Dependent types

Lecture 20 Thursday, April 6, 2021

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 each v_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 $\langle v_1, \ldots, v_n \rangle\ i$ will evaluate to v_{i+1}, provided $0 \leq i < n$. (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.

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

Evaluation rules for the new constructs are defined as follows.

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

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., init $42\ 42$). But we might try to access a vector with an inappropriate index, for example index $\langle \mathsf{false}, \mathsf{false}, \mathsf{true} \rangle\ 7$.

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 index the type **nat** $\rightarrow$ **bool** $\rightarrow$ **boolvec** (where **nat** is the type of natural numbers). But this approach does not stop us from having incorrect indices, as it would still allow the expression index $\langle \mathsf{false}, \mathsf{false}, \mathsf{true} \rangle\ 7$.

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.

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

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_2 is no greater than the length of the vector being accessed: $e_2 \leq e_3$.

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 $\lambda x{:}\textbf{boolvec ()}. \ldots$. 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	$e ::= x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_1 \mid \langle v_1, \ldots, v_n \rangle \mid \ldots$
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** true 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. Relation $\Gamma \vdash e_1 \equiv e_2 : \tau$ means that (under context Γ), expressions e_1 and e_2 are equivalent (and have type τ). Relation $\Gamma \vdash \tau_1 \equiv \tau_2 :: K$ means that (under context Γ), types τ_1 and τ_2 are equivalent (and have kind K). Finally, relation $\Gamma \vdash K_1 \equiv K_2$ means that (under context Γ), kinds K_1 and K_2 are equivalent.

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.

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

$$\frac{\Gamma \vdash \tau :: K}{\Gamma \vdash x{:}\tau}\ x{:}\tau \in \Gamma \qquad \frac{}{\Gamma \vdash n{:}\textbf{nat}}\ n \in \mathbb{N} \qquad \frac{\Gamma \vdash e_1{:}\textbf{nat} \quad \Gamma \vdash e_2{:}\textbf{nat}}{\Gamma \vdash e_1 + e_2{:}\textbf{nat}} \qquad \frac{\text{For all } i \in 1..n. \quad \Gamma \vdash v_i{:}\textbf{bool}}{\Gamma \vdash \langle v_1, \ldots, v_n \rangle{:}\textbf{boolvec}\ n}$$

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

$$\textsc{Conversion}\ \frac{\Gamma \vdash e:\tau' \qquad \Gamma \vdash \tau \equiv \tau' :: \textbf{Type}}{\Gamma \vdash e:\tau}$$

The rule for conversion uses the relation $\Gamma \vdash \tau_1 \equiv \tau_2 :: K$ 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 and index. We can simply assume that these primitive functions are variables that are bound in every context. We will consider types for these primitive functions later.

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

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

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

$$\frac{\Gamma \vdash \tau :: (x:\tau') \Rightarrow K \qquad \Gamma \vdash e:\tau'}{\Gamma \vdash \tau\ e :: K\{e/x\}} \qquad\qquad \textsc{Conversion}\ \frac{\Gamma \vdash \tau :: K' \qquad \Gamma \vdash K \equiv K'}{\Gamma \vdash \tau :: K}$$

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 $(x:\textbf{bool}) \Rightarrow \textbf{Type}$. 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.

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

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

The equivalence 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. The question of what equivalences to include in a dependent type system is a critical one. Many different choices have been studied, leading to very different results. Clearly we would like to, for example. consider the types **boolvec** 42 and **boolvec** $(35+7)$ equivalent. But what about if we are in a context where we have variables x and f of type **nat** and **nat** $\to$ **nat**, respectively, where we know that $f\ x = 7$? Should we consider the types **boolvec** $(f\ x)$ and **boolvec** 7 to be equivalent?

$\boxed{\Gamma \vdash e_1 \equiv e_2 : \tau}$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \textbf{Type} \qquad \Gamma, x:\tau_1 \vdash e_1 \equiv e_2 : \tau}{\Gamma \vdash \lambda x:\tau_1.\, e_1 \equiv \lambda x:\tau_2.\, e_2 : (x:\tau_1) \to \tau} \qquad \frac{\Gamma \vdash e_1 \equiv e_2 : (x:\tau) \to \tau' \qquad \Gamma \vdash e_1' \equiv e_2' : \tau}{\Gamma \vdash e_1\ e_1' \equiv e_2\ e_2' : \tau'\{e_1'/x\}}$$

