Type inference; Curry-Howard isomorphism

Lecture 19 Tuesday, April 6, 2010

1 Type inference

In the simply typed lambda calculus, we must explicitly state the type of function arguments: $\lambda x\!:\!\tau.\, e$. This explicitness makes it possible to type check functions.

$$\frac{\Gamma, x\!:\!\tau \vdash e\!:\!\tau'}{\Gamma \vdash \lambda x\!:\!\tau.\, e\!:\!\tau \to \tau'}$$

Suppose we didn't want to provide type annotations for function arguments. Consider the typing rule for functions without type annotations.

$$\frac{\Gamma, x\!:\!\tau \vdash e\!:\!\tau'}{\Gamma \vdash \lambda x.\, e\!:\!\tau \to \tau'}$$

The type-checking algorithm would need to guess or somehow know what type τ to put into the type context.

Can we still type check our program without these type annotations? For the simply typed lambda calculus (and many of the extensions we have considered so far), the answer is yes: we can *infer* or *reconstruct* the types of a program.

Let's consider an example to see how this type inference could work.

$$\lambda a.\, \lambda b.\, \lambda c.\, \mathsf{if}\ a\ (b+1)\ \mathsf{then}\ b\ \mathsf{else}\ c$$

Since the variable b is used in an addition, the type of b must be **int**. The variable a must be some kind of function, since it is applied to the expression $b+1$. Since a has a function type, the type of the expression $b+1$ (i.e., **int**) must be a's argument type. Moreover, the result of the function application ($a\ (b+1)$) is used as the test of a conditional, so it better be the case that the result type of a is also **bool**. So the type of a should be **int** $\to$ **bool**. Both branches of a conditional should return values of the same type, so the type of c must be the same as the type of b, namely **int**.

We can write the expression with the reconstructed types:

$$\lambda a\!:\!\textbf{int} \to \textbf{bool}.\, \lambda b\!:\!\textbf{int}.\, \lambda c\!:\!\textbf{int}.\, \mathsf{if}\ a\ (b+1)\ \mathsf{then}\ b\ \mathsf{else}\ c$$

1.1 Constraint-based typing

We now present an algorithm that, when given a typing context Γ and an expression e, produces a set of *constraints*—equations between types (including type variables)—that must be satisfied in order for e to be well-typed in Γ.

We first introduce *type variables*, which are just placeholders for types. We use X and Y to range over type variables.

The language we will consider is the lambda calculus with integer constants and addition. We assume that all function definitions contain a type annotation for the argument, but this type may simply be a type variable X.

$$\begin{aligned} e &::= x \mid \lambda x\!:\!\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\ \tau &::= \textbf{int} \mid X \mid \tau_1 \to \tau_2 \end{aligned}$$

To formally define type inference, we introduce a new typing relation:

$$\Gamma \vdash e : \tau \mid C$$

Intuitively, if $\Gamma \vdash e : \tau \mid C$, then expression e has type τ provided that every constraint in the set C is satisfied.

We define the judgment $\Gamma \vdash e : \tau \mid C$ with inference rules and axioms. When read from bottom to top, these inference rules provide a procedure that, given Γ and e, calculates τ and C such that $\Gamma \vdash e : \tau \mid C$.

$$\text{CT-VAR}\ \frac{}{\Gamma \vdash x : \tau \mid \emptyset}\ x : \tau \in \Gamma \qquad\qquad \text{CT-INT}\ \frac{}{\Gamma \vdash n : \textbf{int} \mid \emptyset}$$

$$\text{CT-ADD}\ \frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : \textbf{int} \mid C_1 \cup C_2 \cup \{\tau_1 = \textbf{int}, \tau_2 = \textbf{int}\}}$$

$$\text{CT-ABS}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \mid C}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2 \mid C} \qquad \text{CT-APP}\ \frac{\Gamma \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \vdash e_2 : \tau_2 \mid C_2 \quad C' = C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to X\}}{\Gamma \vdash e_1\ e_2 : X \mid C'}\ X \text{ is fresh}$$

Note that we must be careful with the choice of fresh type variables. We have omitted some of the technical details that ensure the fresh type variables in the rule CT-APP are chosen appropriately.

1.2 Unification

So what does it mean for a set of constraints to be satisfied? To answer this question, we define *type substitutions* (or just *substitutions*, when it's clear from context).

