}}{x : \texttt{int} \vdash +(x, 1) : \texttt{int}}
}{\emptyset \vdash \texttt{fun}\ x \rightarrow +(x, 1) : \texttt{int} \rightarrow \texttt{int}}
\qquad
\dfrac{\dfrac{}{\cdots \vdash f : \texttt{int} \rightarrow \texttt{int}} \qquad \dfrac{}{\cdots \vdash 2 : \texttt{int}}}{f : \texttt{int} \rightarrow \texttt{int} \vdash f\ 2 : \texttt{int}}
}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ f \rightarrow +(x, 1)\ \texttt{in}\ f\ 2 : \texttt{int}}
$$

Expressions without a type

On the other hand, we cannot type the program `1 2`.

$$\frac{\Gamma \vdash 1 : \tau' \to \tau \qquad \Gamma \vdash 2 : \tau'}{\Gamma \vdash 1\ 2 : \tau}$$

nor the program $\texttt{fun}\ x \to x\ x$

$$\frac{\dfrac{\Gamma + x : \tau_1 \vdash x : \tau_3 \to \tau_2 \qquad \Gamma + x : \tau_1 \vdash x : \tau_3}{\Gamma + x : \tau_1 \vdash x\ x : \tau_2}}{\Gamma \vdash \texttt{fun}\ x \to x\ x : \tau_1 \to \tau_2}$$

since $\tau_1 = \tau_1 \to \tau_2$ has no solution (the types are finite by definition)

Many possible types

We can show

$$\emptyset \vdash \texttt{fun}\ x \to x : \texttt{int} \to \texttt{int}$$

but also

$$\emptyset \vdash \texttt{fun}\ x \to x : \texttt{bool} \to \texttt{bool}$$

Be careful: this is not polymorphism

for a given occurrence of $\texttt{fun}\ x \to x$ it's necessary choose a type

Many possible types

Thus, the term $\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$ is not typeable,

because there is no type τ such as

$$f : \tau \to \tau \vdash (f\ 1, f\ \texttt{true}) : \tau_1 \times \tau_2.$$

On the other hand,

$$((\texttt{fun}\ x \to x)\ (\texttt{fun}\ x \to x))\ 42$$

is typable (exercise!)

Primitives

In particular, we cannot give a satisfying type to a primitive like *fst*; you would have to choose between an infinite number of possible types:

$$\texttt{int} \times \texttt{int} \to \texttt{int}$$
$$\texttt{int} \times \texttt{bool} \to \texttt{int}$$
$$\texttt{bool} \times \texttt{int} \to \texttt{bool}$$
$$(\texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int}$$

etc.

But on the other hand we can give a rule of typing for the application of *fst*:

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathit{fst}\ e : \tau_1}$$

Primitives

The same goes for primitives *opif* and *opfix*. We cannot give a satisfactory type to *opfix*, but we can give a rule of typing for its application

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \mathit{opfix}\ e : \tau}$$

And if we want to limit ourselves to *functions*, we can modify it like this

$$\frac{\Gamma \vdash e : (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)}{\Gamma \vdash \mathit{opfix}\ e : \tau_1 \to \tau_2}$$

Recursive function

If we add the construct `let rec` in the language, we could have

$$\frac{\Gamma + x : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma + x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

And again, for functions only

$$\frac{\Gamma + (f : \tau \to \tau_1) + (x : \tau) \vdash e_1 : \tau_1 \qquad \Gamma + (f : \tau \to \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Difference between rules of typing and algorithm of typing

When we type $\texttt{fun}\ x \to e$, how do we find the type to give to x?

This is the whole difference between the typing rules, which define the ternary relation

$$\Gamma \vdash e : \tau$$

and the algorithm of typing which checks that a certain expression e is well-typed in a certain environment Γ.

Simple type of mini-ML

Let us consider the approach where only function parameters are annotated and program it in OCaml

We give the abstract syntax of types

```
type typ =
  | Tint
  | Tarrow of typ * typ
  | Tproduct of typ * typ
```

Simple type of mini-ML

The constructor Fun takes an additional argument

```
type expression =
  | Var   of string
  | Const of int
  | Op    of string
  | Fun   of string * typ * expression (* the only change *)
  | App   of expression * expression
  | Pair  of expression * expression
  | Let   of string * expression * expression
```

Simple type of mini-ML

The environment Γ is realized by a persistent structure

In this case we use the OCaml `Map` module

```
module Smap = Map.Make(String)

type env = typ Smap.t
```

(performance: balanced trees $\Longrightarrow$ insertion and search in $O(\log n)$)