$$\frac{\Gamma, x:\tau \vdash e : \tau' \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash (\lambda x:\tau.\, e)\ e' \equiv e\{e'/x\} : \tau'\{e'/x\}} \qquad \frac{\Gamma \vdash e : (x:\tau) \to \tau' \qquad x \notin FV(e)}{\Gamma \vdash (\lambda x:\tau.\, e\ x) \equiv e : (x:\tau) \to \tau'}$$

$$\frac{\Gamma \vdash e_1 \equiv e_2 : \textbf{nat} \quad \Gamma \vdash e_1' \equiv e_2' : \textbf{nat}}{\Gamma \vdash e_1 + e_1' \equiv e_2 + e_2' : \textbf{nat}} \qquad \frac{}{\Gamma \vdash k + m \equiv n : \textbf{nat}}\ n \text{ is the sum of } k \text{ and } m$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \equiv e : \tau} \qquad \frac{\Gamma \vdash e_1 \equiv e_2 : \tau}{\Gamma \vdash e_2 \equiv e_1 : \tau} \qquad \frac{\Gamma \vdash e_1 \equiv e_2 : \tau \qquad \Gamma \vdash e_2 \equiv e_3 : \tau}{\Gamma \vdash e_1 \equiv e_3 : \tau}$$

$\boxed{\Gamma \vdash \tau_1 \equiv \tau_2 :: K}$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \textbf{Type} \qquad \Gamma, x:\tau_1 \vdash \tau_1' \equiv \tau_2' :: \textbf{Type}}{\Gamma \vdash (x:\tau_1) \to \tau_1' \equiv (x:\tau_2) \to \tau_2' :: \textbf{Type}} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: (x:\tau) \Rightarrow K \qquad \Gamma \vdash e_1 \equiv e_2 : \tau}{\Gamma \vdash \tau_1\ e_1 \equiv \tau_2\ e_2 :: K\{e_1/x\}}$$

$$\frac{\Gamma \vdash \tau :: K}{\Gamma \vdash \tau \equiv \tau :: K} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: K}{\Gamma \vdash \tau_2 \equiv \tau_1 :: K} \qquad \frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: K \qquad \Gamma \vdash \tau_2 \equiv \tau_3 :: K}{\Gamma \vdash \tau_1 \equiv \tau_3 :: K}$$

$\boxed{\Gamma \vdash K_1 \equiv K_2}$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2 :: \textbf{Type} \qquad \Gamma, x:\tau_1 \vdash K_1 \equiv K_2}{\Gamma \vdash (x:\tau_1) \Rightarrow K_1 \equiv (x:\tau_2) \Rightarrow K_2}$$

$$\frac{\Gamma \vdash K \text{ ok}}{\Gamma \vdash K \equiv K} \qquad \frac{\Gamma \vdash K_1 \equiv K_2}{\Gamma \vdash K_2 \equiv K_1} \qquad \frac{\Gamma \vdash K_1 \equiv K_2 \qquad \Gamma \vdash K_2 \equiv K_3}{\Gamma \vdash K_1 \equiv K_3}$$

So, what has LF gained us? The use of kinds ensures 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: $(n : \textbf{nat}) \to \textbf{bool} \to \textbf{boolvec}\ n$. We can also give precise types to other interesting functions, such as join, a function that takes two vectors on length n and k respectively, and combines them to form a vector of length $n + k$ could be given the following type:

$$(n : \textbf{nat}) \to (k : \textbf{nat}) \to \textbf{boolvec}\ n \to \textbf{boolvec}\ k \to \textbf{boolvec}\ (n + k)$$

What about a type for index? That is, how do we ensure that in expression index $e_1\ e_2$ that if e_1 has type **boolvec** n, how do we ensure that e_2 is a natural number that is less than n? And, suppose we wanted 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 $(v_1, (v_2, \ldots (v_n, ()) \ldots))$. LF does not provide us with a way to write down an appropriate type for asPairs.

We can address these remaining issues by considering 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 **nat** 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.

In part, the power of Coq comes from viewing dependent types through the Curry-Howard Isomorphism, whereby we can equate types with propositions, and expressions with proofs. With dependent types, expressions can appear in proofs. One of the ways that we can think about that, is that dependent types are allowing us to express properties (predicates) of values that the program manipulates. That is, one way to consider a dependent type $(x : \tau_1) \to \tau_2$ is that it corresponds to universal quantification in first-order logic: for any value (individual) x of type τ_1, proposition τ_2 (which mentions x) is true. A function with type $(x : \tau_1) \to \tau_2$ is a proof of this universal quantification, because if you give it any value v of type τ_1, then it will give back a proof that $\tau_2\{v/x\}$ holds.

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