**CMSC 336: Type Systems for Programming Languages**

**Lecture 7: Curry-Howard Isomorphism & Derived Forms**

**Acar & Ahmed** **31 January 2008**

# Contents



# 1 Introduction

In the last two classes we talked about simply typed lambda calculus, its operational and static semantics (type system). We proved that the language is type safe. In this lecture, we will review some themes that will recur in the study of languages and review an interesting correspondence between languages and logic, called Curry-Howard Isomorphism. Finally we will consider some extensions to the lambda calculus. The extensions provide some features that are very useful in programming.

# 2 Phase Distinction

The semantics of the simply typed lambda calculus maintains a *phase distinction* between the static and the dynamic phase of processing. The static semantics (typing rules) impose constraints on the formation of expressions that ensure that the expressions when evaluated are well-behaved. We imagine the static phase, or type checking, to be prior to evaluation.

The static phase may be seen as predicting the form of the values of an expression computed in the dynamic phase. For example, if static phase assigns

the type `bool` $\rightarrow$ `nat` to some term, it is predicting that the expression will result in function of that type. This means that, the expression can be applied to `true` without fear of error.

The type safety theorem may be viewed as stating that the predictions of the static semantics of the behavior of the dynamic semantics are accurate.

Phase distinction manifests itself in the syntax as well. The syntax of types do not involved expressions but the syntax of expression may involve types. This is because static phase occurs prior to execution—it is independent of it.

# 3 Introduction and Elimination

Consider the simply typed lambda calculus terms $t ::= c \mid x \mid \lambda x : \tau.t \mid t\ t \mid$ `if` $t$ `then` $t$ `else` $t$, where $c$ denotes some set of constants (e.g., `true`, `false`) and types $\tau ::=$ `bool` $\mid \tau \rightarrow \tau$.

There is an interesting correspondence between the terms and the types. For booleans, we have two terms `true` and `false` that *creates* the elements of the type and `if` $t$ `then` $t$ `else` $t$ that *use* elements of that type. These are called *introduction* and *elimination* forms respectively.

Similarly for arrow types, the lambda abstraction is the introduction form and application is the elimination form.

If has natural numbers as a type, then the elimination forms would be various primitive operations (e.g. sum, multiply) on natural numbers.

When an introduction for is an immediate subterm of an elimination form, the resulting expression is a redex.

We will often consider the introduction and the elimination forms of types when talking about languages and type systems.

# 4 Curry-Howard Isomorphism

The introduction/elimination form terminology refers to a connection between type theory and logic known as the *Curry-Howard isomorphism)* invented by Curry-Feys and Howard. The inspiration comes from constructive logics, where a proof of a proposition $P$ consists of a concrete evidence for $P$. This is because constructive logics do not permit tautologies such as the excluded middle, i.e., that $P \vee \not P$, which allow us to prove a proposition without producing evidence for it. For example, in classical logics, where the law of excluded middle is allowed, you can prove that $P$ holds by proving that $\not P$ is false (by the law of excluded middle, this would imply that $P$ is true). Such a proof does not provide concrete evidence that $P$ holds, just that its opposite does not hold.

Constructive logics have strong connections with computation. For example proving $P \Rightarrow Q$ requires producing a proof of $Q$ from a proof of $P$. Similarly proving $P \wedge Q$, we need to take proofs of $P$ and $A$ and give a proof of $P \wedge Q$. This gives rise to a correspondence between logics and programming languages.

- propositions $\equiv$ types

- proposition $P \Rightarrow Q \equiv$ type $P \ \rightarrow \ Q$.
- proposition $P \wedge Q \equiv$ type $P \times Q$.
- proof of a proposition $P \equiv$ terms of type $P$.
- proposition $P$ is provable $\equiv$ type $P$ is inhabited by some term.

Because of this correspondence between proposition of types, Curry-Howard isomorphism is sometimes called "propositions as types" analogy.