Simple type of mini-ML

```
let rec type_expr env = function
  | Const _ -> Tint
  | Var x -> Smap.find x env
  | Op "+" -> Tarrow (Tproduct (Tint, Tint), Tint)
  | Pair (e1, e2) ->
      Tproduct (type_expr env e1, type_expr env e2)
```

for the function, the type of the variable is given

```
  | Fun (x, ty, e) ->
      Tarrow (ty, type_expr (Smap.add x ty env) e)
```

for the local variable, it is computed as

```
  | Let (x, e1, e2) ->
      type_expr (Smap.add x (type_expr env e1) env)
                e2
```

(note the interest of the persistence of `env`))

Simple type of mini-ML

The only checks are in the application

```
    | App (e1, e2) -> begin match type_expr env e1 with
      | Tarrow (ty2, ty) ->
          if type_expr env e2 = ty2 then ty
          else failwith "error : argument of bad type"
      | _ ->
          failwith "error : function expected"
    end
```

Simple type of mini-ML

Examples

```
# type_expr
    (Let ("f",
      Fun ("x", Tint, App (Op "+", Pair (Var "x", Const 1))),
      App (Var "f", Const 2)));;
```

```
- : typ = Tint
```

```
# type_expr (Fun ("x", Tint, App (Var "x", Var "x")));;
```

```
Exception: Failure "error : function expected".
```

```
# type_expr (App (App (Op "+", Const 1), Const 2));;
```

```
Exception: Failure "error : argument of bad type".
```

In practice

- We do not do

```
failwith "error of typing"
```

  but the origin of the problem is indicated precisely

- types are preserved for later phases of the compiler

Decorated trees

On the one hand we decorate the trees at the input of the typing with a localization in the source file

```
type loc = ...
```

```
type expression =





  | Var of string
  | Const of int
  | Op of string
  | Fun of string * typ * expression
  | App of expression * expression
  | Pair of expression * expression
  | Let of string * expression * expression
```

Decorated trees

On the one hand we decorate the trees at the input of the typing with a localization in the source file

```
type loc = ...
```

```
type expression = {
    desc: desc;
    loc : loc;
}
and desc =
  | Var of string
  | Const of int
  | Op of string
  | Fun of string * typ * expression
  | App of expression * expression
  | Pair of expression * expression
  | Let of string * expression * expression
```

Signal an error

We declare an exception of the form

```
exception Error of loc * string
```

We raise it like this

```
let rec type_expr env e = match e.desc with
  | ...
  | App (e1, e2) -> begin match type_expr env e1 with
  | Tarrow (ty2, ty) ->
      if type_expr env e2 <> ty2 then
        raise (Error (e2.loc, "argument of bad type"));
      ...
```

Signal an error

and we catch up with it, for example in the main program

```
try
  let p = Parser.parse file in
  let t = Typing.program p in
  ...
with Error (loc, msg) ->
  Format.eprintf "File '%s', line ...\n" file loc;
  Format.eprintf "error: %s@." msg;
  exit 1
```

Decorated trees

on the other hand, we decorate the trees at the output of the typing with types

```
type texpression = {
    tdesc: tdesc;
    typ : typ;
}
and tdesc =
  | Tvar of string
  | Tconst of int
  | Top of string
  | Tfun of string * typ * texpression
  | Tapp of texpression * texpression
  | Tpair of texpression * texpression
  | Tlet of string * texpression * texpression
```

It's another type of expressions

Typing of typing

The typing function therefore has a type of the form

```
val type_expr: expression -> texpression
```

Typed trees

the typing function reconstructs trees, this time typed

```
let rec type_expr env e =
  let d, ty = compute_type env e in
  { tdesc = d; typ = ty }
and compute_type env e = match e.desc with
  | Const n ->
      Tconst n, Tint
  | Var x ->
      Tvar x, Smap.find x env
  | Pair (e1, e2) ->
      let te1 = type_expr env e1 in
      let te2 = type_expr env e2 in
      Tpair (te1, te2), Tproduct (te1.typ, te2.typ)
  | ...
```

Type safety

Type safety

well-typed programs do not go wrong

Type Safety

Let us show that our type system is safe wrt our small-steps semantics

Thm. (type safety)
If $\emptyset \vdash e : \tau$, then the evaluation of e is infinite or ends on a value □

Or, equivalently,

Thm.
If $\emptyset \vdash e : \tau$ and $e \xrightarrow{*} e'$ and e' is irreducible, then e' is a value □

Type safety

The proof of this theorem is based on two lemmas, called progression and preservation.

