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 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 type, it turns out that the type corresponds to a true mathematical theorem.

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

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

<table>
<thead>
<tr>
<th>Constructive and Intuitionistic</th>
<th>Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Concept</td>
<td>Programming Concept</td>
</tr>
<tr>
<td>Proof</td>
<td>Program</td>
</tr>
<tr>
<td>Formula</td>
<td>Type</td>
</tr>
<tr>
<td>Valid Formula</td>
<td>Inhabited Type</td>
</tr>
<tr>
<td>Proof Simplification</td>
<td>Program Execution</td>
</tr>
</tbody>
</table>

We can be even more specific and translate specific logical formulas to program types.
<table>
<thead>
<tr>
<th>Logical Formula</th>
<th>Program Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implication</td>
<td>Function type $A \rightarrow B$</td>
</tr>
<tr>
<td>Proof of:</td>
<td>Function</td>
</tr>
<tr>
<td>Conjunction</td>
<td>Pair type $A * B$</td>
</tr>
<tr>
<td>Proof of:</td>
<td>($e_1, e_2$)</td>
</tr>
<tr>
<td>True</td>
<td>Unit</td>
</tr>
<tr>
<td>Proof of:</td>
<td>()</td>
</tr>
<tr>
<td>False</td>
<td>Void</td>
</tr>
<tr>
<td>Proof of:</td>
<td>abort, failwith</td>
</tr>
<tr>
<td>Disjunction</td>
<td>sumtype</td>
</tr>
<tr>
<td>Proof of:</td>
<td>tagged value</td>
</tr>
</tbody>
</table>

**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

$$\tau := \text{unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 * \tau_2. \quad (1)$$

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

$$v := () \mid \lambda x : \tau_1 . e. \quad (2)$$

**Definition 1.6.** Our expressions are

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

**Remark 1.7.** Note that $\pi_1$ is equivalent to $\text{fst}$, and $\pi_2$ is equivalent to $\text{snd}$.

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

$$\Gamma := x_1 : \tau_1, x_2 : \tau_2, ..., x_n : \tau_n. \quad (4)$$

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

$$\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \frac{}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}. \quad (5)$$
Definition 1.10. The 1\textsuperscript{st} pair elimination rule is given by
\[\frac{\Gamma \vdash e : \tau_1 \ast \tau_2}{\Gamma \vdash \pi_1 e : \tau_1}\] (6)
and the 2\textsuperscript{nd} pair elimination rule is given by
\[\frac{\Gamma \vdash e : \tau_1 \ast \tau_2}{\Gamma \vdash \pi_2 e : \tau_2}\]. (7)

Definition 1.11. The lambda introduction rule is given by
\[\frac{\Gamma, x : \tau \vdash e : \tau'}{(x \notin \Gamma)}{\frac{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'}{\Gamma \vdash \Gamma}}\] (8)

Definition 1.12. The composition introduction rule is given by
\[\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2}{\Gamma \vdash e_2 : \tau_1}{\frac{\Gamma \vdash e_1 e_2 : \tau_2}{}}\] (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.
\[\frac{\Gamma \vdash e_1 : \tau_1 \ast \tau_2}{\Gamma \vdash e_2 : \tau_1}{\frac{\Gamma, x_1, x_2 : \tau_1, \tau_2 \vdash e_2 : C}{\Gamma \vdash \text{let } (x_1, x_2) = e_1 \text{ in } e_2 : C}}\] (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:
\[\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau : \tau}\] (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:
\[\vdash \lambda x : \tau_1 \ast \tau_2. \text{let } (y, z) = x \text{ in } (z, y) : \tau_1 \ast \tau_2 \rightarrow \tau_2 \ast \tau_1\] (12)
Then we know that directly above it must be the function introduction rule, so we write

\[
\begin{align*}
  x : \tau_1 \ast \tau_2 \vdash & \quad \text{let } (y, z) = x \textbf{ in } (z, y) : \tau_2 \ast \tau_1 \\
  \vdash \lambda x : \tau_1 \ast \tau_2. & \quad \text{let } (y, z) = x \textbf{ in } (z, y) : \tau_1 \ast \tau_2 \rightarrow \tau_2 \ast \tau_1
\end{align*}
\]  

(13)

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

\[
\begin{align*}
  x : \tau_1 \ast \tau_2 \vdash & \quad x : \tau_1 \ast \tau_2 \\\n  \Gamma \vdash & \quad z : \tau_2 \\\n  \Gamma \vdash & \quad y : \tau_1 \\\n  \vdash & \quad \lambda x : \tau_1 \ast \tau_2. \text{let } (y, z) = x \textbf{ in } (z, y) : \tau_1 \ast \tau_2 \rightarrow \tau_2 \ast \tau_1
\end{align*}
\]  

(14)

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

\[
\Gamma \vdash e : \tau \vdash
\]

(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 \textbf{sumtype}. What does this mean though?

