Polymorphic Type Inference (III)

Loose ends

Sjaak Smetsers

13 March, 2025

Overview

SL vs. SPL

Mutual Recursion

Overloading

Other analyses

Fun

Conclusion

Work your way up

Mandatory

Decide what you will do (see rubric)

- Monomorphic type inference (insufficient)
- Polymorphic type inference
- Polymorphic type checking

Optional

- Overloading
- Mutual recursion
- Return path checking
- Global variables
- …

Overview

SL vs. SPL

Mutual Recursion

Overloading

Other analyses

Fun

Conclusion

SL vs. SPL

Map the typing rules

```
var x = 5;
var y = (True, 1:2:3:[]);
id (x) {
    return (x);
}
myFunction (x,y) {
    if (x) { return id(y); }
    else   { return y + 1; }
}
main () {
    print(myFunction(True, 42));
}
```

$$
\begin{array}{ll}
\textbf{let} & x = 5 \textbf{ in} \\
\textbf{let} & y = (\textbf{True}, [1,2,3]) \textbf{ in} \\
\textbf{let} & id = \lambda x.x \textbf{ in} \\
\textbf{let} & myFunction = \\
 & \quad \lambda xy.\textbf{if } x \textbf{ then } id(y) \textbf{ else } y+1 \\
\textbf{in} & print(myFunction(True, 42))
\end{array}
$$

SL vs. SPL

Statements

$$
\begin{aligned}
\mathscr{C}(\Gamma, s_1; s_2, \sigma) = {} & \mathscr{C}(\Gamma^*, s_2, \sigma^*) \circ * \\
& \; * \; = \mathscr{C}(\Gamma, s_1, \sigma) \\
\mathscr{C}(\Gamma, \textbf{while}\ (p)\ \{s\}, \sigma) = {} & \mathscr{C}(\Gamma^*, s, \sigma^*) \circ * \\
& \; * \; = \mathscr{C}(\Gamma, p, \mathit{Bool}) \\
\mathscr{C}(\Gamma, \textbf{if}\ (p)\ \{s_t\}\ \textbf{else}\ \{s_e\}, \sigma) = {} & \; * \; = \mathscr{C}(\Gamma^{*_2}, s_e, \sigma^{*_2}) \circ *_2 \\
& *_2 = \mathscr{C}(\Gamma^{*_1}, s_t, \sigma^{*_1}) \circ *_1 \\
& *_1 = \mathscr{C}(\Gamma, p, \mathit{Bool}) \\
\mathscr{C}(\Gamma, f(e_1, \ldots, e_n), \sigma) = {} & \mathscr{C}(\Gamma, f(e_1, \ldots, e_n), \alpha) \quad \color{red}{\alpha\ \mathit{fresh}} \\
\mathscr{C}(\Gamma, \textbf{return}, \sigma) = {} & \mathscr{U}(\sigma, \mathit{Void}) \\
\mathscr{C}(\Gamma, \textbf{return}\ e, \sigma) = {} & \mathscr{C}(\Gamma, e, \sigma) \\
\mathscr{C}(\Gamma, v = e, \sigma) = {} & \mathscr{C}(\Gamma, e, \tau) \\
& \mathit{where}\ \Gamma(v) = \forall.\tau
\end{aligned}
$$

Variable restriction

SPL	SL	Values	Types (Γ)
`var x = [];`	$\textbf{let}\ x = []\ \textbf{in}$	x=[]	$x :: A.a : [a]$
`x = 3:x;`	$\textbf{ref}\ x = 3 : x\ \textbf{in}$	x=[3]	$x :: A.a : [a]$
`x = True:x;`	$\textbf{ref}\ x = \mathit{True} : x\ \textbf{in}$	x=[True,3]	$x :: A.a : [a]$
...			