Lem. (progression)
If $\emptyset \vdash e : \tau$, then, either e is a value or there is e' such that $e \to e'$. □

Lem. (preservation)
If $\emptyset \vdash e : \tau$ and $e \to e'$ then $\emptyset \vdash e' : \tau$. □

Progression

Lem. (progression)
If $\emptyset \vdash e : \tau$, then, either e is a value or there is e' such that $e \to e'$. □

Proof.
We proceed by induction on the derivation of typing $\emptyset \vdash e : \tau$. Suppose for instance that $e = e_2\ e_1$, then we have

$$\frac{\emptyset \vdash e_2 : \tau_1 \to \tau_2 \qquad \emptyset \vdash e_1 : \tau_1}{\emptyset \vdash e_2\ e_1 : \tau_2}$$

We apply the induction hypothesis on e_2:

- if $e_2 \to e_2'$, then $e_2\ e_1 \to e_2'\ e_1$ by passage lemma in the AST lecture;
- if e_2 is a value, suppose that $e_2 = \texttt{fun}\ x \to e_3$. We apply the induction hypothesis on e_1:
  - if $e_1 \to e_1'$ then $e_2\ e_1 \to e_2\ e_1'$ by the same lemma;
  - if e_1 is a value, then $e_2\ e_1 \to e_2[x \leftarrow e_1]$.

The other cases are left as exercises. ■

Preservation

We start by two easy lemmas

Lem. (permutation)
If $\Gamma + x : \tau_1 + y : \tau_2 \vdash e : \tau$ and $x \neq y$, then $\Gamma + y : \tau_2 + x : \tau_1 \vdash e : \tau$ and the derivations have the same height. □

Proof.
By direct induction on the typing derivation ■

Lem. (weakening)
If $\Gamma \vdash e : \tau$ and $x \notin \mathrm{dom}\,\Gamma$, then $\Gamma + x : \tau' \vdash e : \tau$ and the derivations have the same height. □

Proof.
Again it follows immediately by induction on the typing derivation. ■

Preservation

We continue by a key lemma

Lem. (preservation under substitution)
If $\Gamma + x : \tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$ then $\Gamma \vdash e[x \leftarrow e'] : \tau$. □

Proof.
We proceed by induction on the derivation $\Gamma + x : \tau' \vdash e : \tau$.

- Case of a variable $e = y$:
  - if $x = y$ then $e[x \leftarrow e'] = e'$ and $\tau = \tau'$;
  - if $x \neq y$, then $e[x \leftarrow e'] = e$ and $\tau = \Gamma(y)$.
- Case of a abstract expression $e = \texttt{fun}\ y \to e_1$: We can assume $y \neq x$, $y \notin \mathrm{dom}\ (\Gamma)$ and y not free in e', even if it means renaming y. We have $\Gamma + x : \tau' + y : \tau_2 \vdash e_1 : \tau_1$ and hence $\Gamma + y : \tau_2 + x : \tau' \vdash e_1 : \tau_1$ by permutation lemma. On the other hand $\Gamma \vdash e' : \tau'$ and hence $\Gamma + y : \tau_2 \vdash e' : \tau'$ by weakening lemma. By induction hypothesis, we therefore have $\Gamma + y : \tau_2 \vdash e_1[x \leftarrow e'] : \tau_1$ and so $\Gamma \vdash (\texttt{fun}\ y \to e_1)[x \leftarrow e'] : \tau_2 \to \tau_1$, that is, $\Gamma \vdash e[x \leftarrow e'] : \tau$.

The other cases are left as an exercise. ■

Preservation

Finally we can prove the preservation lemma

Lem. (preservation)
If $\emptyset \vdash e : \tau$ and $e \to e'$ then $\emptyset \vdash e' : \tau$. □

Proof.
We proceed by induction on the derivation of $\emptyset \vdash e : \tau$.

- Case $e = \texttt{let}\ x = e_1\ \texttt{in}\ e_2$:

$$\frac{\emptyset \vdash e_1 : \tau_1 \qquad x : \tau_1 \vdash e_2 : \tau_2}{\emptyset \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

  - if $e_1 \to e_1'$, by induction hypothesis we have $\emptyset \vdash e_1' : \tau_1$ and hence $\emptyset \vdash \texttt{let}\ x = e_1'\ \texttt{in}\ e_2 : \tau_2$;
  - if e_1 is a value and $e' = e_2[x \leftarrow e_1]$, then we apply the lemma of preservation under substitution.
- Case $e = e_1\ e_2$:
  - if $e_1 \to e_1'$ or if e_1 is a value and $e_2 \to e_2'$, then we use induction hypothesis;
  - if $e_1 = \texttt{fun}\ x \to e_3$ and e_2 is a value, then $e' = e_3[x \leftarrow e_2]$ and we apply again the lemma of preservation under substitution.

The other cases are left as exercises. ■

Type safety

Now the type safety theorem can be easily derived

Thm. (type safety)
If $\emptyset \vdash e : \tau$ and $e \xrightarrow{*} e'$ with e' irreducible, then e' is a value. □

Proof.
We have $e \to e_1 \to \cdots \to e'$ and by repeatedly applying the preservation lemma, we have $\emptyset \vdash e' : \tau$. By the progress lemma, e' is reducible or is a value. By assumption, e' is a value. ■

Polymorphism

Polymorphism

It is restrictive to give a unique type to $\texttt{fun}\ x \to x$ in an expression like

$$\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ e$$

Likewise, we would like to be able to give several types to primitives such as *fst* or *snd*.

