Dependent types

Lecture 18 Thursday, April 1, 2010

Guest Lecturer: Jeff Vaughan

1 Dependent types

Suppose we introduce immutable vectors of booleans to the lambda calculus. A vector $\langle v_1, \ldots, v_n \rangle$ is of length n, and eachv_i is a boolean value. Let's add primitive function init as a way to construct vectors. It will take a natural number and a value as arguments and produce a vector: init $k\ v$ will evaluate to $\langle v, \ldots, v \rangle$, a vector of length k where each element has the value v.

Similarly, we'll add a primitive function index to access an element of the vector: index $i\ \langle v_1, \ldots, v_n \rangle$ will evaluate to v_i, provided $1 \leq i \leq n$. Finally, we'll add a primitive function asPairs that takes a vector, and returns a representation of the vector using pairs: asPairs $\langle v_1, \ldots, v_n \rangle$ evaluates to $(v_1, (v_2, \ldots (v_n, ()) \ldots))$.

(We could add a way to produce vectors that contain different values, but for our purposes just init and index are sufficient.)

We'll also include natural numbers n, pairs, and the unit value in the language. The syntax of expressions and values for the extended language is the following.

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid n \mid (e_1, e_2) \mid () \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{init} \mid \mathsf{index} \mid \mathsf{asPairs}$$
$$v ::= \lambda x.\, e \mid n \mid \langle v_1, \ldots, v_n \rangle \mid (v_1, v_2) \mid () \mid \mathsf{true} \mid \mathsf{false}$$

Evaluation rules for the new constructs are defined as follows.

$$\frac{}{\mathsf{init}\ k\ v \longrightarrow \langle v_1, \ldots, v_k \rangle}\ \forall i \in 1..k.\ v_i = v \qquad \frac{}{\mathsf{index}\ i\ \langle v_1, \ldots, v_k \rangle \longrightarrow v_i}$$

$$\frac{}{\mathsf{asPairs}\ \langle v_1, \ldots, v_k \rangle \longrightarrow (v_1, (v_2, \ldots (v_n, ()) \ldots))}$$

Note that asPairs applied to a vector with a single element $\langle v_1 \rangle$ evaluates to $(v_1, ())$ and asPairs applied to the empty vector $\langle\rangle$ evaluates to $()$.

We could run in to some problems when executing programs in our new language. We could try to apply the new primitive functions to values of the wrong type (e.g., asPairs 42). But we might try to access a vector with an inappropriate index, for example index $7\ \langle 3, 2, 1 \rangle$.

The first problem we can avoid by using type systems like we have seen previously: give vectors a type, say **boolvec**, and give the primitive function asPairs the type **boolvec** $\rightarrow$ **nat** $\rightarrow$ **bool** (where **nat** is the type of natural numbers).

But this approach would mean that we cannot provide a type for the primitive function asPairs, which takes a **boolvec** and returns nested pairs. The type of the pair returned depends on the length of the vector!

Also, this approach does not stop us from having incorrect indices, as it would still allow the expression index $7\ \langle 3, 2, 1 \rangle$.

1.1 First attempt at a type system

To address these problems, we are going to use a *dependent type* for boolean vectors, where the length of the vector is part of the type of a vector. The type **boolvec** e represents boolean vectors of length e, where e is a natural number expression.

The signature for init becomes $(n : \textbf{nat}) \rightarrow \textbf{bool} \rightarrow \textbf{boolvec}\ n$. That is, init takes two arguments, a natural number n, and a boolean, and produces a boolean vector of length n, a **boolvec** n.

With this new type, we define typing rules for vectors are the following.

$$\frac{\forall i \in 1..n. \quad \Gamma \vdash v_i : \textbf{bool}}{\Gamma \vdash \langle v_1, \ldots, v_n \rangle : \textbf{boolvec}\ n} \qquad \frac{\Gamma \vdash e_1 : \textbf{nat} \quad \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash \mathsf{init}\ e_1\ e_2 : \textbf{boolvec}\ e_1} \qquad \frac{\Gamma \vdash e_1 : \textbf{nat} \quad \Gamma \vdash e_2 : \textbf{boolvec}\ e_3}{\Gamma \vdash \mathsf{index}\ e_1\ e_2 : \textbf{bool}}\ e_1 \leq e_3$$

The type of a vector of length n is simply **boolvec** n. The first argument to the primitive function to create vectors, init, is the length of the new vector, and so the type of init $e_1\ e_2$ is **boolvec** e_1, a vector with length e_1. The typing rule for index requires that the index e_1 is no greater than the length of the vector being accessed: $e_1 \leq e_3$.

Now, the type **boolvec** e is very strange! We have at least three problems with this newly proposed type **boolvec** e.

1. In the type for init, $(n : \textbf{nat}) \to \textbf{bool} \to \textbf{boolvec}\ n$, the first argument is somehow bound to the variable n which occurs in the return type of the function. What does this mean?

