Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Curry-Howard Correspondence; Existential types

Lecture 15 Tuesday, March 22, 2022

1 Curry-Howard Correspondence

There is a strong connection between types in programming languages and propositions in *constructive logic* (also called *intuitionistic logic*). This correspondence was noticed by Haskell Curry and William Howard. It is known as the *Curry-Howard correspondence*, and also as the *propositions-as-types* correspondence, and *proofs-as-programs* correspondence.

Constructive logic equates the truth of formula with their provability. That is, for a statement ϕ to be true, there must be a proof of ϕ. The key difference between constructive logic and classical logic is that in constructive logic, the rule of excluded middle does not apply: it is not a tautology that either ϕ or $\neg\phi$.

The inference rules and axioms for typing programs are very similar to the inference rules and axioms for proving formulas in constructive logic. That is, types are like formulas, and programs are like proofs.

Conjunction = Product types For example, suppose we have an expression e_1 with type τ_1, and expression e_2 with type τ_2. Think of e_1 as a proof of some logical formulas τ_1, and e_2 as a proof of some logical formulas τ_2. What would constitute a proof of the formulas $\tau_1 \wedge \tau_2$? We would need a proof of τ_1 and a proof of τ_2. Say we put these proofs together in a pair: (e_1, e_2). This is a program with type $\tau_1 \times \tau_2$. That is, the product type $\tau_1 \times \tau_2$ corresponds to conjunction!

Disjunction = Sum types Similarly, how do we prove $\tau_1 \vee \tau_2$? Under constructive logic, we need either a proof of τ_1, or a proof of τ_2. Thinking about programs and types, this means we need either an expression of type τ_1 or an expression of type τ_2. We have a construct that meets this description: the sum type $\tau_1 + \tau_2$ corresponds to disjunction!

Implication = Function types What does the function type $\tau_1 \to \tau_2$ correspond to? We can think of a function of type $\tau_1 \to \tau_2$ as taking an expression of type τ_1 and producing something of type τ_2, which by the Curry-Howard correspondence, means taking a proof of proposition τ_1 and producing a proof of proposition τ_2. This corresponds to implication: if τ_1 is true, then τ_2 is true.

Universal quantification = Parametric polymorphism The polymorphic lambda calculus introduced universal quantification over types: $\forall X.\ \tau$. As the notation suggests, this corresponds to universal quantification in constructive logic. To prove formula $\forall X.\ \tau$, we would need a way to prove $\tau\{\tau'/X\}$ for all propositions τ'. This is what the expression $\Lambda X.\ e$ gives us: for any type τ', the type of the expression $(\Lambda X.\ e)\ [\tau']$ is $\tau\{\tau'/X\}$, where τ is the type of e.

Invalidity = uninhabited type So under the Curry-Howard correspondence, expression e of type τ is a proof of proposition τ. If we have a proposition τ that is not true, then there is no proof for τ, i.e., there is no expression e of type τ. A type that has no expressions with that type is called an *uninhabited type*. There are many uninhabited types, such as $\forall X.\ X$. Uninhabited types correspond to false formulas. Inhabited types are theorems.

1.1 Examples

Consider the formula

$$\forall \phi_1, \phi_2, \phi_3.\ ((\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_3)) \Rightarrow (\phi_1 \Rightarrow \phi_3).$$

The type corresponding to this formula is

$$\forall X, Y, Z.\ ((X \to Y) \times (Y \to Z)) \to (X \to Z).$$

This formula is a tautology. So there is a proof of the formula. By the Curry-Howard correspondence, there should be an expression with the type $\forall X, Y, Z.\ ((X \to Y) \times (Y \to Z)) \to (X \to Z)$. Indeed, the following is an expression with the appropriate type.

$$\Lambda X, Y, Z.\ \lambda f\!:\!(X \to Y) \times (Y \to Z).\ \lambda x\!:\!X.\ (\#2\ f)\ ((\#1\ f)\ x)$$

We saw earlier in the course that we can curry a function. That is, given a function of type $(\tau_1 \times \tau_2) \to \tau_3$, we can give a function of type $\tau_1 \to \tau_2 \to \tau_3$. We can do this with a function. That is, the expression

$$\lambda f\!:\!(\tau_1 \times \tau_2) \to \tau_3.\ \lambda x\!:\!\tau_1.\ \lambda y\!:\!\tau_2.\ f\ (x, y)$$

has type

$$((\tau_1 \times \tau_2) \to \tau_3) \to (\tau_1 \to \tau_2 \to \tau_3).$$

The corresponding logical formula is $(\phi_1 \wedge \phi_2 \Rightarrow \phi_3) \Rightarrow (\phi_1 \Rightarrow (\phi_2 \Rightarrow \phi_3))$, which is a tautology.