A solution: the parametric polymorphism

Parametric polymorphism

We extend the algebra of types

$$
\begin{array}{rcll}
\tau & ::= & \texttt{int} \mid \texttt{bool} \mid \ldots & \text{basic types} \\
 & \mid & \tau \to \tau & \text{function type} \\
 & \mid & \tau \times \tau & \text{pair type} \\
 & \mid & \alpha & \text{variable of type} \\
 & \mid & \forall \alpha . \tau & \text{polymorphism type}
\end{array}
$$

Free variables

We denote by $\mathcal{L}(\tau)$ the set of free type variables in τ, defined by

$$
\begin{array}{rcl}
\mathcal{L}(\texttt{int}) & = & \emptyset \\
\mathcal{L}(\alpha) & = & \{\alpha\} \\
\mathcal{L}(\tau_1 \to \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\tau_1 \times \tau_2) & = & \mathcal{L}(\tau_1) \cup \mathcal{L}(\tau_2) \\
\mathcal{L}(\forall \alpha . \tau) & = & \mathcal{L}(\tau) \setminus \{\alpha\}
\end{array}
$$

For a typing environment,we set

$$\mathcal{L}(\Gamma) = \bigcup_{x \in \mathrm{dom}\ \Gamma} \mathcal{L}(\Gamma(x)).$$

Substitution

We denote by $\tau[\alpha \leftarrow \tau']$ the substitution of α in τ by τ', defined by

$$
\begin{array}{rcl}
\texttt{int}[\alpha \leftarrow \tau'] & = & \texttt{int} \\
\alpha[\alpha \leftarrow \tau'] & = & \tau' \\
\beta[\alpha \leftarrow \tau'] & = & \beta \quad \text{if } \beta \neq \alpha \\
(\tau_1 \rightarrow \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \rightarrow \tau_2[\alpha \leftarrow \tau'] \\
(\tau_1 \times \tau_2)[\alpha \leftarrow \tau'] & = & \tau_1[\alpha \leftarrow \tau'] \times \tau_2[\alpha \leftarrow \tau'] \\
(\forall \alpha.\tau)[\alpha \leftarrow \tau'] & = & \forall \alpha.\tau \\
(\forall \beta.\tau)[\alpha \leftarrow \tau'] & = & \forall \beta.\tau[\alpha \leftarrow \tau'] \quad \text{if } \beta \neq \alpha
\end{array}
$$

F system

The rules are exactly the same as before, plus

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha . \tau} \qquad \qquad \frac{\Gamma \vdash e : \forall \alpha . \tau}{\Gamma \vdash e : \tau[\alpha \leftarrow \tau']}$$

The system obtained is call the F system (J.-Y. Girard and J. C. Reynolds)

Example

$$
\dfrac{
\dfrac{\dfrac{x : \alpha \vdash x : \alpha}{\vdash \texttt{fun}\ x \to x : \alpha \to \alpha}}{\vdash \texttt{fun}\ x \to x : \forall \alpha. \alpha \to \alpha}
\qquad
\dfrac{
\dfrac{\dfrac{\cdots \vdash f : \forall \alpha. \alpha \to \alpha}{\cdots \vdash f : \texttt{int} \to \texttt{int}} \quad \vdots}{\cdots \vdash f\ 1 : \texttt{int}}
\qquad
\dfrac{\vdots}{\cdots \vdash f\ \texttt{true} : \texttt{bool}}
}{f : \forall \alpha. \alpha \to \alpha \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}
}{\vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}
$$

Primitives

We can now give a satisfying type for primitives

$$\begin{aligned} \mathit{fst} &: \quad \forall \alpha.\forall \beta.\alpha \times \beta \to \alpha \\ \mathit{snd} &: \quad \forall \alpha.\forall \beta.\alpha \times \beta \to \beta \\ \mathit{opif} &: \quad \forall \alpha.\texttt{bool} \times \alpha \times \alpha \to \alpha \\ \mathit{opfix} &: \quad \forall \alpha.(\alpha \to \alpha) \to \alpha \end{aligned}$$

Exercise

Give a typing derivation for the expression $\Gamma \vdash \texttt{fun}\ x \to x\ x : (\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha)$.

Remark

The condition $\alpha \notin \mathcal{L}(\Gamma)$ in the rule

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\tau}$$

is crucial.

Without it, we could type $\texttt{fun}\ x \to x$ with the type $\alpha \to \forall \alpha.\alpha$ as follows:

$$\cfrac{\cfrac{\cfrac{\Gamma + x : \alpha \vdash x : \alpha}{\Gamma + x : \alpha \vdash x : \forall \alpha.\alpha}}{\Gamma \vdash \texttt{fun}\ x \to x : \alpha \to \forall \alpha.\alpha}}{\Gamma \vdash \texttt{fun}\ x \to x : \forall \alpha.\alpha \to \forall \alpha.\alpha}$$

and successfully type the expression $(\texttt{fun}\ x \to x)\ 1\ 2$, that is, a program whose execution results in the use of an integer as a function. The safety of the typing would therefore not be guaranteed.

