

---

## CMCS 312: Programming Languages

### Lecture 8: Extensions and Derived Forms

Umut A. Acar

19 October 2006

---

## Contents

1	Announcements	1
2	Introduction	1
3	Base Types	1
4	Let Bindings and Derived Forms	2
5	Pairs and Product Types	3
6	Heterogeneous Data and Sum Types	4
7	Recursion	5

## 1 Announcements

Homework 3 is due today. Please turn them in. Homework 4 is available on the web and is due next Thursday (Oct 26). This is a programming homework and is reasonably substantial, so start early.

## 2 Introduction

We completed an important part of the class: typed lambda calculus and its type safety. Although lambda calculus is interesting on its own to study the properties of languages and language design, it is not a practical language—it lacks many features that you would expect from a practical language. In this class, we will grow typed lambda calculus with some features including base types, let bindings, product and sum types and recursion.

## 3 Base Types

We often want our language to have various *base types* such as *unit type*, *naturals*. Let's add unit types and naturals to our language. We first have to decide the values of these types. Let denote the only value in the unit type as  $\star$  and define the natural numbers as usual. We then decide what kind of *primitive operations* that we want to operate on these base types. For the unit type, we don't require

any primitive operations, but for naturals, we can have addition, multiplication, comparison.

<i>Types</i>	$\tau ::= \mathbf{unit} \mid \mathbf{nat} \mid \tau_1 \rightarrow \tau_2$
<i>Numbers</i>	$n ::= 0 \mid 1 \mid \dots$
<i>Prim op's</i>	$o ::= + \mid - \mid \times$
<i>Values</i>	$v ::= \star \mid n \mid \lambda x : \tau. t$
<i>Terms</i>	$t ::= x \mid v \mid o(t, t) \mid t t$
<i>Context</i>	$\Gamma ::= \emptyset \mid \Gamma, x : \tau$

We can give the following type system for this extension of lambda calculus.

$\frac{}{\Gamma \vdash \star : \mathbf{unit}}$ ( <b>unit</b> )	$\frac{}{\Gamma \vdash n : \mathbf{nat}}$ ( <b>naturals</b> )
$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$ ( <b>variables</b> )	$\frac{\Gamma \vdash t_1 : \mathbf{nat} \quad \Gamma \vdash t_2 : \mathbf{nat}}{\Gamma \vdash o(t_1, t_2) : \mathbf{nat}}$ ( <b>prim. op's</b> )
$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1. t : \tau_1 \rightarrow \tau_2}$ ( <b>lambda</b> )	$\frac{\Gamma \vdash t_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1 t_2 : \tau_2}$ ( <b>app</b> )

Similarly, we can extend the CBV (call-by-value) operational semantics for lambda calculus to support our base types. For the operational semantics, we assume that we have a *primitive application* denoted @ that given a primitive operation and the arguments for that operation gives us the value back. For example @(+, 2, 3) = 5.

$\frac{t_1 \rightarrow t'_1}{o(t_1, t_2) \rightarrow o(t'_1, t_2)}$	$\frac{t_2 \rightarrow t'_2}{o(t_1, t_2) \rightarrow o(t_1, t'_2)}$	$\frac{}{o(n_1, n_2) \rightarrow @(o, n_1, n_2)}$
$\frac{t_1 \rightarrow t'_1}{t_1 t_2 \rightarrow t'_1 t_2}$	$\frac{t_2 \rightarrow t'_2}{t_1 t_2 \rightarrow t_1 t'_2}$	$\frac{}{(\lambda x : \tau. t)v \rightarrow [v/x] t}$

## 4 Let Bindings and Derived Forms

It is often useful to be able to bind the value of an expression to a variable (e.g., SML's `let` construct). We can do this by extending our language with a `let` binding.

$t ::= \dots \mid \text{let } x : \tau_1 = t \text{ in } t \text{ end}$

Why do we need to specify the type of the variable being bound? As it will become clear when we write the typing rule for let, we will not be able to know what type to give to first part otherwise.

$$\frac{t_1 \rightarrow t'_1}{\text{let } x : \tau_1 = t_1 \text{ in } t_2 \text{ end} \rightarrow \text{let } x : \tau_1 = t'_1 \text{ in } t_2 \text{ end}} \text{ (eval-let-1)}$$

$$\frac{}{\text{let } x : \tau_1 = v \text{ in } t_2 \text{ end} \rightarrow [v/x] t_2} \text{ (eval-let-2)}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } x : \tau_1 = t_1 \text{ in } t_2 \text{ end} : \tau_2} \text{ (type-let)}$$

Both the evaluation rules and the typing rules look familiar to another expression that we know: application. Indeed, the let expression  $\text{let } x : \tau_1 = t_1 \text{ in } t_2 \text{ end}$  is equivalent to the application  $(\lambda x : \tau_1. t_2) t_1$ . In other words let bindings are *derived forms*—they can be derived using simply typed lambda calculus.