Recap

$$
\begin{aligned}
\mathscr{C}(\Gamma, \textbf{let}\ x = e_1\ \textbf{in}\ e_2, \sigma) &= \mathscr{C}((\Gamma^*, x{:}\forall\vec{\beta}.\alpha^*), e_2, \sigma^*) \circ * \\
\vec{\beta} &= \mathsf{TV}(\alpha^*) - \mathsf{TV}(\Gamma^*) \\
* &= \mathscr{C}((\Gamma, x{:}\alpha), e_1, \alpha),\ \alpha \text{ fresh}
\end{aligned}
$$

- Variables may be assigned to, destructively updated
- Problem when type variables appear, e.g. lambdas
- SPL only has one polymorph value: Empty list
- Variable restriction: don't generalise variables.

Overview

SL vs. SPL

Mutual Recursion

Overloading

Other analyses

Fun

Conclusion

Mutual recursion

```
odd (x) {
    if (x == 0) { return (False); }
    else { return (even (x − 1)); }
}
even (x) {
    if (x == 0) { return (True); }
    else { return (odd (x − 1)); }
}
```

```
let odd  x = if (x == 0) False (even (x-1)) in
let even x = if (x == 0) True  (odd  (x-1)) in
  odd 42
```

```
let odd  x = if (x == 0) False (even (x-1))
    even x = if (x == 0) True  (odd  (x-1)) in
   odd 42
```

Multiple let

$$
\begin{array}{llcl}
\textbf{let} & x_1 & = & e_1 \\
 & x_2 & = & e_2 \\
 & & \cdots & \\
 & x_n & = & e_n\,\textbf{in} \\
 & e & &
\end{array}
$$

Mutual Recursion (2)

- Partition the `let` in strongly connected components, and then for each one

$$\mathscr{C}(\Gamma, \textbf{let } \vec{x} = \vec{e} \textbf{ in } e, \sigma) = *$$

- where

$$
\begin{array}{rcl}
\Gamma' & = & \Gamma, \vec{x}{:}\vec{\alpha} \quad \text{where } \vec{\alpha} \text{ fresh} \\
*_1 & = & \mathscr{C}(\Gamma', e_1, \alpha_1) \\
*_2 & = & \mathscr{C}(\Gamma'^{*_1}, e_2, \alpha_2^{*_1}) \circ *_1 \\
\cdots & & \\
*_n & = & \mathscr{C}(\Gamma'^{*_{n-1}}, e_n, \alpha_2^{*_{n-1}}) \circ *_{n-1} \\
* & = & \mathscr{C}(\Gamma^{*_n}, \ldots, x_i{:}\forall\vec{\beta}_i.\alpha_i^{*}, \ldots, e, \sigma^{*_n}) \circ *_n \\
\vec{\beta}_i & = & \mathsf{TV}(\alpha_i^{*_n}) - \mathsf{TV}(\Gamma^{*_n})
\end{array}
$$

Too Much Isn't Good

If you unify definitions simultaneously that aren't mutually recursive

$$
\begin{array}{lrcl}
\textbf{let} & id & = & \lambda x.x \\
 & x & = & id\ \textbf{True} \\
 & y & = & id\ 10 \\
\textbf{in} & (x,y) & &
\end{array}
$$

$$
\begin{array}{rcl}
id & : & int \to int \\
id & : & bool \to bool \\
\mathscr{U}(int & , & bool)\ \lightning
\end{array}
$$

What to do in SPL?

Disallow mutual recursion

- First step
- A bit annoying

Burden the programmer

- Introduce syntax to group the functions that are mutually recursive

  ```
  mutrec { ... }
  ```

- Annoying too

Partition the functions in the compiler

- Strongly connected components analysis
- Awesome

How to partition

- See the AST as a (call) graph
- Function calls are edges
- Functions are vertices
- Do the analysis (tarjan's)[1]
- Convert back
- Side effect, ordering is correct as well
- Complicated in a functional language (libraries available)

Tarjan's algorithm

The data structures that he devised for this problem fit together in an amazingly beautiful way, so that the quantities you need to look at while exploring a directed graph are always magically at your fingertips. And his algorithm also does ***topological sorting as a byproduct****. — Knuth*