1.2 Negation and continuations

In constructive logic, if $\neg\tau$ is true, then τ is false, meaning there is no proof of τ.We can think of $\neg\tau$ as being equivalent to $\tau \Rightarrow$ **False**, or, as the type $\tau \to \bot$, where $\bot$ is some uninhabited type such as $\forall X.\ X$. That is, if $\neg\tau$ is true, then if you give me a proof of τ, I can give you a proof of **False**.

We have seen functions that take an argument, and never produce a result: continuations. Continuations can be thought of as corresponding to negation.

Suppose that we have a special type **Answer** that is the return type of continuations. That is, a continuation that takes an argument of type τ has the type $\tau \to$ **Answer**. Assume further that we have no values of type **Answer**, i.e., **Answer** is an uninhabited type.

A continuation-passing style translation of an expression e of type τ, $\mathcal{CPS}[\![e]\!]$, has the form $\lambda k\!:\![\![\tau]\!] \to$ **Answer**. $\dots$, where k is a continuation, $[\![\tau]\!]$ is the translated type of τ, and the translation will evaluate e, and give the result to k. Thus, the type of $\mathcal{CPS}[\![e]\!]$ is $([\![\tau]\!] \to$ **Answer**$) \to$ **Answer**. Under the Curry-Howard correspondence, this type corresponds to $([\![\tau]\!] \Rightarrow$ **False**$) \Rightarrow$ **False**, or, equivalently, $\neg(\neg[\![\tau]\!])$, the double negation of (the translation of) τ, which is equivalent to τ. CPS translation converts an expression of type τ to an expression of type $([\![\tau]\!] \to$ **Answer**$) \to$ **Answer**, which is equivalent to τ!

1.3 Theorem Proving

The Curry-Howard correspondence tells us that types correspond to formulas, or propositions, and that programs correspond to proofs. So, by this analogy, when we have a proof of a theorem, we have a program. What does it mean to run this program? That is, what is the executable content of a proof?

Well, let's consider the following tautology in a propositional logic, where p, q, and r are propositions.

$$(p \Rightarrow (q \Rightarrow r)) \quad \Rightarrow \quad (p \wedge q) \Rightarrow r$$

The following is a derivation of this tautology.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \Rightarrow (q \Rightarrow r) \vdash p \Rightarrow (q \Rightarrow r)} \qquad \dfrac{\dfrac{}{p \wedge q \vdash p \wedge q}}{p \wedge q \vdash p}}{p \Rightarrow (q \Rightarrow r), p \wedge q \vdash q \Rightarrow r} \qquad \dfrac{\dfrac{}{p \wedge q \vdash p \wedge q}}{p \wedge q \vdash q}}{p \Rightarrow (q \Rightarrow r), p \wedge q, p \wedge q \vdash r}}{p \Rightarrow (q \Rightarrow r), p \wedge q \vdash r}}{p \Rightarrow (q \Rightarrow r) \vdash (p \wedge q) \Rightarrow r}}{\vdash (p \Rightarrow (q \Rightarrow r)) \quad \Rightarrow \quad (p \wedge q) \Rightarrow r}$$

(Note the use of Contraction to duplicate the assumption $p \land q$.)

Now, let's use the Curry-Howard correspondence, and think about what a program would look like that had the type

$$(\tau_1 \to (\tau_2 \to \tau_3)) \to (\tau_1 \times \tau_2) \to \tau_3$$

which corresponds to the tautological formula. Here we present the proof that such a program is well-typed. There are a few things to note. First, the structure of the proof of well typedness has exactly the same structure as the proof of the formula. We have highlighted the terms in red, leaving the types in black. If we think about constructing the terms from the leaves of the proof down towards the root, note that term construction is mechanical, based on which inference rule is applied.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f\!:\!\tau_1 \to (\tau_2 \to \tau_3) \vdash f\!:\!\tau_1 \to (\tau_2 \to \tau_3)} \quad \dfrac{\dfrac{}{a\!:\!\tau_1 \times \tau_2 \vdash a\!:\!\tau_1 \times \tau_2}}{a\!:\!\tau_1 \times \tau_2 \vdash \#1\ a\!:\!\tau_1}}{f\!:\!\tau_1 \to (\tau_2 \to \tau_3), a\!:\!\tau_1 \times \tau_2 \vdash f\ (\#1\ a)\!:\!\tau_2 \to \tau_3} \quad \dfrac{\dfrac{}{a\!:\!\tau_1 \times \tau_2 \vdash a\!:\!\tau_1 \times \tau_2}}{a\!:\!\tau_1 \times \tau_2 \vdash \#2\ a\!:\!\tau_2}}{f\!:\!\tau_1 \to (\tau_2 \to \tau_3), a\!:\!\tau_1 \times \tau_2, a\!:\!\tau_1 \times \tau_2 \vdash f\ (\#1\ a)\ (\#2\ a)\!:\!\tau_3}}{f\!:\!\tau_1 \to (\tau_2 \to \tau_3), a\!:\!\tau_1 \times \tau_2 \vdash f\ (\#1\ a)\ (\#2\ a)\!:\!\tau_3}}{f\!:\!\tau_1 \to (\tau_2 \to \tau_3) \vdash \lambda a\!:\!\tau_1 \times \tau_2.\ f\ (\#1\ a)\ (\#2\ a)\ :(\tau_1 \times \tau_2) \to \tau_3}}{\vdash \lambda f\!:\!(\tau_1 \to (\tau_2 \to \tau_3)).\ \lambda a\!:\!\tau_1 \times \tau_2.\ f\ (\#1\ a)\ (\#2\ a)\!:\!(\tau_1 \to (\tau_2 \to \tau_3)) \to (\tau_1 \times \tau_2) \to \tau_3}$$