We can formally prove that let bindings are derived forms by supplying an *elaboration function* that maps the terms of the language with let bindings—call this the external language—to typed lambda calculus—call this the internal language. Formally  $\eta : t^e \mapsto t^i$ , where  $t^e$  and  $t^i$  are the terms for the external and the internal languages respectively. The elaboration function  $\eta$  simply replaces let bindings with the corresponding application and leaves all other terms the same. We can then prove derivability by proving

1.  $\Gamma \vdash^e t^e : \tau$  if and only if  $\Gamma \vdash^i (\eta(t^e)) : \tau$
2.  $t_1^e \xrightarrow{e} t_2^e$  if and only if  $\eta(t_1^e) \xrightarrow{i} \eta(t_2^e)$

**Exercise:** Prove that let bindings is derived form by specifying the elaboration function  $\eta$  and proving the two properties.

## 5 Pairs and Product Types

Most languages provide a way to build compound data structures. Perhaps the most basic form for this is pairing. Extending typed lambda calculus to support pairs is reasonably straightforward. We first introduce a *product type* for representing pairs: the type of a pair the the product of the types of its components.

$\tau ::= \dots \mid \tau \times \tau \quad t ::= \dots \mid \langle t, t \rangle \mid \text{first}(t) \mid \text{second}(t)$

The `first(·)` and `second(·)` forms project out the first and second parts of a pair. For example `first(⟨1, 2⟩) = 1`, `second(⟨1, 2⟩) = 2`.

The following are the typing rules.

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \langle t_1, t_2 \rangle : \tau_1 \times \tau_2} \text{ (type-pair)} \quad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \text{first}(t) : \tau_1} \text{ (type-project-1)}$$

$$\frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \text{second}(t) : \tau_2} \text{ (type-project-2)}$$

Here are the evaluation rules for pairs.

$$\frac{t_1 \rightarrow t_2}{\langle t_1, t_2 \rangle \rightarrow \langle t_1', t_2 \rangle} \text{ (eval-pair/1)} \quad \frac{t_2 \rightarrow t_2'}{\langle t_1, t_2 \rangle \rightarrow \langle t_1, t_2' \rangle} \text{ (eval-pair/2)}$$

$$\frac{t \rightarrow t'}{\text{first}(t) \rightarrow \text{first}(t')} \text{ (eval-first/1)} \quad \frac{}{\text{first}(\langle v_1, v_2 \rangle) \rightarrow v_1} \text{ (eval-first/2)}$$

$$\frac{t \rightarrow t'}{\text{second}(t) \rightarrow \text{second}(t')} \text{ (eval-second/1)} \quad \frac{}{\text{second}(\langle v_1, v_2 \rangle) \rightarrow v_2} \text{ (eval-second/2)}$$

## 6 Heterogeneous Data and Sum Types

We often want to express data that has heterogeneous nature. For example, a list can be empty (nil) or can have a head or tail (a cons cell). Similarly, a tree can be empty or it can be a node consisting of two children and some data.

It is well known that many programming bugs simply result from misuse of such data. For example, in the C language, any pointer is either a valid pointer or it is null. But the pointer type in C does not reflect this fact; typical C programs are full of such pointer errors (accessing null pointers, etc). It is therefore critical to ensure type safety of heterogeneous data so that their misuse can be reduced.

What should the type of a heterogeneous data be? First, the type must represent all possible forms of data. Second, it should be possible to determine the form of the data by inspecting it; one way to achieve this is to tag data.

As a concrete example, suppose we want to have data that can either be of type `unit` or of type `nat`. We can write the type of such data as `unit + nat` to indicate that it can be either one of these types. If we think of types as sets of terms, the set of this type is the union of the set of terms of type `unit` unioned with the set of terms of type `nat`. How can we write terms of this type. Remember that we want a way to tell which form the data is. So one option is to write `inl(t)` for terms where `t` has type `unit`. and `inr(t)`, where `t` has type `nat`. The tags can then tell us what to expect from the enclosed term.

Why do we need the tags? So far we have only talked about introduction forms for sums. We need the tags for the elimination form : `case` that allows us

to investigate the tag of a sum type and perform an operation on its contents. For example, the following term inspect  $t$  and prints “star” if the term is a  $\star$  or prints the natural number.

`case t of inl(x) => print"star" | inr x => print"natural : "x.`

Let’s make this intuitive description more concrete by giving the typing and evaluation rules.

