

Extention of Lambda Calculus

Chapter 11

Speaker: Karl
Scribe: Pritam
12 April 2005

1 Derived Form

Definition 1 *Derived Form* : Construct that can be built from the typed λ - calculus

- usually implementing some more features
- translated before runtime

Example 1 $t_1 ; t_2 =^{def} (\lambda x : Unit. t_2)t_1$, where $x \notin FV(t_2)$. Here $t_1 : Unit$ is enforced by the type-checker.

2 A base type : Unit

Unit is an useful base type found in languages in the ML family. "Unit" is similar to "void" in C,C++,Java.

New Syntactic forms,
 $v ::= unit | \dots$ (values)
 $t ::= unit | \dots$ (terms)

New Types,
 $T ::= Unit | \dots$ (types)

New typing rules,
 $\frac{}{\Gamma \vdash unit : Unit}$ (T-UNIT)

New derived forms,
 $t_1 ; t_2 =^{def} (\lambda x : Unit. t_2)t_1$, where $x \notin FV(t_2)$

We can say, $(\lambda x. Unit.x)unit \equiv t ; unit$

Question: How many values have type Unit?

Answer: *unit* is the *only* possible result of expression of type *Unit*.

3 Derived Forms: Sequencing

In languages with side-effects, there should be a way to evaluate two or more expressions in sequence. The *Sequencing Notation* $t_1 ; t_2$ has the effect of evaluating t_1 , throwing away its trivial result, and going to evaluate t_2 .

Let us define an extension of simply typed lambda calculus called as λ_{seq} .

Syntactic form: $t = t ; t | \dots$ (terms)

New rules:

- $\frac{t_1 \rightarrow t'_1}{t_1 ; t_2 \rightarrow t'_1 ; t_2}$ (SEQ1)
- $\frac{}{unit ; t_2 \rightarrow t_2}$ (SEQ2)

- $\frac{\Gamma \vdash t_1 : Unit \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 ; t_2 : T_2}$ (TSEQ)

Question: How to check we have covered all rules?

Answer: By Progress Theorem. Let us recall the definition.

Definition 2 *Progress* : A well typed term is not stuck (either it is a value or it can take a step according to the evaluation rule). The definition implies a well typed term t either is value v or there is an evaluation rule $t \rightarrow t'$.

Suppose $\vdash t_1 ; t_2 : T_2$, suppose $t_1 ; t_2$ is stuck normal value. By Seq1 Rule, t_1 is in normal form of type *Unit*. As stated before, this implies $t_1 = unit$.

Seq2 would apply, so not stuck. Hence progress still holds.

4 Elaboration Function(e)

Let e be the *elaboration function* that translates the extended typed lambda calculus to the simply typed lambda calculus.

The definition of e is:

- $e(x) = x$
- $e(\lambda x : T_1. t_2) = \lambda x : T_1. e(t_2)$
- $e(t_1 t_2) = e(t_1)e(t_2)$
- $e(t_1 ; t_2) = e((\lambda x : Unit. t_2)t_1) = (\lambda x : Unit. e(t_2)).e(t_1)$ where $x \notin FV(t_2)$

Claim: For each term $t \in \lambda_{i}seq$

- $t \rightarrow t'$ iff $e(t) \rightarrow e(t')$
- $\Gamma \vdash t : T$ iff $\Gamma \vdash e(t) : T$

Proof: Each direction of each "iff" proceeds by straightforward induction on structure of t .

5 Let Binding

Most of the languages have its own way of giving names to its subexpressions and then using the name for further computations. In ML, for example, we write $let\ x = t_1\ in\ t_2$ to mean "evaluate the expression t_1 and bind the name x to the resulting value while evaluating t_2 ."

Language λ_{let} has

New syntactic forms

$t ::= let\ x = t_1\ in\ t_2 \text{ --- } \dots$ (terms)

New evaluation rules

- $let\ x = v_1\ in\ t_2 \rightarrow [x \mapsto v_1]t_2$ (E-LETV)
- $\frac{t_1 \rightarrow t'_1}{let\ x = t_1\ in\ t_2 \rightarrow let\ x = t'_1\ in\ t_2}$ (E-LET)

$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash let\ x = t_1\ in\ t_2 : T_2}$ (New typing rule T-LET)

Question: Why this is called less derived type?

Answer: The let binding is defined as $let\ x = t_1\ in\ t_2 =^{def} (\lambda x : T_1. t_2)\ t_1$. The right hand side of the above definition includes the type annotation T_1 , which does not appear on the l.h.s. Type-checker provides the information to the parser about the type annotation. Thus let is a "little less derived" than the other derived types: we can derive its evaluation behavior by desugaring (replacing a derived form with its lower level definition) it, but its typing behavior must be built into the internal language.

Note: In *untyped* lambda calculus let is derived form.

6 Elaboration Function with Let

- $e(x, \Gamma) = x$
- $e(t_1 t_2, \Gamma) = e(t_1, \Gamma) e(t_2, \Gamma)$
- $e(\lambda x : T_1. t_2 : \Gamma) = \lambda x : T_1. e(t_2, (\Gamma, x : T_1))$
- If $\Gamma \vdash t_1 : T_1$, then $e(\text{let } x = t_1 \text{ in } t_2, \Gamma) = e((\lambda x : T_1. t_2) t_1, \Gamma)$
- Otherwise, $e(\text{let } x = t_1 \text{ in } t_2, \Gamma) = e((\lambda x : \text{Unit}. t_2) t_1, \Gamma)$

Question: Is the elaboration function *total* for let extension?

Answer: The function is total. This can be shown by a case analysis on the structure of terms. There is one potential problem: The last line produces a nonsensical result. However, such a result is produced only if the elaboration function is given a term that is not well-typed. We conjecture that this elaboration function can be used to show that the let binding is a pure derived form.

7 Recursive Function-fix

Let us start with an iterative version of factorial.

```
/* returns n! when 0 <= n <= 99 */
int Factorial99(n){
  ...
}

/* returns n! when 0 <= n <= 100 and uses function Factorial99*/
int Factorial100(n){
  if(n==0) return 1;
  else return n * Factorial99(n-1);
}
```

```
g = λfact99 : Nat → Nat
    λn : Nat. if (equal n 0) then 1
              else (times n (fact99 (pred n)))
```

Let us erase 99 from the function body of g. g becomes..

```
g = λfact : Nat → Nat
    λn : Nat. if (equal n 0) then 1
              else (times n (fact (pred n)))
```

Say we need to define an arbitrary function to compute an answer to a problem. We can do this by defining a function $g = \lambda f : T_1. t_2$ that follows the following specification:

- Input of g : g takes a function f which returns the right answer for inputs 0...(n-1)
- Output of g : g returns a function that returns correct answer for inputs 0...n

Now (fix g) will return right answer for all natural numbers. By the definitions of fix and g, we can say that (fix g) n should have the same conceptual meaning as (g (fix g)) n. This would suggest the following rule: (fix g) n → (g (fix g)) n.

There is a problem that if this was the reduction rule for (fix g) n, then the program would not terminate. (fix g) would just be repeatedly evaluated forever. We can avoid this problem by applying g to (fix g), even though (fix g) is not a value. The idea is captured by the following rule:

```
(fix g)n → ([x → (fix g)]t2)n
```

This ensures termination of the computation, and we get something we can evaluate further.