Compiler Design - Notes Week 7

Ruben Schenk, ruben.schenk@inf.ethz.ch

November 9, 2021

9. Menhir In Practice

9.1 Menhir Output

You can get verbose ocamlyacc debugging information by doing:

`menhir --examplain`

or, if using `ocamlbuild`:

`ocamlbuild -use-menhir -yaccflag --explain`

The result is a `<basename>.conflicts` file describing the error. The flag `--dump` generates a full description of the automaton.

9.2 Precedence and Associativity Declarations

Parser generators often support **precedence/associativity declarations**. Those hint to the parser about how to resolve conflicts.

- Pros:
  - Avoids having to manually resolve those ambiguities by manually introducing extra non-terminals
  - Easier to maintain the grammar
- Cons:
  - Can't as easily re-use the same terminal
  - Introduces another level of debugging

10 Untyped Lambda Calculus

10.1 Functional Languages

Languages like ML, Haskell, Scheme, Python etc. support different operations on and with functions:

- Functions can be passed as arguments (e.g. `map` or `fold`)
- Functions can be returned as values (e.g. `compose`)
- Functions can be nested, i.e. inner functions refer to variables bound in the outer function

Example:

```
let add = fun x -> fun y -> x + y
let inc = add 1
let dec = add -1

let compose = fun f -> fun g -> fun x -> f (g x)
let id = compose inc dec
```

But how do we implement such functions in an interpreter or in a compiled language?

10.2 Lambda Calculus

The **lambda calculus** is a minimal programming language. It has variables, functions, and function application. That's it! It is, however, still touring complete.

The abstract syntax in OCaml is:

```
type exp =
    | Var of var          (* variables *)
    | Fun of var * exp    (* functions: fun x -> e *)
    | App of exp * exp    (* function application *)
```

The concrete syntax is:

```
exp ::=
    | x
    | fun x -> exp
    | exp_1 exp_2
    | (exp)
```

10.3 Values and Substitution

The only **values** of the lambda calculus are (closed) functions:

```
val ::=
    | fun x -> exp
```

To **substitute** value `v` for variable `x` in expression `e`:

- Replace all *free occurrences* of `x` in `e` by `v`
- In OCaml written as `subst v x e`

Function application is interpreted by substitution:

```
(fun x -> fun y -> x + y) 1
= subst 1 x (fun y -> x + y)
= (fun y -> 1 + y)
```

10.4 Lambda Calculus Operational Semantics

x{v/x}	= v	*(replace the free x by v)*
y{v/x}	= y	*(assuming y ≠ x)*
(fun x -> exp){v/x}	= (fun x -> exp)	*(x is bound in exp)*
(fun y -> exp){v/x}	= (fun y -> exp{v/x})	*(assuming y ≠ x)*
(e_1 e_2){v/x}	= (e_1{v/x} e_2{v/x})	*(substitute everywhere)*

10.5 Free Variables and Scoping

We look at the following example code:

```
let add = fun x -> fun y -> x + y
let inc = add 1
```

The result of `add 1` is a function. After calling `add`, we can't throw away its arguments (or its local variables) because those are needed in the function returned by `add`.

- We say that variable `x` is **free** in `fun y -> x + y` (the variable is defined in the outer scope)
- We say that variable `y` is **bound** by `fun y`. Its scope is the body `x + y` in `fun y -> x + y`

A term with no free variables is called **closed**. In contrast, a term with one or more free variables is called **open**.

10.6 Free Variable Calculation

The following OCaml code computes the set of free variables in lambda expressions:

```
let rec free_vars (e:exp) : VarSet.t =
    begin match e with
        | Var x          -> VarSet.singleton x
        | Fun (x, body) -> VarSet.remove x (free_vars body)
        | App (e1, e2)  -> VarSet.union (free_vars e1) (free_vars e2)
    end
```

We then say a lambda expression `e` is *closed* if `free_vars e` is `VarSet.empty`.