[1] `https://rosettacode.org/wiki/Tarjan`

Overview

SL vs. SPL

Mutual Recursion

Overloading

Other analyses

Fun

Conclusion

Type Inference vs. Overloading

- SPL knows polymorphic functions and overloaded functions
  - Polymorph: a single implementation for the same function (parametric polymorphism)
  - Doesn't touch the values
  - Overloaded: a different implementation for the same function (ad-hoc polymorphism)
  - Touch the values
- Example: **print** (5) and **print** (**True**)
- Code generator decides which function to call
- foo (x) :: a → **Void** { **print** (x); }

Solutions

- Limit the types that you can print
  - If argument type of **print** isn't supported, e.g. a type var: error
  - Supported: **Int**, **Bool**, **Char**...
  - [**Int**], (**Bool**, **Char**).
- Monomorphise the overloaded functions
  - Generate a specific version for every call
  - Don't do this for actual polymorph functions!
  - Some languages do this
- Proper overloading using class dictionaries
  - Suitable extension
  - Many languages do this

Proper overloading

- `foo (x) { print (x); }` :: a → **Void**
- Change type to: :: (a → **Void**) a → **Void**
- Change implementation to: `foo (printfun, x) { printfun (x); }`
- Either generate the specific function* or build it on the fly*
- ```
  printInt   (x) :: Int  → Void { ... }
  printBool  (x) :: Bool → Void { ... }
  printChar  (x) :: Char → Void { ... }
  printTuple (pLeft, pRight, tuple) :: (a → Void) (b → Void) (a, b) → Void {
  printList  (pEl, list) :: (a → Void) [a] → Void {
  ```

  ```
  print(True);          →  printBool(True);
  print(42);            →  printInt(42);
  print([42]);          →  printList(printInt, [42]);
  print((True, [42]));  →  printTuple(printBool, printList(printInt), (True, [42]));
  ```

- Requires higher order functions (above) or function pointers (generate all but the top level functions at compiletime), or generate all functions.
  ```

Proper overloading (2)

When using multiple overloaded functions, the extra arguments increase (or placed in a struct):

```
foo(x, y) :: a a → Void {




    if (x == y) {
        print('e':'q':':':[]);

        print(x);
    } else {
        print(x);
        print(' ':'n':'e':'q':' ':[]);

        print(y);
    }
}
```

```
foo(eqa, printa, x, y) ::
            (a a → Bool)
            (a → Void)
            a a → Void {
    if (eqa(x, y)) {
            printList(printChar
                , 'e':'q':':':[]);
            printa(x);
    } else {
            printa(x);
            printList(printChar
                , ' ':'n':'e':'q':' ':[]);
            printa(y);
    }
}
```

Overview

SL vs. SPL

Mutual Recursion

Overloading

Other analyses

Fun

Conclusion

Assignment: Semantic Analyses

- Check restrictions on globals
- Binding time analysis (definition checking)
- Type checking/inference
- Return path analyses

Restriction on globals

- Can globals call functions?
- Can globals use other globals?
- What happens if you call the print function in a global?

Make a well-founded design choice and document this.

Definition checking

- Global variables
- Local variables
- Function arguments

(Polymorphic) type inference/checking mostly does this already

Return Path Analysis

Does every function return if it needs to?

- Extra phase or during type checking
- Smart to this before type checking

Overview

SL vs. SPL

Mutual Recursion

Overloading

Other analyses

Fun

Conclusion

Fun fact

- Type inference might take very long
- ```
  f0 x = (x, x)
  f1 x = f0 (f0 x)
  f2 x = f1 (f1 x)
  f3 x = f2 (f2 x)
  f4 x = f3 (f3 x)
  f5 x = f4 (f4 x)
  ```
- What is the type signature?
  ```

Overview

SL vs. SPL

Mutual Recursion

Overloading

Other analyses

Fun

Conclusion

Work your way up

Main course

Decide what you will do (see rubric)

- Monomorphic type inference
- Polymorphic type inference
- Polymorphic type checking†

Side dishes

- Overloading
- Mutual recursion
- Return paths
- Global variables