Bad news

For terms without annotations, there are the two problems

- inference: given e, does there exist τ such that $\vdash e : \tau$?
- verification: given e and τ, do we have $\vdash e : \tau$?

are not decidable

[Wel99] J. B. Wells. Typability and type checking in the second-order lambda-calculus are equivalent and undecidable, 1994.

Hindley-Milner system

To obtain a decidable type inference, we will restrict the power of the F system

$\Longrightarrow$ Hindley-Milner system, used in OCaml, SML, Haskell, ...etc

Hindley-Milner system

We limit the universal quantifier $\forall$ at the head of the types (prenex quantification)

$$
\begin{array}{rcll}
\tau & ::= & \mathtt{int} \mid \mathtt{bool} \mid \dots & \text{basic types} \\
 & \mid & \tau \to \tau & \text{function type} \\
 & \mid & \tau \times \tau & \text{pair type} \\
 & \mid & \alpha & \text{type variable} \\
 \\
\sigma & ::= & \tau & \text{schema} \\
 & \mid & \forall \alpha . \sigma &
\end{array}
$$

The environment Γ associates a scheme of type to each identifier and the typing relation now has the form $\Gamma \vdash e : \sigma$

Example

In Hindley-Milner system, the following types are always accepted

$$\forall\alpha.\alpha \to \alpha$$

$$\forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$$

$$\forall\alpha.\texttt{bool} \times \alpha \times \alpha \to \alpha$$

$$\forall\alpha.(\alpha \to \alpha) \to \alpha$$

but not types such as

$$(\forall\alpha.\alpha \to \alpha) \to (\forall\alpha.\alpha \to \alpha).$$

Notation in OCaml

note: in OCaml syntax, prenex quantification is implicit

```
# fst;;
```

```
- : 'a * 'b -> 'a = <fun>
```