2. The type contains an expression e. This could be an arbitrary expression. If e is a literal natural number, then the type maybe makes sense, for example **boolvec** 7 is the type of vectors of length 7. But what do the types **boolvec** $(9 + 1)$ or **boolvec** x mean? And what does it mean in the proposed typing rule for index to have a side condition $e_1 \leq e_3$?

3. The expression e in the type **boolvec** e should be of type **nat**. For example, we shouldn't let someone write code where e is not of type **nat**, such as $\lambda x : \textbf{boolvec unit}. \ldots$. How do we ensure that e is limited to expressions of type **nat**.

1.2 LF

We address some of these problems by considering boolean vectors in the language LF, which stands for Logical Framework.

We give expressions types to allow us to restrict and reason about the use of expressions. Types can be thought of as describing sets of expressions. In LF, just as expressions have types, types have *kinds*. That is, kinds describe sets of types, and we use kinds to restrict and reason about the use of types.

The syntax of LF is given by the following grammar. Here, we use metavariable K to range over kinds.

Expressions	$e ::= x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid n \mid \langle v_1, \ldots, v_n \rangle \mid \ldots$
Types	$\tau ::= \textbf{nat} \mid \textbf{boolvec} \mid \textbf{bool} \mid \textbf{unit} \mid \tau\ e \mid (x{:}\tau_1) \to \tau_2$
Kinds	$K ::= \textbf{Type} \mid (x{:}\tau) \Rightarrow K$

In LF, the literal n has type **nat**, just as in the simply-typed lambda calculus. Also, the type of a vector $\langle v_1, \ldots, v_n \rangle$ is **boolvec** n. However, we will prevent the use of ill-formed types such as **boolvec** () by using kinds to restrict how types may be composed. The kind of **boolvec** will be $(x : \textbf{nat}) \Rightarrow \textbf{Type}$: it takes a natural number and produces a type.

We will have three judgments, one each for expressions, types, and kinds. The form of the judgment for expressions is $\Gamma \vdash e{:}\tau$, and means that under context Γ, expression e has type τ. We extend contexts so that they include both the types of program variables $(x{:}\tau)$ and the kinds of type variables $(X :: K)$.

The form of the judgment for types is $\Gamma \vdash \tau :: K$, meaning that under context Γ, type τ has kind K. In LF, kinds do not have their "types" (i.e., there is no entity that describes sets of kinds), so the judgment for kinds is of the form $\Gamma \vdash K$ ok, which means that under context Γ, kind K is well-formed.

Since types may contain expressions, in order to perform type checking, we may need to evaluate expressions to determine if two types are equivalent, for example the types **boolvec** 19 and **boolvec** $(12 + 7)$. Similarly, since kinds may contain types (in the production $(x : \tau) \Rightarrow K$), we may need to evaluate expressions when checking kinds. As such, we also define another three relations that describe equivalence between expressions, types, and kinds, respectively. We denote all three relations with the symbol $\equiv$.

Types are similar to what we have seen previously. We have primitive types (**nat**, **bool**, **boolvec**, etc.). The function type gives the argument type a binder: $(x{:}\tau_1) \to \tau_2$ allows the program variable x to appear in the type τ_2, where x represents the argument given to the expression. We also have type application $\tau\ e$,

which applies a type to an expression. The key example of this in our system is the application of **boolvec** to an expression, such as **boolvec** 7.

The rules for the judgment $\Gamma \vdash e : \tau$ are the following.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash T :: K}{\Gamma \vdash x : T}\, x : T \in \Gamma \qquad \frac{}{\Gamma \vdash n : \textbf{nat}}\, n \in \mathbb{N} \qquad \frac{}{\Gamma \vdash \langle v_1, \ldots, v_n \rangle : \textbf{boolvec}\ n}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau' \quad \Gamma \vdash (x : \tau) \to \tau' :: \textbf{Type}}{\Gamma \vdash \lambda x : \tau.\, e : (x : \tau) \to \tau'} \qquad \frac{\Gamma \vdash e_1 : (x : \tau') \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau\{e_2/x\}}$$

$$\textsc{Conversion}\ \frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash \tau :: K \quad \tau \equiv \tau'}{\Gamma \vdash e : \tau}$$

The rule for conversion uses the relation $\equiv$ that we will define later. We also need rules for base values, such as $()$, true, false. We omit these rules.

Note that we need to provide types for our primitive functions init, index and asPairs. We can simply assume that these primitive functions are variables that are bound in every context.

The judgment $\Gamma \vdash \tau :: K$ is given by the following rules and axioms.

$\boxed{\Gamma \vdash \tau :: K}$

$$\frac{\Gamma \vdash K\ \mathsf{ok}}{\Gamma \vdash X :: K}\, X : K \in \Gamma \qquad \frac{\Gamma \vdash \tau :: \textbf{Type} \quad \Gamma, x : \tau \vdash \tau' :: \textbf{Type}}{\Gamma \vdash (x : \tau) \to \tau' :: \textbf{Type}} \qquad \frac{\Gamma \vdash \tau :: (x : \tau') \Rightarrow K \quad \Gamma \vdash e : \tau'}{\Gamma \vdash \tau\ e :: K\{e/x\}}$$

$$\textsc{Conversion}\ \frac{\Gamma \vdash \tau :: K' \quad \Gamma \vdash K\ \mathsf{ok} \quad K \equiv K'}{\Gamma \vdash \tau :: K}$$

Note that we can just assume that the primitive types **unit**, **bool**, **nat**, and **boolvec**, are just type variables that appear in the context. We assume that the kind of **unit**, **bool**, and **nat** is **Type**, and that the kind of **boolvec** is $(n : \textbf{nat}) \Rightarrow \textbf{Type}$, that is, it takes an expression of type **nat**, and produces a type.

