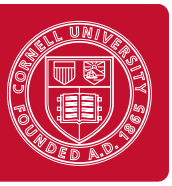
CS 4110 – Programming Languages and Logics
Lecture #21: Propostions-as-Types, Existentials



Based on material by Stephen Chong, Greg Morrisett, Andrew Myers, George Necula, and Radu Rugina

1 Propositions-as-Types

There is a strong connection between types in programming languages and propositions in *intuitionistic logic*. This connection is known as the *propositions-as-types correspondence* or sometimes the *Curry-Howard isomorphism*.

Intuitionistic logic equates the truth of a formula with its provability (where the notion of provability is more restrictive than in classical logic, requiring constructive evidence for the truth of the formula). That is, for a statement ϕ to be true, we must be able to find a proof of ϕ. Unlike classical logic, the rule of excluded middle does not apply in intuitionistic logic: $\phi \vee \neg\phi$ is not always true. It turns out that the typing rules for λ-calculus are similar to the inference rules and axioms for proving formulas in intuitionistic logic. That is, types are like formulas, and programs are like proofs. Consider some examples:

Products 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!

Sums Similarly, how would we prove $\tau_1 \vee \tau_2$? Under intuitionistic 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!

Functions 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 isomorphism, means taking a proof of proposition τ_1 and producing a proof of proposition τ_2.

Universal Types The polymorphic lambda calculus introduced universal quantification over types: $\forall X.\, \tau$. As the notation suggests, this corresponds to universal quantification in intuitionistic 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.

So under the Curry-Howard isomorphism, an 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 formulas that do not have a proof. 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 isomorphism, 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.

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

Now consider the higher-order function that, given a function of type $(\tau_1 \times \tau_2) \to \tau_3$, produces a function of type $\tau_1 \to \tau_2 \to \tau_3$. This function, often called *curry*,

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

has type

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

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

2 Modules

Some simple languages such as C and FORTRAN, have a single global namespace. This often causes problems because in large programs, name collisions—i.e., two different programmers (or pieces of code) using the same name for different purposes—are likely. Also, components of a program may be more tightly coupled, since two components are coupled simply by one using a name defined by the other.

Modular programming addresses these issues. A *module* is a collection of named entities that are related to each other in some way. Modules provide separate namespaces: different modules have different name spaces, and so can freely use names without worrying about name collisions.

Typically, a module can choose what names/entities to export (i.e., which names to allow to be used outside of the module), and what to keep hidden. The exported entities are declared in an *interface*, and the interface typically does not export details of the implementation. This means that different modules can implement the same interface in different ways. Also, by hiding the details of module implementation, and preventing access to these details except through the exported interface, programmers of modules can be confident that code invariants are not broken.

Packages in Java are a form of modules. A package provides a separate namespace (we can have a class called `Foo` in package `p1` and package `p2` without any conflicts). A package can hide details of its implementation by using private and package-level visibility.

How do we access the names exported by a module? Given a module m that exports an entity names x, common syntax for accessing x is $m.x$. Many languages also provide a mechanism to use all exported names of a module using shorter notation—e.g., "Open m", or "import m", or "using m".

3 Existential types

In this section, we will 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 isomorphism 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 intuitionistic logic, what would it mean for the formula "there exists some X such that ϕ is true" to be true? Recall that a formula 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.

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

Note that in this grammar, we annotate existential values with their existential type. The construct to create an existential value, $\mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2$, 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 an example to get an intuition for packing and unpacking.

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

$$
\begin{aligned}
&\mathsf{let}\ counterADT = \\
&\qquad \mathsf{pack}\ \{\mathbf{int}, \{\ \mathsf{new} = 0, \mathsf{get} = \lambda i{:}\mathbf{int}.\ i, \mathsf{inc} = \lambda i{:}\mathbf{int}.\ i + 1\ \}\ \} \\
&\qquad \mathsf{as}\ \exists \mathbf{Counter}.\ \{\ \mathsf{new} : \mathbf{Counter}, \mathsf{get} : \mathbf{Counter} \to \mathbf{int}, \mathsf{inc} : \mathbf{Counter} \to \mathbf{Counter}\ \} \\
&\mathsf{in}\ \ldots
\end{aligned}
$$

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

can use the existential value $counterADT$ as follows.

$$\begin{array}{l}\mathsf{unpack}\ \{C, c\} = counterADT\ \mathsf{in}\\ \mathsf{let}\ y = c.\mathsf{new}\ \mathsf{in}\\ c.\mathsf{get}\ (c.\mathsf{inc}\ (c.\mathsf{inc}\ y))\end{array}$$

Note that we annotate the pack construct with the existential type. That is, we explicitly state the type

$$\exists\textbf{Counter}.\ \{\mathsf{new}:\textbf{Counter}, \mathsf{get}:\textbf{Counter}\to\textbf{int}, \mathsf{inc}:\textbf{Counter}\to\textbf{Counter}\}.$$

Why do we do 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 for existentials. We add two new evaluation contexts, and one evaluation rule for unpacking an existential value.

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

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

The typing rules ensure that existential values are used correctly.

$$\frac{\Delta, \Gamma \vdash e : \tau_2\{\tau_1/X\} \quad \Delta \vdash \exists X.\ \tau_2\ \mathsf{ok}}{\Delta, \Gamma \vdash \mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2 : \exists X.\ \tau_2}$$

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

Note that in the typing rule for unpack, the side condition $\Delta \vdash \tau_2\ \mathsf{ok}$ ensures that the existentially quantified type variable X does *not* appear free in τ_2. This rules out programs such as,

$$\begin{array}{l}\mathsf{let}\ m =\\ \quad \mathsf{pack}\ \{\textbf{int}, \{a = 5, f = \lambda x:\textbf{int}.x + 1\}\}\ \mathsf{as}\ \exists X.\ \{a:X, f:X\to X\}\\ \mathsf{in}\\ \mathsf{unpack}\ \{X, x\} = m\ \mathsf{in}\ x.f\ x.a\end{array}$$

where the type of $(f.x\ x.a)$ has X free.