2 *Dependent Types*

David Aspinall and Martin Hofmann

In the most general sense, dependent types are type-valued functions. This definition includes, for example, the type operators of F^{ω} such as `Pair`. When applied to two types `S` and `T`, this yields the type `Pair S T` whose elements are pairs `(s,t)` of elements `s` from `S` and `t` from `T` (see *TAPL*, Chapter 29). However, the terminology "dependent types" is usually used to indicate a particular class of type-valued functions: those functions which send *terms* to types. In this chapter we study this kind of dependency.

2.1 Motivations

We begin the chapter by looking at a few representative examples of where type-term dependency occurs naturally.

Programming with Vectors and Format Strings

The prototypical example of programming with dependent types is introduced by the *type family* of vectors (one-dimensional arrays):

```
Vector :: Nat → *
```

This kinding assertion states that `Vector` maps a natural number `k:Nat` to a type. The idea is that the type `Vector k` contains vectors of length `k` of elements of some fixed type, say `data`.

To use vectors, we need a way of introducing them. A useful initialization function takes a length `n`, a value `t` of type `data`, and returns a vector with `n`

The system studied in this chapter is the dependently typed lambda-calculus, λLF (Figures 2-1, 2-2), extended with Σ-types (Figure 2-5) and the Calculus of Constructions (Figure 2-7). The associated OCaml implementation, called `deptypes`, can be found on the book's web site.

elements all set to `t`. The typing of such an `init` function is written like this,

```
init : Πn:Nat. data → Vector n
```

and the application `init k t` has type `Vector k`.

The type of `init` introduces the *dependent product type* (or "Pi type"), written `Πx:S.T`. This type generalizes the arrow type of the simply typed lambda-calculus. It is the type of functions which map elements `s:S` to elements of `[x ↦ s]T`. In contrast to the simply-typed case, the result type of a function with a Π-type can vary according to the argument supplied. According to Seldin (2002), the Π-type goes back to Curry and is thus almost as old as the lambda calculus itself.

A more interesting way of building up vectors is given by the constant empty vector `empty : Vector 0` and a constructor for building longer vectors:

```
cons : Πn:Nat. data → Vector n → Vector (n+1).
```

The typing of `cons` expresses that `cons` takes three arguments: a natural number `n`, an element of type `data`, and a vector of length `n`. The result is a vector of length `n+1`. This means that, for example, if `v : Vector 5` and `x : data`, then `cons 5 x v : Vector 6`.

The dependent product type `Πx:S.T` is somewhat analogous to the universal type `∀X.T` of System F. The type of a term `t` with type `∀X.T` also varies with the argument supplied; but in the case of a type abstraction, the argument is a type rather than a term. If `A` is a type, then `t A:[X ↦ A]T`. In System F (and F^{ω}), type variation occurs only with type arguments, whereas in dependent type theory it may occur with term-level arguments.

The reader familiar with programming with ordinary arrays will have realized that by using a type of arrays instead of vectors we could avoid dependent typing. The initialization function for one-dimensional arrays could be given the simple type `Nat → data → Array`, where `Array` is the type of arrays with entries of type `data`. The point of the dependent typing is that it reveals more information about the behavior of a term, which can be exploited to give more precise typings and exclude more of the badly behaved terms in a type system. For example, with the dependent type of vectors, we can type a function that returns the first element of a non-empty vector:

```
first : Πn:Nat.Vector(n+1) → data
```

The function `first` can never be applied to an empty vector—non-emptiness is expressed within the type system itself! This is a useful gain. With ordinary arrays instead of dependently-typed vectors, we would need some special way to deal with the case when `first` is applied to the empty array. We could return an ad-hoc default element, or we might use a language-based exception mechanism to indicate the error. Either mechanism is more clumsy than simply prohibiting illegal applications of `first` from being written.

We suggested that Π`x:S.T` generalizes the function space `S→T` of simply typed lambda calculus. In fact, we can treat `S→T` simply as an abbreviation:

`S→T` = Π`x:S.T`   *where* `x` *does not appear free in* `T`

For example, Π`x:Nat.Nat` is exactly equivalent to `Nat → Nat`. We will continue to write the arrow → whenever possible, to increase readability.