Now, what is this program?

$$\lambda f\!:\!(\tau_1 \to (\tau_2 \to \tau_3)).\ \lambda a\!:\!\tau_1 \times \tau_2.\ f\ (\#1\ a)\ (\#2\ a)$$

This is the uncurry function! It takes in a function of type $\tau_1 \to (\tau_2 \to \tau_3)$ and uncurrys the arguments, producing a function of type $(\tau_1 \times \tau_2) \to \tau_3$. We got this program in a mechanical way, given the proof of the formula $(p \Rightarrow (q \Rightarrow r)) \quad \Rightarrow \quad (p \land q) \Rightarrow r$.

More generally, if the logic is set up correctly, it is possible to extract executable content from proofs of many theorems. For example, a proof of the pigeon-hole principal can provide an algorithm to find a pigeon hole with at least two elements. More impressively, the executable content of a proof that it is possible for a distributed system to reach an agreement is a protocol for achieving agreement.

2 Existential types

We saw above that universal quantification corresponds to parametric polymorphism. What does existential quantification correspond to? Turns out we can define *existential types*, which, by the Curry-Howard correspondence, correspond to existential quantification.

We extend the simply-typed lambda calculus with *existential types* (and records). An existential type is written $\exists X.\ \tau$, where type variable X may occur in τ. If a value has type $\exists X.\ \tau$, it means that it is a pair $\{\tau', v\}$ of a type τ' and a value v, such that v has type $\tau\{\tau'/X\}$.

Thinking about the Curry-Howard correspondence may provide some intuition for existential types. As the notation and name suggest, the logical formula that corresponds to an existential type $\exists X.\ \tau$ is an existential formula $\exists X.\ \phi$, where X may occur in ϕ. In constructive logic, what would it mean for the statement "there exists some X such that ϕ is true" to be true? In constructive logic, a statement is true only if there is a proof for it. To prove "there exists some X such that ϕ is true" we must actually provide a *witness* ψ, an entity that is a suitable replacement for X, and also, a proof that ϕ is true when we replace X with witness ψ.

A value $\{\tau', v\}$ of type $\exists X.\ \tau$ exactly corresponds to a proof of an existential statement: type τ' is the witness type, and v is a value with type $\tau\{\tau'/X\}$.

We introduce a language construct to create existential values, and a construct to use existential values. The syntax of the new language is given by the following grammar.

$$\begin{aligned} e ::= {} & x \mid \lambda x\!:\!\tau.\, e \mid e_1\; e_2 \mid n \mid e_1 + e_2 \\ & \mid \{\, l_1 = e_1, \ldots, l_n = e_n \,\} \mid e.l \\ & \mid \textsf{pack}\ \{\tau_1, e\}\ \textsf{as}\ \exists X.\, \tau_2 \mid \textsf{unpack}\ \{X, x\} = e_1\ \textsf{in}\ e_2 \\ v ::= {} & n \mid \lambda x\!:\!\tau.\, e \mid \{\, l_1 = v_1, \ldots, l_n = v_n \,\} \mid \textsf{pack}\ \{\tau_1, v\}\ \textsf{as}\ \exists X.\, \tau_2 \\ \tau ::= {} & \textbf{int} \mid \tau_1 \to \tau_2 \mid \{\, l_1\!:\!\tau_1, \ldots, l_n\!:\!\tau_n \,\} \mid X \mid \exists X.\, \tau \end{aligned}$$

Note that in this grammar, we annotate existential values with their existential type. The construct to create an existential value, $\textsf{pack}\ \{\tau_1, e\}\ \textsf{as}\ \exists X.\, \tau_2$, is often called *packing*, and the construct to use an existential value is called *unpacking*.