10.7 Variable Capture

Note that if we try to naively substitute an open term, a bound variable might **capture** the free variables. Example:

```
(fun x -> (x y)) {(fun z -> x)/y}    // x is free in (fun z -> x)
= fun x -> (x (fun z -> x))          // the free x is now captured
```

This is usually not the desired behavior! The meaning of `x` is determined by where it is bound dynamically, not where it is bound statically (*dynamic scoping*).

10.8 Alpha Equivalence

Note that the names of bound variables don't matter. `(fun x -> y x)` is the same as `(fun z -> y z)`. Two terms that differ only by consistent renaming of bound variables are called **alpha equivalent.**

However, the names of free variables do matter! `(fun x -> y x)` is not the same as `(fun x -> z x)`.

10.9 Fixing Substitution

We can fix the substitution problem. For this, let us consider the following substitution operation: $e_1 e_2/x$

To avoid capture, we define the substitution to pick an alpha equivalent version of e_1, such that the bound names of e_1 don't mention the free names of e_2. Then we can do the simple naive substitution.

Example:

```
(fun x -> (x y)) {(fun z -> x)/y}
= (fun x' -> (x' (fun z -> x)))      // rename x to x'
```

10.10 Operational Semantics

Specified with 2 inference rules with judgments of the form exp ⇓ v

- Read this notation as "program exp evaluates to value v"
- We give a *call-by-value* semantics
  - Function arguments are evaluated before substitution

$$\frac{}{v \Downarrow v}$$

"Values evaluate to themselves"

$$\frac{exp_1 \Downarrow (\text{fun } x \text{ -> } exp_3) \qquad exp_2 \Downarrow v \qquad exp_3\{v/x\} \Downarrow w}{exp_1\ exp_2 \Downarrow w}$$

"To evaluate function application: Evaluate the function to a value, evaluate the argument to a value, and then substitute the argument for the function. "

10.11 Adding Integers to Lambda Calculus

We might extend our previously described Lambda Calculus with **integer values** by modifying our previous definitions in the following way:

```
exp ::=
    | ...
    | n             // constant integers
    | exp1 + exp2   // binary arithmetic operation

val ::=
    | fun x -> exp  // functions are values
    | n             // integers are values

n{v/x} = n          // constants have no free variables
(e1 + e2){v/x} = (e1{v/x} + e2{v/x})
```

11. Static Analysis

11.1 Variable Scoping

We have the following problem: How do we determine whether a declared variable is in scope?

Example: The code below is syntactically correct, but not well-formed! `y` and `q` are used without being defined anywhere.

```
int fact(int x) {
    var acc = 1;
    while(x > 0) {
        acc = acc * y;
        x = q - 1;
    }
    return acc;
}
```

11.2 Contexts and Inference Rules

We somehow need to keep track of **contextual information**, i.e. what variables are in the current scope and what their types are.

One way to describe this is that the compiler keeps a mapping from variables to information about them using a **symbol table.**

11.2.1 Inference Rules

A **judgement** is of the form $G; L \vdash e : t$ is read as "*the expression* `e` *is well typed and has type* `t`".

For any **environment** $G; L$, expression `e`, and statements `s1, s2`:

$$G; L; rt \vdash \text{if } (e)\, s_1 \text{ else } s_2$$

holds if $G; L \vdash e : \text{bool}$, $G; L; rt \vdash s_1$, $G; L; rt \vdash s_2$ all hold.

More succinctly, we can summarize these constraints as an **inference rule:**

$$\frac{G; L \vdash e : \text{bool} \quad G; L; rt \vdash s_1 \quad G; L; rt \vdash s_2}{G; L; rt \vdash \text{if } (e)\, s_1 \text{ else } s_2}$$

11.2.2 Checking Derivations

We can build a **derivation tree** by making the nodes to be judgements and the edges to connect premises to a conclusion (according to the inference rules). Leaves of the tree are **axioms**, i.e. rules with no premises. The goal of the **type checker** is to verify that such a *tree exists.*

11.2.3 Compilation as Translating Judgements

Consider the typing judgement for source expressions: $C \vdash e : t$. How do we interpret this information in the target language? I.e. $[[C \vdash e : t]] = ?$ We have that:

- $[[t]]$ is a target type
- $[[e]]$ translates to a (possibly empty) sequence of instructions

We can state the following *invariant:* If $[[C \vdash e : t]] =$ ty, operand, stream, then the type of the operand is $ty = [[t]]$.

