COS 510 Notes: Curry-Howard Isomorphism

Kiran Vodrahalli

April 7, 2014

1 The Curry-Howard Isomorphism

1.1 Introduction

The Curry-Howard isomorphism is a direct analogy between computer programs and mathematical proofs of program correctness. A pithy way that people put it is "Proofs are programs."

Definition 1.1. An **inhabited type** is a type is a type which has values. In the Curry-Howard isomorphism, we are concerned with when a given arbitrary type has values since inhabited types correspond with logically valid formulas. If we can find the values that exist for a given a type, it turns out that the type corresponds to a true mathematical theorem.

Theorem 1.2. *The **Curry-Howard isomorphsim** states that proofs of formula are programs with with a corresponding type.*

Following is a table of what programming concept the Curry-Howard isomorphism maps each logical concept to.

Constructive and Intuitionistic	Functional
Logical Concept	Programming Concept
Proof	Program
Formula	Type
Valid Formula	Inhabited Type
Proof Simplification	Program Execution

We can be even more specific and translate specific logical formulas to program types.

Logical Formula	Program Type
Implication	Function type $A \to B$
Proof of:	Function
Conjunction	Pair type $A * B$
Proof of:	(e_1, e_2)
True	Unit
Proof of:	()
False	Void
Proof of:	abort, failwith
Disjunction	sumtype
Proof of:	tagged value

Remark 1.3. The term **tagged value** refers to the notion that the value is tagged either "left" or "right". We will see this later on in the lecture.

1.2 The Language of Types

Now we define the language of types:

Definition 1.4. We define a **type**

$$\tau := \textbf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 * \tau_2. \tag{1}$$

Definition 1.5. We will be working in call-by-value lambda calculus since this is easier. Our value constructors are

$$v := () \mid \lambda x : \tau_1.e. \tag{2}$$

Definition 1.6. Our expressions are

$$e := x \mid v \mid e_1 e_2 \mid (e_1, e_2) \mid \pi_1 e \mid \pi_2 e. \tag{3}$$

Remark 1.7. Note that π_1 is equivalent to fst, and π_2 is equivalent to snd.

Definition 1.8. We define our context as a list of assumptions:

$$\Gamma := x_1 : \tau_1, x_2 : \tau_2, ..., x_n : \tau_n. \tag{4}$$

We can then proceed to give rules for proving things about this type language.

1.3 Type Rules

Definition 1.9. The **pair introduction rule** is given by

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}. \tag{5}$$

Definition 1.10. The 1^{st} **pair elimination rule** is given by

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \pi_1 e : \tau_1} \tag{6}$$

and the 2^{nd} **pair elimination rule** is given by

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash \pi_2 e : \tau_2}. \tag{7}$$

Definition 1.11. The **lambda introduction rule** is given by

$$\frac{\Gamma, x : \tau \vdash e : \tau' \quad (x \notin \Gamma)}{\Gamma \vdash \lambda x : \tau . e : \tau \to \tau'}. \tag{8}$$

Definition 1.12. The **composition introduction rule** is given by

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1 e_2 : \tau_2} \tag{9}$$

This rule describes function composition for appropriately typed functions.

Definition 1.13. We can now introduce another elimination form for pairs, the **let expression**.

$$\frac{\Gamma \vdash e_1 : \tau_1 * \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e_2 : C}{\Gamma \vdash \textbf{ let } (x_1, x_2) = e_1 \textbf{ in } e_2 : C} \tag{10}$$

The meta-idea here is that we took a concept we learned about in logic and thought, what programming language concept does this become? What types will do is prevent you from generating expressions such as $()(x_1, x_2)$ – this is a stuck state. Execution can't continue, and it's not a value. So this is bad! Types remove this possibility. Not every expression has a type, only the ones that allow us to build derivations with the values (in other words, the type must be **inhabited**).

Remark 1.14. Another useful way to think of types is to view them as predictions. You predict this expression will be a certain type, and if the expression terminates, you know what form the expression is.