In other words, a term of a simply typed lambda calculus is a proof of a logical proposition. Evaluation (or reduction) corresponds to the logical operation of proof simplification by cut elimination.

This correspondence between constructive logics and programming languages covers many different logics. For example, linear logics gives rise to linear type systems.

# 5 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.

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

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

$$\frac{}{\Gamma \vdash \star : \texttt{unit}}\ \textbf{(unit)} \qquad \frac{}{\Gamma \vdash n : \texttt{nat}}\ \textbf{(naturals)}$$

$$\frac{(x,\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \textbf{(variables)} \qquad \frac{\Gamma \vdash t_1 : \texttt{nat} \quad \Gamma \vdash t_2 : \texttt{nat}}{\Gamma \vdash o(t_1, t_2) : \texttt{nat}}\ \textbf{(prim. op's)}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . t : \tau_1 \ \rightarrow\ \tau_2}\ \textbf{(lambda)} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \ \rightarrow\ \tau_2 \quad \Gamma \vdash \tau_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}\ \textbf{(app)}$$

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 $@(+, 2, 3) = 5$.

$$\frac{t_1 \ \rightarrow\ t_1'}{o(t_1, t_2) \ \rightarrow\ o(t_1', t_2)} \qquad \frac{t_2 \ \rightarrow\ t_2'}{o(t_1, t_2) \ \rightarrow\ o(t_1, t_2')} \qquad \frac{}{o(n_1, n_2) \ \rightarrow\ @(o, n_1, n_2)}$$

$$\frac{t_1 \ \rightarrow\ t_1'}{t_1\ t_2 \ \rightarrow\ t_1'\ t_2} \qquad \frac{t_2 \ \rightarrow\ t_2'}{t_1\ t_2 \ \rightarrow\ t_1\ t_2'} \qquad \frac{}{(\lambda x : \tau . t) v \ \rightarrow\ [v/x]\ t}$$

# 6 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 `let` bindings.

$$t \ ::= \ \ldots \mid \texttt{let}\ x : \tau_1 \ =\ t\ \texttt{in}\ t\ \texttt{end}$$

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.

$$\frac{t_1 \ \rightarrow\ t_1'}{\texttt{let}\ x : \tau_1 \ =\ t_1\ \texttt{in}\ t_2\ \texttt{end} \ \rightarrow\ \texttt{let}\ x : \tau_1 \ =\ t_1'\ \texttt{in}\ t_2\ \texttt{end}}\ \textbf{(eval-let-1)}$$

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

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

Both the evaluation rules and the typing rules look familiar to another expression that we know: application. Indeed, the let expression $\texttt{let}\ x : \tau_1\ =\ t_1\ \texttt{in}\ t_2\ \texttt{end}$ 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 $\eta(t_1^e) \ \xrightarrow{i}\ \eta(t_2^e)$

We can specify such a function as follows:

$$\begin{array}{rcl} \eta(v) & = & v \\ \eta(t_1\ t_2) & = & \eta(t_1)\ \eta(t_2) \\ \eta(\texttt{let}\ x : tau\ =\ t_1\ \texttt{in}\ t_2\ \texttt{end}) & = & (\lambda x : \tau_1.\eta(t_1))\ (\eta(t_2)) \end{array}$$

**Exercise:** Prove that the two properties for the $\eta$ function.

# 7 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.

$$\tau\ ::=\ \ldots \mid \tau \times \tau t\ ::=\ \ldots \mid \langle t, t\rangle \mid \texttt{first}(t) \mid \texttt{second}(t)$$

The pairing construct is the introduction form and the $\texttt{first}(\cdot)$ and $\texttt{second}(\cdot)$ are the elimination forms; they project out the first and second parts of a pair. For example $\texttt{first}(\langle 1, 2\rangle) = 1$, $\texttt{second}(\langle 1, 2\rangle) = 2$.

The following are the typing rules.

