Curry-Howard Isomorphism

Lecture 17 Tuesday, April 2, 2013

1 Curry-Howard Isomorphism

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

Intuitionistic 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 intuitionistic logic and classical logic is that in intuitionistic 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 intuitionistic 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 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!

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 isomorphism, 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 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.

Invalidity = uninhabited type So under the Curry-Howard isomorphism, 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 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.

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

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

1.2 Negation and continuations

In intuitionistic 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**. $\ldots$, where k is a continuation, 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 isomorphism, this type corresponds to $(\tau \Rightarrow$ **False**$) \Rightarrow$ **False**, or, equivalently, $\neg(\neg\tau)$, the double negation 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!

2 Insights for PL from Logic, and Vice-Versa

The Curry-Howard isomorphism gives a connection between logic and types. This connection goes both ways: we can use insights from logic to think about programming languages, and develop new language features that correspond to logical entities; and we can use insights from programming languages influence our study and use of logic?

We consider two examples: how substructural logics give rise to substructural type systems, and how type theory can help us prove theorems.

2.1 Substructural Type Systems

2.1.1 Natural deduction and structural inference rules

Natural deduction is a kind of proof calculus that can be used to formalize mathematical logic. It's called "natural deduction" because it is meant to correspond to a "natural" way of reasoning about truth. In natural deduction, we write

$$A_1, \ldots, A_n \vdash B$$

to mean that whenever formulas A_1 through A_n are true, then formula B is true. For example, in a propositional calculus, we may write the judgment

$$p, \neg q \vdash q \Rightarrow p,$$

which intuitively means that if proposition p is true, and formula $\neg q$ is true, then the formula $q \Rightarrow p$ is true.

Like we do in programming languages, we can write inference rules and axioms for a given natural deduction calculus, that define which judgments are true. For example, we may have an axiom

$$\phi, \psi \vdash \phi \wedge \psi$$

That is, if formulas ϕ and ψ are true, then the conjunction $\phi \wedge \psi$ is true.

The inference rules that are concerned with the manipulation of the assumptions (i.e., the formulas on the left of the turnstile "$\vdash$") are known as the *structural inference rules*. The structural inference rules allow us to treat the assumptions like a set. That is, there are inference rules to re-order the assumptions, to collapse identical assumptions, and to remove unneeded assumptions. We use Γ and Δ to range over (possibly empty) sequences of formulas.

$$\textsc{Exchange}\ \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C} \qquad \textsc{Contraction}\ \frac{\Gamma, A, A, \Delta \vdash B}{\Gamma, A, \Delta \vdash B} \qquad \textsc{Weakening}\ \frac{\Gamma, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}$$

There are, of course, other inference rules, depending on the logic. Here are some of the inference rules for a propositional logic. Note that the assumptions in the conclusion of the inference rules are conserved in the premises, e.g., there is no duplication or dropping of assumptions.

$$\frac{}{A \vdash A} \qquad \frac{\Gamma, B \vdash A}{\Gamma \vdash B \Rightarrow A} \qquad \frac{\Gamma \vdash B \Rightarrow A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

If we drop any of the structural inference rules from the definition of our logic, we have a *substructural logic*.

For example, if we allow Exchange but drop Weakening and Contraction we have a *linear logic*: every assumption must be used exactly once. If we allow both Exchange and Weakening, but drop Contraction, we have an *affine logic*: every assumption may be used at most once.

2.1.2 Substructural type systems

So, what new programming language features or designs do substructural logic give us? You may have noticed the similarity between the natural deduction judgments for logic and the type judgments we use in programming languages. A type judgment looks like

$$\Gamma \vdash e : \tau$$

where we can think of the type context Γ as being a sequence $x_1 : \tau_1, \ldots, x_n : \tau_n$. Inference rules for such a type system would need to have rules for manipulating the type context.

$$\textsc{Exchange}\ \frac{\Gamma, x:\tau_1, y:\tau_2, \Delta \vdash e:\tau}{\Gamma, y:\tau_2, x:\tau_1, \Delta \vdash e:\tau} \qquad \textsc{Contraction}\ \frac{\Gamma, x:\tau, x:\tau, \Delta \vdash e:\tau'}{\Gamma, x:\tau, \Delta \vdash e:\tau'}$$

$$\textsc{Weakening}\ \frac{\Gamma, \Delta \vdash e:\tau}{\Gamma, x:\tau', \Delta \vdash e:\tau}\ x \text{ not in } \Gamma, \Delta$$

If we drop any of these structural inference rules, we have a *substructural type system*.

Linear type systems So if we drop Contraction and Weakening, we have a *linear type system*, by analogy with a linear logic. So how does dropping Contraction and Weakening affect the set of programs that will be well typed? Well, variables placed into the typing context must be used exactly once along any control flow path. The rule Contraction would allow us to use a variable multiple times, and Weakening allows us to not use a variable at all. This makes linear type systems good for tracking the use of resources. For example, we can use a linear type system to track open file handles, and ensure that a client must always close a file (i.e., must use the file handle at least once), and cannot close a file multiple times (i.e., must use the file handle at most once). In a similar way, we can use a linear type system to track objects allocated on the heap, and (with some additional language support) ensure that we have always exactly one pointer to a heap object. This can be useful for reasoning about aliasing: if we maintain an invariant that each heap object has exactly one pointer, then if a function is given two pointers, it knows they must not alias.

To use linear type systems in practice, it is often necessary to allow both linear types, and non-linear types (i.e., variables that can contraction and weakening applied to them), and/or to allow linearity to be weakened locally.

Ordered type systems Think about a type system where we drop all three structural rules (Exchange, Contraction, and Weakening). This is known as an *ordered type system*. Every variable is used exactly once, in the order it was introduced. In the same way that we can use a linear type system to help us reason about heap-allocated memory, we can use an ordered type system to help us reason about stack-allocated memory: not only must we use (i.e., deallocate) every piece of memory exactly once, but we must do so in stack order: i.e., the most recently allocated memory on the stack must be deallocated first.

2.2 Theorem Proving

The Curry-Howard isomorphism 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.

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

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

Now, let's use the Curry-Howard isomorphism, 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 blue, 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 $(p \Rightarrow (q \Rightarrow r)) \quad \Rightarrow \quad (p \wedge q) \Rightarrow r$.

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.