Another favorite example of a function with a useful dependent typing is `sprintf` of the C language.[1] Recall that `sprintf` accepts a format string and list of arguments whose types must correspond to the declarations made in the format string. It then converts the given arguments into a string and returns it. A simplified form of `sprintf` might have the typing:

```
sprintf : Πf:Format. Data(f) → String
```

where we suppose that `Format` is a type of valid print formats (for example, considered as character lists) and that `Data(f)` is the type of data corresponding to format `f`. The function `Data(f)` evaluates the type that the format string describes, which might include clauses like these:

```
Data([])          =  Unit
Data("%d"::cs)    =  Nat * Data(cs)
Data("%s"::cs)    =  String * Data(cs)
Data(c::cs)       =  Data(cs)
```

This example is rather different to the case of vectors. Vectors are uniform: we introduce operations that are parametric in the length `n`, and the family of types `Vector n` is indexed by `n`. In the case of format strings, we use case analysis over values to construct the type `Data(f)` which depends on `f` in an arbitrary way. Unsurprisingly, this non-uniform kind of dependent type is more challenging to deal with in practical type systems for programming.

2.1.1 Exercise [★]: Suggest some dependent typings for familiar data types and their operations. For example, consider matrices of size `n * m` and the typing of matrix multiplication, and a type of dates where the range of the day is restricted according to the month. □

1. A `sprintf`-like formating function can also be typed in ML without dependent types if formats are represented as appropriate higher-order functions rather than strings. For details see Danvy (1998).

The Curry-Howard Correspondence

A rather different source for dependent typing is the Curry-Howard correspondence, also known by the slogan *propositions-as-types* (Howard, 1980). Under this correspondence simple types correspond to propositions in the implicational fragment of constructive logic. A formula has a proof if and only if the corresponding type is inhabited. For instance, the formula

$$((A\to B) \to A) \to (A\to B) \to B$$

is valid in constructive logic and at the same time is inhabited, namely by $\lambda f.\lambda u.u(f\,u)$. The underlying philosophical idea behind this correspondence is that a constructive proof of an implication $A \implies B$ ought to be understood as a procedure that transforms any given proof of A into a proof of B.

If propositions are types, then proofs are terms. We can introduce a type constructor Prf which maps a formula A (understood as a type) into the type of its proofs $\mathrm{Prf}\,A$, and then a proof of $A \implies B$ becomes a λ-term of type $\mathrm{Prf}\,A \to \mathrm{Prf}\,B$. Often the type constructor Prf is omitted, notationally identifying a proposition with the type of its proofs. In that case, a proof of $A \implies B$ is simply any term of type $A\to B$.

Generalizing the correspondence to first-order predicate logic naturally leads to dependent types: a predicate B over type A is viewed as a type-valued function on A; a proof of the universal quantification $\forall x{:}A.B(a)$ is—constructively—a procedure that given an arbitrary element x of type A produces a proof of $B(x)$. Hence under the Curry-Howard correspondence we should identify universal quantification with dependent product: a proof of $\forall x{:}A.B(x)$ is a member of $\Pi x{:}A.B(x)$. Indeed, Per Martin-Löf, one of the protagonists of dependent typing (1984), was motivated by this extension. In particular, he introduced type-theoretic equivalents to existential quantification (Σ-types) and equality (identity types), used in the next example.

An important application of the Curry-Howard correspondence is that it allows one to freely mix propositions and (programming language) types. For example, an indexing function $\mathrm{ith}(n)$ to access elements of vectors of length n could be given the type

$$\Pi n{:}\mathrm{Nat}.\Pi l{:}\mathrm{Nat}.\mathrm{Lt}(l,n)\to\mathrm{Vector}(n)\to T$$

where $\mathrm{Lt}(l,n)$ is the proposition asserting that l is less than n. Perhaps more interestingly, we can package up types with axioms restricting their elements. For instance, the type of binary, associative operations on some type T may be given as

$$\Sigma m{:}T\to T\to T.\Pi x{:}T.\Pi y{:}T.\Pi z{:}T.\mathrm{Id}\ (m(x,m(y,z)))\ (m(m(x,y),z))$$

Here Σ`x:A.B(x)` is the type of pairs `(a,b)` where `a:A` and `b:B(a)` and `Id` t_1 t_2 is the type of proofs of the equality t_1=t_2. In Martin-Löf's type theory existential quantification is rendered with Σ-types, the idea being that a constructive proof of $\exists$`x:A.B(x)` would consist of a member `a` of type `A` and a proof, thus a member, of `B(a)`—in other words, an element of Σ`a:A.B(a)`.