Remark 1.15. Another thing to keep in mind is that we only place typing annotations on variables. Often, we add other expressions, and we may express that as follows:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau : \tau} \tag{11}$$

Example 1.16. We'll use the let expression to build a derivation for a function that swaps the order of arguments. We begin what we want to prove:

$$\frac{}{\vdash \lambda x : \tau_1 * \tau_2 . \textbf{ let } (y, z) = x \textbf{ in } (z, y) : \tau_1 * \tau_2 \to \tau_2 * \tau_1} \tag{12}$$

Then we know that directly above it must be the function introduction rule, so we write

$$\frac{x : \tau_1 * \tau_2 \vdash \ \textbf{let}\ (y, z) = x\ \textbf{in}\ (z, y) : \tau_2 * \tau_1}{\vdash \lambda x : \tau_1 * \tau_2.\ \textbf{let}\ (y, z) = x\ \textbf{in}\ (z, y) : \tau_1 * \tau_2 \to \tau_2 * \tau_1} \tag{13}$$

To get here, we must have used the let intro rule, the pair intro rule, and the hypothesis rule.

$$\frac{\dfrac{\dfrac{}{x : \tau_1 * \tau_2 \vdash x : \tau_1 * \tau_2} \qquad \dfrac{\dfrac{}{\Gamma\vdash z:\tau_2} \quad \dfrac{}{\Gamma\vdash y:\tau_1}}{x:\tau_1*\tau_2, y:\tau_1, z:\tau_2\vdash(z,y)}}{x:\tau_1*\tau_2\vdash\ \textbf{let}\ (y,z)=x\ \textbf{in}\ (z,y):\tau_2*\tau_1}}{\vdash\lambda x:\tau_1*\tau_2.\ \textbf{let}\ (y,z)=x\ \textbf{in}\ (z,y):\tau_1*\tau_2\to\tau_2*\tau_1} \tag{14}$$

Remark 1.17. Note that we can't use the following hypothesis rule:

$$\overline{\Gamma \vdash e : \tau : \tau} \tag{15}$$

This could be wrong! We need to check whether the expression is allowed first. Types always show up at the end of judgement (and not in expressions).

1.4 Disjunction, or the Logical Or

So far, we have not considered how logical disjunction appears in programs. In the table above, we have attached to disjunction the term **sumtype**. What does this mean though?

Definition 1.18. The **sumtype** $\tau_1 + \tau_2$ is essentially τ_1 OR τ_2. We expressed this in OCaml as follows:

$$\textbf{type}\ \textit{either} = \textbf{left}\ \textit{of}\ \textbf{int} \mid \textbf{right}\ \textit{of}\ \textbf{int} \tag{16}$$

Remark 1.19. Note that sumtypes are different from both pairs and records, and that a pair is basically a record with anonymous fields. As another aside, Haskell and other languages do lots of inlining and you should never decide to create a record in an attempt to optimize code – Haskell for instance has "fst" and "snd". On the other hand, a sumtupe is a *datatype* as opposed to a structure.

So we can extend τ to $\tau_1 + \tau_2$. We need to similarly extend our values and expressions.

Definition 1.20. Define

$$\textbf{inl}_{\tau_1+\tau_2}(v) \tag{17}$$

and

$$\textbf{inr}_{\tau_1+\tau_2}(v) \tag{18}$$

as the two possible values of the sumtype.

Definition 1.21. We also can add

$$\textbf{inl}_{\tau_1+\tau_2}(e) \mid \textbf{inl}_{\tau_1+\tau_2} \mid \textbf{case}\, e\ \textit{of}\,\textbf{inl}\, x \to e_1 | \,\textbf{inr}\, x \to e_2 \tag{19}$$

to our list of allowed expressions.

1.5 Operational Semantics of Sumtypes

Definition 1.22. Now we define some operational semantics for **inl** and **inr**.

$$\frac{e \to e'}{\textbf{inl}\, e \to \textbf{inl}\, e'} \ (\textbf{inl}\, 1) \tag{20}$$

$$\frac{e \to e'}{\textbf{inr}\, e \to \textbf{inr}\, e'} \ (\textbf{inr}\, 1) \tag{21}$$

Definition 1.23. We also need to be able to use **case**, as we defined it in our expressions.

$$\overline{\textbf{case}\ (\textbf{inl}\, v)\ :\ (\textbf{inl}\, x \to e_1 | \textbf{inr}\, x \to e_2)\ \to e_1[v/x]} \tag{22}$$

$$\overline{\textbf{case}\ (\textbf{inr}\, v)\ :\ (\textbf{inl}\, x \to e_1 | \textbf{inr}\, x \to e_2)\ \to e_2[v/x]} \tag{23}$$

These rules basically means that we look at the tag and decide the branch to go along. We allow e_1 to use x. We just take v out and replace it with x. Similarly we can do the same for e_2 with **inr**.

Definition 1.24. Finally, we have a transition rule for **case**.

$$\frac{e \to e'}{\textbf{case}\ e(\textbf{inl}\, x \to e_1 | \textbf{inr}\, x \to e_2) \to \textbf{case}\ e'(\textbf{inl}\, x \to e_1 | \textbf{inr}\, x \to e_2)} \ (\textbf{case'}) \tag{24}$$

1.6 Type Rules for Sumtypes

We also give the rules for sumtypes, which are very similar to the logical rules for disjunction.