$$\forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$$

```
# List.fold_left;;
```

```
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

$$\forall\alpha.\forall\beta.(\alpha \to \beta \to \alpha) \to \alpha \to \beta\ \texttt{list} \to \alpha$$

Hindley-Milner system

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash n : \mathtt{int}} \text{ ETC.} \qquad \frac{}{\Gamma \vdash + : \mathtt{int} \times \mathtt{int} \rightarrow \mathtt{int}} \text{ ETC.}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \mathtt{fun}\ x \rightarrow e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_2 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_2\ e_1 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma + x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma} \qquad \frac{\Gamma \vdash e : \forall \alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau']}$$

Hindley-Milner system

Note that only the `let` construction allows a polymorphic type to be introduced into the environment

$$\frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma + x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \sigma_2}$$

In particular, we can always give a type of

$$\texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true})$$

with $f : \forall\alpha.\alpha \to \alpha$ in the context to type $(f\ 1, f\ \texttt{true})$

Hindley-Milner system

On the other hand, the typing rule

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\; x \to e : \tau_1 \to \tau_2}$$

does not introduce a polymorphic type, because otherwise $\tau_1 \to \tau_2$ would be poorly formed.

In particular, we can no longer type

$$\texttt{fun}\; x \to x\; x$$

Type inference

Algorithmic considerations

To program a verification or a type inference for the Hindley-Milner system, we will try to proceed by induction on the structure of the program.

However, for a given expression, three rules can apply: the rule of the monomorphic system, the rule of generalization

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathcal{L}(\Gamma)}{\Gamma \vdash e : \forall \alpha . \sigma}$$

or the rule of specialization

$$\frac{\Gamma \vdash e : \forall \alpha . \sigma}{\Gamma \vdash e : \sigma[\alpha \leftarrow \tau]}$$

How to choose? Will we have to proceed by trial and error?

Syntax-driven Hindley-Milner system

We will modify the presentation of the Hindley-Milner system so that it is syntax driven, i.e., so that, for any expression, at most one rule applies.

The rules will have the same power of expression: any closed term is typable in one system if and only if it is typable in the other.

Syntax-driven Hindley-Milner system

$$\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\tau \leq type(op)}{\Gamma \vdash op : \tau}$$

$$\frac{\Gamma + x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x \to e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma + x : Gen(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Syntax-driven Hindley-Milner system

Two operations appear

- instantiation, in the rule

$$\frac{\tau \leq \Gamma(x)}{\Gamma \vdash x : \tau}$$

the relation $\tau \leq \sigma$ reads "τ is an instance of σ" and is defined by

$$\tau \leq \forall \alpha_1 \ldots \alpha_n.\tau' \quad \text{iff} \quad \exists \tau_1 \ldots \exists \tau_n.\tau = \tau'[\alpha_1 \leftarrow \tau_1, \ldots, \alpha_n \leftarrow \tau_n]$$

example: $\texttt{int} \times \texttt{bool} \to \texttt{int} \leq \forall \alpha.\forall \beta.\alpha \times \beta \to \alpha$.

Syntax-driven Hindley-Milner system

- and the generalization, in the rule

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma + x : \mathit{Gen}(\tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

where

$$\mathit{Gen}(\tau_1, \Gamma) \overset{\text{def}}{=} \forall \alpha_1 \ldots \forall \alpha_n . \tau_1 \quad \text{where} \quad \{\alpha_1, \ldots, \alpha_n\} = \mathcal{L}(\tau_1) \setminus \mathcal{L}(\Gamma)$$

Example

$$
\dfrac{
\dfrac{\dfrac{\alpha \le \alpha}{x : \alpha \vdash x : \alpha}}{\emptyset \vdash \texttt{fun}\ x \to x : \alpha \to \alpha}
\qquad
\dfrac{
\dfrac{\dfrac{\texttt{int} \to \texttt{int} \le \forall \alpha.\alpha \to \alpha}{\Gamma \vdash f : \texttt{int} \to \texttt{int}} \quad \vdots}{\Gamma \vdash f\ 1 : \texttt{int}}
\qquad
\dfrac{\dfrac{\texttt{bool} \to \texttt{bool} \le \forall \alpha.\alpha \to \alpha}{\Gamma \vdash f : \texttt{bool} \to \texttt{bool}} \quad \vdots}{\Gamma \vdash f\ \texttt{true} : \texttt{bool}}
}{\Gamma \vdash (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}
}{\emptyset \vdash \texttt{let}\ f = \texttt{fun}\ x \to x\ \texttt{in}\ (f\ 1, f\ \texttt{true}) : \texttt{int} \times \texttt{bool}}
$$

with

$$\Gamma \stackrel{\text{def}}{=} \emptyset + f : \mathit{Gen}(\alpha \to \alpha, \emptyset) = f : \forall \alpha.\alpha \to \alpha$$

Type inference for mini-ML

To infer the type of an expression, there remain problems

- in `fun` $x \rightarrow e$, give which type to x?