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

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

Here are the evaluation rules for pairs.

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

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

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

# 8 Heterogeneous Data and Sum Types

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

It is well known that many programming bugs simply result 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 $\texttt{unit} + \texttt{nat}$ 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` and 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 $\texttt{inl}(t)$ for terms where $t$ has type `unit`. and $\texttt{inr}(t)$, 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.

$$\texttt{case}\ t\ \texttt{of}\ \texttt{inl}(x) \Rightarrow \texttt{print}\text{"}star\text{"} \mid \texttt{inr}x \Rightarrow \texttt{print}\text{"}natural : \text{"}x.$$

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

$$
\begin{array}{rcl}
\tau & ::= & \ldots \mid \tau_1 + \tau_2 \\
v & ::= & \texttt{inl}_{\tau_1+\tau_2}(v) \mid \texttt{inr}_{\tau_1+\tau_2}(t) \\
t & ::= & \texttt{inl}_{\tau_1+\tau_2}(t) \mid \texttt{inr}_{\tau_1+\tau_2}(t) \mid (\texttt{case } t_1 \texttt{ of inl}(x) \Rightarrow t_2 \mid \texttt{inr}(x) \Rightarrow t_3)
\end{array}
$$

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

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

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, $\texttt{inl}(1)$ can have type $\texttt{bool} + \texttt{nat}$ or $\texttt{unit} + \texttt{nat}$.

The evaluation rules follow:

$$
\frac{t \;\rightarrow\; t'}{\texttt{inl}_\tau t \;\rightarrow\; \texttt{inl}_\tau t'} \qquad \frac{t \;\rightarrow\; t'}{\texttt{inr}_\tau t \;\rightarrow\; \texttt{inr}_\tau t'}
$$

$$
\frac{t_0 \;\rightarrow\; t_0'}{\begin{array}{c}\texttt{case } t_0 \texttt{ of inl}_{\tau_1+\tau_2}(x_1) \Rightarrow t_1 \mid \texttt{inr}_{\tau_1+\tau_2}(x_2) \Rightarrow t_2 \\ \rightarrow\; \texttt{case } t_0' \texttt{ of inl}_{\tau_1+\tau_2}(x_1) \Rightarrow t_1 \mid \texttt{inr}_{\tau_1+\tau_2}(x_2) \Rightarrow t_2\end{array}}
$$

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

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

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

# 9 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 (finite) type. It is instructive to try to give a type for Y and Z and see where things fail.

We therefore do not know have a way to express recursion in the typed lambda calculus in the calculus itself. In this class, we use direct support for recursion by allowing the programmer express recursive functions directly. In particular, we will write a recursive function as $\texttt{fix } f\texttt{(}x\texttt{)} : \tau \texttt{ is } t \texttt{ end}$.

For example a factorial function can be written as
$\texttt{fix}\ fact\texttt{(}x\texttt{)}\ \texttt{:}\ \texttt{nat}\ \rightarrow\ \texttt{nat}\ \texttt{is}\ \texttt{if}\ x < 1\ \texttt{then}\ x\ \texttt{else}\ x * fact(x-1)\ \texttt{end}$

$$\begin{array}{rcl} t & ::= & \ldots \mid \texttt{fix}\ f\texttt{(}x\texttt{)}\ \texttt{:}\ \tau_1\ \rightarrow\ \tau_2\ \texttt{is}\ t\ \texttt{end} \\ v & ::= & \ldots \mid \texttt{fix}\ f\texttt{(}x\texttt{)}\ \texttt{:}\ \tau_1\ \rightarrow\ \tau_2\ \texttt{is}\ t\ \texttt{end} \end{array}$$

Note that here $f$ is a meta variables, just like the terms $t$ or variables $x$ and ranges over the set of function names. 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 allow the body of the defined function to mention itself recursively.

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

In the operational semantics, we change the application rule so that the function is substituted for itself—this provides for recursion.

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

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