Definition 1.25. First we give the rules for **inl** and **inr**.

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash\ \textbf{inl}_{\tau_1 + \tau_2} e : \tau_1 + \tau_2} \ (\textbf{inl}) \tag{25}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash\ \textbf{inr}_{\tau_1 + \tau_2} e : \tau_1 + \tau_2} \ (\textbf{inr}) \tag{26}$$

Definition 1.26. Then, the introduction rule for **case** is exactly like that of the logical rule for disjunction.

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \ \vdash e_1 : \tau_3 \quad \Gamma, y : \tau_2 \ \vdash e_2 : \tau_3}{\Gamma \vdash\ \textbf{case}\, e\ \mathit{of}(\textbf{inl}\, x \to e_1 \mid \textbf{inr}\, y \to e_2) : \tau_3} \tag{27}$$

e can have free variables, but they better appear in the context. Note that we also assume the types of x and y are correct.

2 Introduction to Type Safety

Type systems are supposed to ensure that we never get stuck.

Theorem 2.1. *The* ***Safety Property*** *states that if* $\vdash e : \tau$ *and* $e \rightarrow^* e'$*, then* e *is not stuck: either* e' *is a value or* $e' \rightarrow e''$*, meaning we can continue on.*

We can show a rule is bad if we can provide a counterexample to the safety theorem.

Example 2.2. What would happen if we replaced our pair type rule with a rule analagous to disjunction ?

We call our rule **P*** (pair star):

$$\frac{\Gamma \vdash e_1 : \tau_1 + \tau_2 \quad \Gamma, x : A, y : B \vdash C}{\Gamma \vdash \textbf{ let } (x, y) = e_1 ine_2 : C} \tag{28}$$

We want to know how this affects the legitimacy of our type system.

First we have to define the new operational semantics for $P*$:

$$\frac{e_1 \rightarrow e_1'}{\textbf{let } (x, y) = e_1 \textit{ in } e_2 \rightarrow \textbf{ let } (x, y) = e_1' \textit{ in } e_2} \tag{29}$$

$$\overline{\textbf{let}(x, y) = \textbf{ inl}\, v \textit{ in } e_2 \rightarrow e_2[v/x]} \tag{30}$$

$$\overline{\textbf{let}(x, y) = \textbf{ inr}\, v \textit{ in } e_2 \rightarrow e_2[v/y]} \tag{31}$$

So, what's going to break? We want an expression that typechecks such that $\vdash e : \tau$ and $e \rightarrow^* e'$ and e' is still stuck.

We construct the expression **let** $(x, y) = \textbf{inl}_{()+()}()\ \textit{in } y$.

We can use (30) to step to y, but y is not a value. Morever, there are no operational rules that allow us to get from a variable to anywhere else. So we are stuck at something that is not a value, and we know that we have a bad typing rule.

More generally, for any good set of typing rules, a few other properties must also hold:

Lemma 2.3. *The* ***Preservation Lemma*** *states that if we have* $\vdash e : \tau$ *and* $e \rightarrow e'$*, then* $\vdash e' : \tau$.

Lemma 2.4. *The* ***Progress Lemma*** *states that if* $\vdash e : \tau$ *then* e *is not stuck.*

In our example, we saw that progress was violated since we got to a state which did not have a type. We will prove Progress and Preservation next time, but in the meantime, we'll assume them to be true. Assuming Progress and Preservation, we are able to prove the **Safety Property**. We proceed by induction on the derivation $e \rightarrow^* e'$ (multi-step operational semantics). Our induction hypothesis is that Safety is satisfied for a slightly smaller form of each case. Our method of proof relies on reducing each case to its smaller case and applying the induction hypothesis.

Case $\overline{e \to^* e}$ (reflex): In this case, $e = e'$ and we are automatically done since by assumption of the Safety Theorem, $\Gamma \vdash e : \tau$, and e is thus not stuck by Progress.

Case $\frac{e_1 \to e_2 \quad e_2 \to^* e_3}{e_1 \to^* e_3}$ (trans): We must prove that e_3 is not stuck. Using the implication intro rule, we have $\vdash e_1 : \tau$. By Preservation, $e_1 \to e_2$, and $\vdash e_1 : \tau$, we have $\vdash e_2 : \tau$. By the induction hypothesis, e_3 is not stuck since $e_2 \to^* e_3$ and $\vdash e_2 : \tau$.

Since these are all cases for multi-step operational semantics, we are done. Q.E.D.