Before we present the operational semantics and typing rules, let's see some examples to get an intuition for packing and unpacking. Existential types provide us with a mechanism to reason about *modules* which can hide their implementation details. That is, a module that wants to hide away its internal details tells the external world that there is *some* type or types that describe its internal structures and implementation, but the clients are not allowed to know anything about these implementation types, simply that they exist.

Here we create an existential value that implements a counter, without revealing details of its implementation.

$$\begin{aligned} & \textsf{let}\ \mathit{counterADT} = \\ & \quad \textsf{pack} \\ & \quad\quad \{\textbf{int}, \{\ \textsf{new} = 0, \textsf{get} = \lambda i\!:\!\textbf{int}.\, i, \textsf{inc} = \lambda i\!:\!\textbf{int}.\, i+1\ \}\ \} \\ & \quad \textsf{as} \\ & \quad\quad \exists \textbf{Counter}.\ \{\ \textsf{new} : \textbf{Counter}, \textsf{get} : \textbf{Counter} \to \textbf{int}, \textsf{inc} : \textbf{Counter} \to \textbf{Counter}\ \} \\ & \textsf{in} \ldots \end{aligned}$$

The abstract type name is **Counter**, and its concrete representation is **int**. The type of the variable $\mathit{counterADT}$ is $\exists\textbf{Counter}.\ \{\ \textsf{new} : \textbf{Counter}, \textsf{get} : \textbf{Counter} \to \textbf{int}, \textsf{inc} : \textbf{Counter} \to \textbf{Counter}\ \}$.

We can use the existential value $\mathit{counterADT}$ as follows.

$$\textsf{unpack}\ \{C, x\} = \mathit{counterADT}\ \textsf{in let}\ y\!:\!C = x.\textsf{new}\ \textsf{in}\ x.\textsf{get}\ (x.\textsf{inc}\ (x.\textsf{inc}\ y))$$

Note that we annotate the pack construct with the existential type. That is, we explicitly state the type $\exists\textbf{Counter}.\ \ldots$. Why is this? Without this annotation, we would not know which occurrences of the witness type are intended to be replaced with the type variable, and which are intended to be left as the witness type. In the counter example above, the type of expressions $\lambda i\!:\!\textbf{int}.\, i$ and $\lambda i\!:\!\textbf{int}.\, i+1$ are both $\textbf{int} \to \textbf{int}$, but one is the implementation of get, of type $\textbf{Counter} \to \textbf{int}$ and the other is the implementation of inc, of type $\textbf{Counter} \to \textbf{Counter}$.

We now define the operational semantics. We add two new evaluation contexts, and one evaluation rule for unpacking an existential value.

$$E ::= \cdots \mid \textsf{pack}\ \{\tau_1, E\}\ \textsf{as}\ \exists X.\, \tau_2 \mid \textsf{unpack}\ \{X, x\} = E\ \textsf{in}\ e$$

$$\frac{}{\textsf{unpack}\ \{X, x\} = (\textsf{pack}\ \{\tau_1, v\}\ \textsf{as}\ \exists Y.\, \tau_2)\ \textsf{in}\ e \longrightarrow e\{v/x\}\{\tau_1/X\}}$$

The new typing rules make sure that existential values are used correctly. Note that code using an existential value (e_2 in $\textsf{unpack}\ \{X, x\} = e_1\ \textsf{in}\ e_2$) does not know the witness type of the existential value of type $\exists X.\, \tau_1$.

$$\frac{\Delta, \Gamma \vdash e\!:\!\tau_2\{\tau_1/X\}}{\Delta, \Gamma \vdash \textsf{pack}\ \{\tau_1, e\}\ \textsf{as}\ \exists X.\, \tau_2 : \exists X.\, \tau_2}$$

$$\frac{\Delta, \Gamma \vdash e_1 : \exists X.\, \tau_1 \quad X \notin \Delta \quad \Delta \cup \{X\}, \Gamma, x\!:\!\tau_1 \vdash e_2 : \tau_2 \quad \Delta \vdash \tau_2\ \textsf{ok}}{\Delta, \Gamma \vdash \textsf{unpack}\ \{X, x\} = e_1\ \textsf{in}\ e_2 : \tau_2} \qquad \frac{\Delta \cup \{X\} \vdash \tau\ \textsf{ok}}{\Delta \vdash \exists X.\, \tau\ \textsf{ok}}$$

Note that we define well-formedness of existential types, similar to well-formedness of universal types. In the typing rule for $\mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2$, note that we have the premises $X \notin \Delta$ and $\Delta \vdash \tau_2\ \mathsf{ok}$. The first ensures that X is not currently a type variable in scope (and we can alpha-vary it to ensure that this holds true). Why do we need the premise $\Delta \vdash \tau_2\ \mathsf{ok}$?