$\tau ::= \dots \mid \tau_1 + \tau_2$   
 $v ::= \mathbf{inl}_{\tau_1 + \tau_2}(v) \mid \mathbf{inr}_{\tau_1 + \tau_2}(t)$   
 $t ::= \mathbf{inl}_{\tau_1 + \tau_2}(t) \mid \mathbf{inr}_{\tau_1 + \tau_2}(t) \mid (\mathbf{case } t_1 \mathbf{ of } \mathbf{inl}(x) \Rightarrow t_2 \mid \mathbf{inr}(x) \Rightarrow t_3)$

$$\frac{\Gamma \vdash t_1 : \tau_1}{\Gamma \vdash \mathbf{inl}_{\tau_1 + \tau_2}(t) : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \mathbf{inr}_{\tau_1 + \tau_2}(t) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash t_0 : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \mathbf{case } t_0 \mathbf{ of } \mathbf{inl}_{\tau_1 + \tau_2}(x_1) \Rightarrow t_1 \mid \mathbf{inr}_{\tau_1 + \tau_2}(x_2) \Rightarrow t_2 : \tau}$$

Note that we have to require the programmer specify the type of a sum type. This is important because otherwise we don’t know what type to assign to a term. For example,  $\mathbf{inl}(1)$  can have type  $\mathbf{bool} + \mathbf{nat}$  or  $\mathbf{unit} + \mathbf{nat}$ . By requiring the programmer to specify the sum type, we can ensure that each term has unique type.

The evaluation rules follow:

$$\frac{t \rightarrow t'}{\mathbf{inl}_{\tau} t \rightarrow \mathbf{inl}_{\tau} t'} \quad \frac{t \rightarrow t'}{\mathbf{inr}_{\tau} t \rightarrow \mathbf{inr}_{\tau} t'}$$

$$\frac{t_0 \rightarrow t'_0}{\mathbf{case } t_0 \mathbf{ of } \mathbf{inl}_{\tau_1 + \tau_2}(x_1) \Rightarrow t_1 \mid \mathbf{inr}_{\tau_1 + \tau_2}(x_2) \Rightarrow t_2 \rightarrow \mathbf{case } t'_0 \mathbf{ of } \mathbf{inl}_{\tau_1 + \tau_2}(x_1) \Rightarrow t_1 \mid \mathbf{inr}_{\tau_1 + \tau_2}(x_2) \Rightarrow t_2}$$

$$\frac{}{\mathbf{case } \mathbf{inl}_{\tau} v \mathbf{ of } \mathbf{inl}_{\tau_1 + \tau_2}(x_1) \Rightarrow t_1 \mid \mathbf{inr}_{\tau_1 + \tau_2}(x_2) \Rightarrow t_2 \rightarrow [v/x] t_1}$$

$$\frac{}{\mathbf{case } \mathbf{inr}_{\tau} v \mathbf{ of } \mathbf{inl}_{\tau_1 + \tau_2}(x_1) \Rightarrow t_1 \mid \mathbf{inr}_{\tau_1 + \tau_2}(x_2) \Rightarrow t_2 \rightarrow [v/x] t_2}$$

**Exercise:** Given the extension of lambda calculus with sum types, prove that booleans are derived forms.

## 7 Recursion

Previously, we showed that recursion can be “simulated” in untyped lambda calculus using the Y and Z combinators as a *fixed-point operator*. The Y combinator worked for call-by-name semantics, whereas for call by value, we needed

a slightly more complicated version, which is known as the Z combinator. It turns out that none of these combinators can be given a type. Note that by definition, types must be finite. It is instructive to try to give a type for Y and Z and see where things fail.

We therefore do not know how to express recursion in the typed lambda calculus in the calculus itself. In this class, we will instead introduce direct support for recursion by allowing the programmer to express recursive functions directly. In particular, we will allow the programmer to write a recursive function as `fix f(x) : τ is t end`.

For example a factorial function can be written as  
`fix fact(x) : nat → nat is if x < 1 then x else x * fact(x - 1) end`

$$\begin{aligned} t &::= \dots \mid \mathbf{fix} \ f(x) : \tau_1 \rightarrow \tau_2 \ \mathbf{is} \ t \ \mathbf{end} \\ v &::= \dots \mid \mathbf{fix} \ f(x) : \tau_1 \rightarrow \tau_2 \ \mathbf{is} \ t \ \mathbf{end} \end{aligned}$$

The typing rule for the `fix` operator is very similar to that of lambda abstraction, except that when type-checking the body, we get to assume that the function being defined `f` has the specified type. This allows the body of the defined function to mention itself recursively.

$$\frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \mathbf{fix} \ f(x) : \tau_1 \rightarrow \tau_2 \ \mathbf{is} \ t \ \mathbf{end} : \tau_1 \rightarrow \tau_2}$$

In the operational semantics, we change the application rule so that the function is substituted for itself—this provides for recursion. It is instructive to contrast this to what the Y and Z combinators achieve in the untyped setting.

$$\frac{t_2 \rightarrow t'_2}{(\mathbf{fix} \ f(x) : \tau_1 \rightarrow \tau_2 \ \mathbf{is} \ t \ \mathbf{end}) \ t_2 \rightarrow \mathbf{fix} \ f(x) : \tau_1 \rightarrow \tau_2 \ \mathbf{is} \ t \ \mathbf{end} \ t'_2}$$

$$\frac{}{(\mathbf{fix} \ f(x) : \tau_1 \rightarrow \tau_2 \ \mathbf{is} \ t \ \mathbf{end}) \ v \rightarrow [v/x, \mathbf{fix} \ f(x) : \tau_1 \rightarrow \tau_2 \ \mathbf{is} \ t \ \mathbf{end}/f]t}$$