1.2.1 Type substitution

A type substitution is a finite map from type variables to types. For example, we write $[X \mapsto \textbf{int}, Y \mapsto \textbf{int} \to \textbf{int}]$ for the substitution that maps type variable X to **int**, and type variable Y to **int** $\to$ **int**.

Note that the same variable could occur in both the domain and range of a substitution. In that case, the intention is that all substitutions are performed simultaneously. For example the substitution $[X \mapsto \textbf{int}, Y \mapsto \textbf{int} \to X]$ mays Y to **int** $\to$ **int**.

More formally, we define substitution of type variables as follows.

$$\sigma(X) = \begin{cases} \tau & \text{if } X \mapsto \tau \in \sigma \\ X & \text{if } X \text{ not in the domain of } \sigma \end{cases}$$
$$\sigma(\textbf{int}) = \textbf{int}$$
$$\sigma(\tau \to \tau') = \sigma(\tau) \to \sigma(\tau')$$

Note that we don't need to worry about avoiding variable capture, since there are no constructs in the language that bind type variables. If we had polymorphic types $\forall X.\ \tau$ from the polymorphic lambda calculus, we would need to be concerned with this.

Given two substitutions σ and σ', we write $\sigma \circ \sigma'$ for the composition of the substitutions: $\sigma \circ \sigma'(\tau) = \sigma(\sigma'(\tau))$.

1.2.2 Unification

Constraints are of the form $\tau = \tau'$. We say that a substitution σ *unifies* constraint $\tau = \tau'$ if $\sigma(\tau) = \sigma(\tau')$. We say that substitution σ *satisfies* (or *unifies*) set of constraints C if σ unifies every constraint in C.

For example, the substitution $\sigma = [X \mapsto \textbf{int}, Y \mapsto \textbf{int} \to \textbf{int}]$ unifies the constraint $X \to (X \to \textbf{int}) = \textbf{int} \to Y$, since

$$\sigma(X \to (X \to \textbf{int})) \quad = \quad \textbf{int} \to (\textbf{int} \to \textbf{int}) \quad = \quad \sigma(\textbf{int} \to Y)$$

So to solve a set of constraints C, we need to find a substitution that unifies C. More specifically, suppose that $\Gamma \vdash e : \tau \mid C$; a solution for (Γ, e, τ, C) is a pair $\sigma, \tau')$ such that σ satisfies C and $\sigma(\tau) = \tau'$. If there are no substitutions that satisfy C, then we know that e is not typable.

1.2.3 Unification algorithm

To calculate solutions to constraint sets, we use the idea, due to Hindley and Milner, or using *unification* to check that the set of solutions is non-empty, and to find a "best" solution (from which all other solutions can be easily generated).

The algorithm for unification is defined as follows.

$$\begin{aligned}
\mathit{unify}(\emptyset) &= [] \qquad \text{(the empty substitution)} \\
\mathit{unify}(\{\tau = \tau'\} \cup C') &= \text{if } \tau = \tau' \text{ then} \\
&\qquad \mathit{unify}(C') \\
&\quad \text{else if } \tau = X \text{ and } X \text{ not a free variable of } \tau' \text{ then} \\
&\qquad \mathit{unify}(C'\{\tau'/X\}) \circ [X \mapsto \tau'] \\
&\quad \text{else if } \tau' = X \text{ and } X \text{ not a free variable of } \tau \text{ then} \\
&\qquad \mathit{unify}(C'\{\tau/X\}) \circ [X \mapsto \tau] \\
&\quad \text{else if } \tau = \tau_o \to \tau_1 \text{ and } \tau' = \tau'_o \to \tau'_1 \text{ then} \\
&\qquad \mathit{unify}(C' \cup \{\tau_0 = \tau'_0, \tau_1 = \tau'_1\}) \\
&\quad \text{else} \\
&\qquad \textit{fail}
\end{aligned}$$

The check that X is not a free variable of the other type ensures that the algorithm doesn't produce a cyclic substitution (e.g., $X \mapsto X \to X$), which doesn't make sense with the finite types that we currently have.

The unification algorithm always terminates. (How would you go about proving this?) Moreover, it produces a solution if and only if a solution exists. The solution found is the most general solution, in the sense that if $\sigma = \mathit{unify}(C)$ and σ' is a solution to C, then there is some σ'' such that $\sigma' = \sigma \circ \sigma''$.

2 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 true 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.

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!

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!

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.

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, 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.

2.1 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.

2.2 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!