- for a variable x, which instance of $\Gamma(x)$ to choose?

There exists a solution: W algorithm (Milner, Damas, Tofte [DM82])

W algorithm

Two ideas:

- new type variables are used to represent unknown types
  - for the type of x in $\mathtt{fun}\ x \to e$
  - to instantiate the schema variables $\Gamma(x)$
- the value of these variables is determined later, by unification between types at the moment of typing the application

Unification

Given two types τ_1 and τ_2 containing type variables $\alpha_1, \ldots, \alpha_n$,

is there am instantiation θ, that is, a function of the variables α_i to types, such as $\theta(\tau_1) = \theta(\tau_2)$?

We call it the unification problem

Example

$$
\begin{array}{rcl}
\tau_1 & = & \alpha \times \beta \to \texttt{int} \\
\tau_2 & = & \texttt{int} \times \texttt{bool} \to \gamma \\
\text{solution} & = & \alpha \mapsto \texttt{int}, \beta \mapsto \texttt{bool}, \gamma \mapsto \texttt{int}
\end{array}
$$

Example

$$
\begin{array}{rcl}
\tau_1 & = & \alpha \times \texttt{int} \to \alpha \times \texttt{int} \\
\tau_2 & = & \gamma \to \gamma \\
\text{solution} & = & \gamma \mapsto \alpha \times \texttt{int}
\end{array}
$$

Unification

Example

$$\begin{aligned}\tau_1 &= \alpha \to \mathtt{int}\\ \tau_2 &= \beta \times \gamma\end{aligned}$$

No solution

Example

$$\begin{aligned}\tau_1 &= \alpha \to \mathtt{int}\\ \tau_2 &= \alpha\end{aligned}$$

No solution

Unification

unifier(τ_1, τ_2) determines whether there exists an instance of variables of types of τ_1 and τ_2 such that $\tau_1 = \tau_2$

$$
\begin{array}{rcl}
\mathit{unifier}(\tau, \tau) & = & \text{success} \\
\mathit{unifier}(\tau_1 \to \tau_1', \tau_2 \to \tau_2') & = & \mathit{unifier}(\tau_1, \tau_1)\ ;\ \mathit{unifier}(\tau_1', \tau_2') \\
\mathit{unifier}(\tau_1 \times \tau_1', \tau_2 \to \tau_2') & = & \mathit{unifier}(\tau_1, \tau_1)\ ;\ \mathit{unifier}(\tau_1', \tau_2') \\
\mathit{unifier}(\alpha, \tau) & = & \text{if } \alpha \notin \mathcal{L}(\tau),\ \text{replace } \alpha \text{ by } \tau \text{ everywhere} \\
 & & \text{if not, fail} \\
\mathit{unifier}(\tau, \alpha) & = & \mathit{unifier}(\alpha, \tau) \\
\mathit{unifier}(\tau_1, \tau_2) & = & \text{fail in all the other cases}
\end{array}
$$

Idea of W algorithm

Consider the expression $\texttt{fun}\ x \to +(\mathit{fst}\ x, 1)$.

- give x the type α_1, a new type variable
- the primitive $+$ has the type $\texttt{int} \times \texttt{int} \to \texttt{int}$
- type the expression $(\mathit{fst}\ x, 1)$
  - fst has the type of schema $\forall\alpha.\forall\beta.\alpha \times \beta \to \alpha$,
  - we therefore give it the type $\alpha_2 \times \beta_1 \to \alpha_2$,
  - $\mathit{fst}\ x$ requires unifying α_1 and $\alpha_2 \times \beta_1 \Rightarrow \{\alpha_1 \mapsto \alpha_2 \times \beta_1\}$.
- $(\mathit{fst}\ x, 1)$ therefore has the type $\alpha_2 \times \texttt{int}$
- the application $+(\mathit{fst}\ x, 1)$ unifies them $\texttt{int} \times \texttt{int}$ and $\alpha_2 \times \texttt{int}$, $\Rightarrow \{\alpha_2 \mapsto \texttt{int}\}$.

In the end, we obtain the type $\texttt{int} \times \beta_1 \to \texttt{int}$, that is,

$$\vdash \texttt{fun}\ x \to +(\mathit{fst}\ x, 1) : \texttt{int} \times \beta \to \texttt{int}$$

and if we generalize (in a `let`) we therefore obtain $\forall\beta.\texttt{int} \times \beta \to \texttt{int}$

W algorithm

We define a function W which takes as arguments an environment Γ and an expression e and returns the inferred type for e

- if e is a variable x,
  return a trivial instance of $\Gamma(x)$
- if e is a constant c
  return a trivial instance of its type (think `[]` : α `list`)
- if e is a primitive op
  return a trivial instance of its type
- if e is a pair (e_1, e_2)
  compute $\tau_1 = W(\Gamma, e_1)$
  compute $\tau_2 = W(\Gamma, e_2)$
  return $\tau_1 \times \tau_2$

W algorithm

- if e is a function `fun` $x \to e_1$,
  let α be a new variable
  compute $\tau = W(\Gamma + x : \alpha, e_1)$
  return $\alpha \to \tau$

- if e is an application $e_1\ e_2$,
  compute $\tau_1 = W(\Gamma, e_1)$
  compute $\tau_2 = W(\Gamma, e_2)$
  let α be a new variable
  $\mathit{unifier}(\tau_1, \tau_2 \to \alpha)$
  return α

- if e is `let` $x = e_1$ `in` e_2,
  compute $\tau_1 = W(\Gamma, e_1)$
  return $W(\Gamma + x : \mathit{Gen}(\tau_1, \Gamma), e_2)$

Results

Thm. (Damas, Milner, 1982)
The W algorithm is correct, in the sense that

$$\text{if } W(\emptyset, e) = \tau \text{ then } \emptyset \vdash e : \tau,$$

and it determines the most general possible type, also known as principal type, in the sense that

$$\text{if } \emptyset \vdash e : \tau \text{ then } \tau \leq Gen(W(\emptyset, e), \emptyset).$$

□

Thm. (Type safety)
The Hindley-Milner system is safe.
i.e., if $\emptyset \vdash e : \tau$, then the reduction of e is infinite or ends on a value. □

Algorithmic considerations

There are several ways to achieve unification

- by explicitly manipulating a substitution