Note also that we restrict the kinds of types used in function type $(x : \tau) \to \tau'$: both τ and τ' are restricted to **Type**s, and cannot be arbitrary kinds, such as $\textbf{Type} \Rightarrow \textbf{Type}$. We do this because lambda abstractions can be thought of as machines that take an expression as input, and produce an expression as output; the kind of the type of any expression is **Type**.

The judgment $\Gamma \vdash K\ \mathsf{ok}$ simply ensures that variables in kinds are used correctly.

$\boxed{\Gamma \vdash K\ \mathsf{ok}}$

$$\frac{}{\Gamma \vdash \textbf{Type}\ \mathsf{ok}} \qquad \frac{\Gamma \vdash \tau :: K' \quad \Gamma, x : \tau \vdash K\ \mathsf{ok}}{\Gamma \vdash (x : \tau) \Rightarrow K\ \mathsf{ok}}$$

The conversion relations $\equiv$ define what it means for expressions, types, and kinds to be equivalent. In a nutshell, it is if they evaluate to the same value. We define a special evaluation relation $\twoheadrightarrow$ that allows the evaluation of expressions with free variables, and allows the evaluation of expressions that occur within types.

$$\frac{e_1 \twoheadrightarrow^* e \quad e_2 \twoheadrightarrow^* e}{e_1 \equiv e_2} \qquad \frac{\tau_1 \twoheadrightarrow^* \tau \quad \tau_2 \twoheadrightarrow^* \tau}{\tau_1 \equiv \tau_2} \qquad \frac{K_1 \twoheadrightarrow^* K \quad K_2 \twoheadrightarrow^* K}{K_1 \equiv K_2}$$

We use several evaluation contexts to define the evaluation relation.

$$\begin{aligned}\mathcal{E}x &::= [\cdot]_{exp} \mid \mathcal{E}\ e \mid e\ \mathcal{E} \mid \lambda x{:}\mathcal{T}.\, e \mid \lambda x{:}\tau.\, \mathcal{E}\\ \mathcal{T} &::= [\cdot]_{type} \mid \tau\ \mathcal{E} \mid (x{:}\mathcal{T}) \to \tau \mid (x{:}\tau) \to \mathcal{T}\\ \mathcal{K} &::= (x{:}\mathcal{T}) \Rightarrow K\end{aligned}$$

$$\frac{e \twoheadrightarrow e'}{\mathcal{E}[e] \twoheadrightarrow \mathcal{E}[e']} \qquad \frac{\tau \twoheadrightarrow \tau'}{\mathcal{E}[\tau] \twoheadrightarrow \mathcal{E}[\tau']} \qquad \frac{}{(\lambda x{:}\tau.\, e_1)\ e_2 \twoheadrightarrow e_1\{e_2/x\}}$$

$$\frac{\tau \twoheadrightarrow \tau'}{\mathcal{T}[\tau] \twoheadrightarrow \mathcal{T}[\tau']} \qquad \frac{e \twoheadrightarrow e'}{\mathcal{T}[e] \twoheadrightarrow \mathcal{T}[e']}$$

$$\frac{K \twoheadrightarrow K'}{\mathcal{K}[K] \twoheadrightarrow \mathcal{K}[K']}$$

Note that other than the evaluation context rules, the only rule that performs computation is the β-reduction rule for expressions.

So, what has LF gained us? The use of kinds ensure that we cannot mis-use a type constructor. For example, it will rule out any program that contains **boolvec** e where e does not have type **nat**.

LF also provides us with a clearly defined meaning for binding program variables in types. The type we gave for init now makes sense: $(n : \textbf{nat}) \to \textbf{bool} \to \textbf{boolvec}\ n$.

But, LF does not give us a way to write down an appropriate type for asPairs, and it cannot give a precise type for the primitive function index.

We can address these remaining issues by consider the *calculus of constructions* (CoC). The calculus of constructions is the language behind the proof assistant Coq.

We have seen functions from expressions to expressions (which are just the standard abstractions, $\lambda x.\, e$); polymorphic lambda calculus gave us functions from types to terms ($\Lambda X.\, e$); dependent types are functions from expressions to types (e.g., **boolvec** is a function from expressions of type **int** to types). To express the type of asPairs, we need functions from types to types. That is, the type of asPairs e depends on the type of e. CoC gives us

Due to time constraints, we did not get to discuss CoC in detail. If you are interested, a good introduction can be found in Chapter 2 of *Advanced Topics in Types and Programming Languages*, edited by Benjamin C. Pierce, MIT Press, 2005.