**Definition 1.18.** The \textbf{sumtype} \(\tau_1 + \tau_2\) is essentially \(\tau_1\) OR \(\tau_2\). We expressed this in OCaml as follows:

\[
\text{type either} = \text{left of int} \mid \text{right of int}
\]  

(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 sumtype is a \textbf{datatype} as opposed to a structure.

So we can extend \(\tau\) to \(\tau_1 + \tau_2\). We need to similarly extend our values and expressions.

**Definition 1.20.** Define

\[
\text{inl}_{\tau_1 + \tau_2}(v)
\]  

(17)

and

\[
\text{inr}_{\tau_1 + \tau_2}(v)
\]  

(18)

as the two possible values of the sumtype.

**Definition 1.21.** We also can add

\[
\text{inl}_{\tau_1 + \tau_2}(e) \mid \text{inl}_{\tau_1 + \tau_2} \mid \text{case } e \text{ of } \text{inl } x \rightarrow e_1 \mid \text{inr } x \rightarrow e_2
\]  

(19)

to our list of allowed expressions.
1.5 Operational Semantics of Sumtypes

**Definition 1.22.** Now we define some operational semantics for \texttt{inl} and \texttt{inr}.

\[
\begin{align*}
\text{e} &\rightarrow \text{e}' \quad \text{(inl 1)} \\
\text{inl e} &\rightarrow \text{inl e}' \\
\text{e} &\rightarrow \text{e}' \quad \text{(inr 1)} \\
\text{inr e} &\rightarrow \text{inr e}'
\end{align*}
\]

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

\[
\begin{align*}
\text{case (inl v)} : \text{(inl x \rightarrow e1 | inr x \rightarrow e2)} &\rightarrow e1[v/x] \\
\text{case (inr v)} : \text{(inl x \rightarrow e1 | inr x \rightarrow e2)} &\rightarrow e2[v/x]
\end{align*}
\]

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 \texttt{inr}.

**Definition 1.24.** Finally, we have a transition rule for \texttt{case}.

\[
\begin{align*}
\text{e} &\rightarrow \text{e}' \\
\text{case e (inl x \rightarrow e1 | inr x \rightarrow e2)} &\rightarrow \text{case e' (inl x \rightarrow e1 | inr x \rightarrow e2)} \quad \text{(case')} \quad (24)
\end{align*}
\]

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 \texttt{inl} and \texttt{inr}.

\[
\begin{align*}
\Gamma \vdash e : \tau_1 \quad \text{(inl)} \\
\Gamma \vdash \text{inl}_1 + \tau_2 e : \tau_1 + \tau_2 \quad (25)
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \tau_2 \quad \text{(inr)} \\
\Gamma \vdash \text{inr}_1 + \tau_2 e : \tau_1 + \tau_2 \quad (26)
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma, x : \tau_1 \vdash e_1 : \tau_3 \quad \Gamma, y : \tau_2 \vdash e_2 : \tau_3 \quad \Gamma \vdash \text{case e of (inl x \rightarrow e1 | inr y \rightarrow e2)} : \tau_3 \quad (27)
\end{align*}
\]

\(e\) 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 analogous to disjunction?

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

$$
\frac{\Gamma \vdash e_1 : \tau_1 + \tau_2 \quad \Gamma, x : A, y : B \vdash C}{\Gamma \vdash \text{let} (x,y) = e_1 \text{in} e_2 : C}
$$

(28)

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

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

$$
e_1 \rightarrow e'_1
$$

$$
\text{let} (x,y) = e_1 \text{ in } e_2 \rightarrow \text{let} (x,y) = e'_1 \text{ in } e_2
$$

(29)

$$
\text{let} (x,y) = \text{inl} v \text{ in } e_2 \rightarrow e_2[v/x]
$$

(30)

$$
\text{let} (x,y) = \text{inr} v \text{ in } e_2 \rightarrow e_2[v/y]
$$

(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 $\text{let} (x,y) = \text{inl} (\_ + (\_)) \text{ in } y$.

We can use (30) to step to $y$, but $y$ is not a value. Moreover, 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 $e \rightarrow^* e$ (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 $e_1 \rightarrow e_2 \quad e_2 \rightarrow^* e_3$ (trans): We must prove that $e_3$ is not stuck. Using the implication intro rule, we have $\vdash e_1 : \tau$. By Preservation, $e_1 \rightarrow e_2$, and $\vdash e_1 : \tau$, we have $\vdash e_2 : \tau$. By the induction hypothesis, $e_3$ is not stuck since $e_2 \rightarrow^* e_3$ and $\vdash e_2 : \tau$.

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