Example: What is $[[C \vdash 341 + 5 : int]]$?

```
⟦ ⊢ 341 : int ⟧ = (i64, Const 341, [])          ⟦⊢ 5 : int⟧ = (i64, Const 5, [])
----------------------------------------        ---------------------------------------
⟦C ⊢ 341 : int⟧ = (i64, Const 341, [])          ⟦C ⊢ 5 : int⟧ = (i64, Const 5, [])
-------------------------------------------------------------------------------------------
⟦C ⊢ 341 + 5 : int⟧ = (i64, %tmp, [%tmp = add i64 (Const 341) (Const 5)])
```

11.2.4 Contexts

What is $[[C]]$? Source level C has bindings like $x :$ int, $y :$ bool, etc. $[[C]]$ maps source identifiers x to source types $[[x]]$.

The interpretation of a variable $[[x]]$ can is:

$$\frac{x : t \in L}{G ; L \vdash x : t}\ \text{TYP\_VAR} \qquad \frac{x : t \in L \quad G ; L \vdash exp : t}{G ; L ; rt \vdash x = exp ; \Rightarrow L}\ \text{TYP\_ASSN}$$

as expressions (which denote values) — as addresses (which can be assigned)

11.2.5 Other Judgements

Establish invariant for expressions:

$$\left[\!\!\left[\frac{x : t \in L}{G ; L \vdash x : t}\ \text{TYP\_VAR} \right]\!\!\right] = \texttt{(\%tmp, [\%tmp = load i64* \%id\_x])}$$

as expressions (which denote values)

where (`i64`, `%id_x`) = lookup ⟦L⟧ x

Statements:

$$\left[\!\!\left[\frac{x : t \in L \quad G ; L \vdash exp : t}{G ; L ; rt \vdash x = exp ; \Rightarrow L}\ \text{TYP\_ASSN} \right]\!\!\right] = \text{stream @ } \texttt{[store [[t]] opn, [[t]]* \%id\_x]}$$

as addresses (which can be assigned)

where (t, `%id_x`) = lookup ⟦L⟧ x
and ⟦G;L ⊢ exp : t⟧ = (⟦t⟧, `opn`, stream)

$$[\![C; rt \vdash stmt \Rightarrow C']\!] = [\![C']\!], \text{stream}$$

Declaration:

⟦G;L ⊢ t x = exp ⇒ G;L,x:t ⟧ = ⟦G;L,x:t⟧, stream

Invariant: stream is of the form

```
stream' @
[ %id_x = alloca ⟦t⟧;
  store ⟦t⟧ opn, ⟦t⟧* %id_x ]
```

and ⟦G;L ⊢ exp : t ⟧ = (⟦t⟧, **opn**, stream′)

11.3 Compiling Control

11.3.1 Translating while

`⟦C;rt ⊢ while(e) s ⇒ C'⟧ = ⟦C'⟧,`

```
lpre:
   opn = ⟦C ⊢ e : bool⟧
   %test = icmp eq i1 opn, 0
   br %test, label %lpost, label %lbody
lbody:
    ⟦C;rt ⊢ s ⇒ C'⟧
    br %lpre
lpost:
```

11.3.2 Translating If-Then-Else

⟦C;rt ⊢ if (e_1) s_1 else s_2 ⇒ C′⟧ = ⟦C′⟧,

```
   opn = ⟦C ⊢ e : bool⟧
   %test = icmp eq i1 opn, 0
   br %test, label %else, label %then
then:
    ⟦C;rt ⊢ s1 ⇒ C'⟧
    br %merge
else:
    ⟦C;rt ⊢ s2 ⇒ C'⟧
    br %merge
merge:
```