```
type tvar = int
type subst = typ TVmap.t
```

- using destructive type variables

```
type tvar = { id: int; mutable def: typ option; }
```

There are also several ways to program the W algorithm

- with explicit schemes and by calculating $Gen(\tau, \Gamma)$

```
type schema = { tvars: TVset.t; typ: typ; }
```

- with the level

$$\frac{\Gamma \vdash_{n+1} e_1 : \tau_1 \qquad \Gamma + x : (\tau_1, n) \vdash_n e_2 : \tau_2}{\Gamma \vdash_n \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Extensions

mini-ML can be extended in many ways

- recursion
- constructed types (n-tuples, lists, sum and product types)
- references

Recursion

As already explained, we can define

$$\texttt{let rec}\ f\ x \stackrel{\text{def}}{=} \texttt{let}\ f = \mathit{opfix}\ (\texttt{fun}\ f \to \texttt{fun}\ x \to e_1)\ \texttt{in}\ e_2$$

where

$$\mathit{opfix} : \forall\alpha.\forall\beta.((\alpha \to \beta) \to (\alpha \to \beta)) \to (\alpha \to \beta)$$

In an equivalent way, we can give the rule

$$\frac{\Gamma + f : \tau \to \tau_1 + x : \tau \vdash e_1 : \tau_1 \qquad \Gamma + f : \mathit{Gen}(\tau \to \tau_1, \Gamma) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2 : \tau_2}$$

Constructed types

We have already seen the pairs

Lists do not pose any difficulty

$$\begin{array}{lcl} \texttt{[]} & : & \forall\alpha.\alpha\ \texttt{list} \\ \texttt{::} & : & \forall\alpha.\alpha \times \alpha\ \texttt{list} \rightarrow \alpha\ \texttt{list} \end{array}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{list} \qquad \Gamma \vdash e_2 : \tau_1 \qquad \Gamma + x : \tau + y : \tau\ \texttt{list} \vdash e_3 : \tau_1}{\Gamma \vdash \texttt{match}\ e_1\ \texttt{with}\ \texttt{[]} \rightarrow e_2 \mid \texttt{::}(x, y) \rightarrow e_3 : \tau_1}$$

easily generalizes to sum and product types

References

For the references, one can naively think that it is enough to add the primitives

$$\begin{aligned}
\texttt{ref} &: \forall\alpha.\alpha \to \alpha\ \texttt{ref} \\
! &: \forall\alpha.\alpha\ \texttt{ref} \to \alpha \\
:= &: \forall\alpha.\alpha\ \texttt{ref} \to \alpha \to \texttt{unit}
\end{aligned}$$

References

Unfortunately this is wrong !

$$
\begin{array}{ll}
\texttt{let}\ r = \texttt{ref}\ (\texttt{fun}\ x \to x)\ \texttt{in} & r : \forall\alpha.(\alpha \to \alpha)\ \texttt{ref} \\
\texttt{let}\ \_ = r\texttt{:=}(\texttt{fun}\ x \to x\ 1)\ \texttt{in} & \\
!r = \texttt{true} & \text{boom!}
\end{array}
$$

This is the so-called polymorphic reference problem [Gar04].

Polymorphic reference

To get around this problem, there is an extremely simple solution, namely a syntactic restriction of the `let` construct

Defn. (value restriction, Wright 1995 [WF94])
A program satisfies the value restriction criterion if every let subexpression whose type is generalized is of the form

$$\texttt{let}\ x = v_1\ \texttt{in}\ e_2$$

where v_1 is a value. □

Polymorphic references

In practice, we continue to write

$$\texttt{let}\ r = \texttt{ref}\ (\texttt{fun}\ x \to x)\ \texttt{in}\ \ldots$$

but the type of r is not generalized

as if we had written

$$(\texttt{fun}\ r \to\ \ldots)\ (\texttt{ref}(\texttt{fun}\ x \to x))$$

Polymorphic references

In OCaml, a non-generalized variable is of the form '_a

```
# let x = ref (fun x -> x);;
```

```
val x : ('_a -> '_a) ref
```

The value restriction is also slightly relaxed to allow safe expressions, such as application constructor

```
# let l = [fun x -> x];;
```

```
val l : ('a -> 'a) list = [<fun>]
```

Polymorphic references

There are still some minor inconveniences

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# let f = id id;;
val f : '_a -> '_a = <fun>
# f 1;;
- : int = 1
# f true;;
This expression has type bool but is here used with type int
# f;;
- : int -> int = <fun>
```

Polymorphic references

The solution: expand to reveal a function, i.e., a value

```
# let f x = id id x;;
```

```
val f : 'a -> 'a = <fun>
```

(this is called η-expansion)

Polymorphic references

In the presence of the module system, reality is even more complex

Given a module *M*

```
module M : sig
  type 'a t
  val create : int -> 'a t
end
```

am I allowed to generalize the type of `M.create 17`?

The answer depends on the nature of the type `'a t`: no for an array, yes for a list, etc.

In OCaml, a variance indication allows us to distinguish the two

```
type +'a t (* we can generalize *)
type 'a u  (* we cannot *)
```

The solution implemented in OCaml is relatively sophisticated, see [Gar04], in particular to make it possible to indicate which type variables of an abstract type can be generalized

References i

- Luis Damas and Robin Milner.
  Principal type-schemes for functional programs.
  In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery.

- Jacques Garrigue.
  Relaxing the value restriction.
  In Yukiyoshi Kameyama and Peter J. Stuckey, editors, *Functional and Logic Programming*, pages 196–213, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

- Benjamin C. Pierce.
  Types and Programming Languages.
  The MIT Press, 1st edition, 2002.

References ii

J.B. Wells.
Typability and type checking in system f are equivalent and undecidable.
Annals of Pure and Applied Logic, 98(1):111–156, 1999.

Andrew K. Wright and Matthias Felleisen.
A syntactic approach to type soundness.
Inf. Comput., 115:38–94, 1994.

Questions?