2.1.2 EXERCISE [$\star$]: Write down a type which represents a constructive version of the axiom of choice, characterised by: *if for every element* `a` *of a type* `A` *there exists an element* `b` *of* `B` *such that* `P(a,b)` *then there exists a function* `f` *mapping an arbitrary* `x:A` *to an element of* `B` *such that* `P(x, f x)`. □

2.1.3 EXERCISE [$\star$]: Suppose that `f : A→C`, `g : B→C` are two functions with equal target domain. Using set-theoretic notation we can form their *pullback* as $\{(a, b) \in A \times B \mid f\ a = g\ b\}$. Define an analogous type using Σ and `Id`. □

Logical Frameworks

Dependent types have also found application in the representation of other type theories and formal systems. Suppose that we have an implementation of dependent types and want to get a rough-and-ready typechecker for simply typed lambda calculus. We may then make the following declarations:

```
Ty    :: *
Tm    :: Ty → *
base  : Ty
arrow : Ty → Ty → Ty
app   : ΠA:Ty.ΠB:Ty.Tm(arrow A B) → Tm A →Tm B
lam   : ΠA:Ty.ΠB:Ty.(Tm A → Tm B) → Tm(arrow A B)
```

Here `Ty` represents the type of simple type expressions and for `A:Ty` the type `Tm A` represents the type of lambda terms of type `A`. We have a constant `base:Ty` representing the base type and a function `arrow` representing the formation of arrow types. As for terms we have a function `app` that accepts to types `A,B`, a term of type `arrow A B`, a term of type `A` and yields a term of type `B`: the application of the two.

Somewhat intriguingly, the function corresponding to lambda abstraction takes a "function" mapping terms of type `A` to terms of type `B` and returns a term of type `arrow A B`. This way of using functions at one level to represent dependencies at another level is particularly useful for representing syntax with binders, and the technique is known as *higher-order abstract syntax.*

We can now represent familiar terms such as the identity on `A:Ty` by

```
idA = lam A A (λx:Tm A.x)
```

or the Church numeral 2 on type A by

```
two = λA:Ty.lam A (arrow (arrow A A) A)
        (λx:Tm A.lam _ _ (λf:Tm(arrow A A).
              app _ _ f (app _ _ f x)))
```

(replacing some obvious type arguments by underscores to aid readability).

Logical frameworks are systems which provide mechanisms for representing syntax and proof systems which make up a logic. The exact representation mechanisms depend upon the framework, but one approach exemplified in the Edinburgh Logical Framework (Harper, Honsell, and Plotkin, 1993) is suggested by the slogan *judgments-as-types*, where types are used to capture the judgments of a logic.[2]

2.1.4 Exercise [★]: Write down some typing declarations which introduce a judgment expressing an evaluation relation for the representation of simply typed terms shown above. You should begin with a type family Eval which is parameterized on a simple type A and two terms of type Tm A, and declare four terms which represent the rules defining the compatible closure of one-step beta-reduction. □

2.2 Pure First-Order Dependent Types

In this section we introduce one of the simplest systems of dependent types, in a presentation called λLF. As the name suggests, this type system is based on a simplified variant of the type system underlying the Edinburgh LF, mentioned above. The λLF type theory generalizes simply typed lambda-calculus by replacing the arrow type S→T with the dependent product type Πx:S.T and by introducing type families. It is pure, in the sense that it has only Π-types; it is first-order, in the sense that it does not include higher-order type operators like those of F^{ω}. Under the Curry-Howard correspondence, this system corresponds to the ∀,→-fragment of first-order predicate calculus.

Syntax

The main definition of λLF appears in Figure 2-1 and 2-2. The terms are the same as those of the simply typed lambda calculus $\lambda_{\rightarrow}$. The types include type variables X which can be declared in the context but never appear bound. Type variables range over proper types as well as type familes such as

2. Judgments are the statements of a logic or a type system. For example, well-formedness, derivability, well-typedness. In LF these judgments are represented as types and derivations of a judgment are represented as members.

λLF

Syntax

$\mathtt{t}$ ::=		*terms:*
	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x}{:}\mathtt{T}.\mathtt{t}$	*abstraction*
	$\mathtt{t}\ \mathtt{t}$	*application*
$\mathtt{T}$ ::=		*types:*
	$\mathtt{X}$	*type/family variable*
	$\Pi\mathtt{x}{:}\mathtt{T}.\mathtt{T}$	*dependent product type*
	$\mathtt{T}\ \mathtt{t}$	*type family application*
$\mathtt{K}$ ::=		*kinds:*
	$*$	*kind of proper types*
	$\Pi\mathtt{x}{:}\mathtt{T}.\mathtt{K}$	*kind of type families*
Γ ::=		*contexts:*
	$\varnothing$	*empty context*
	$\Gamma, \mathtt{x}{:}\mathtt{T}$	*term variable binding*
	$\Gamma, \mathtt{X}{::}\mathtt{K}$	*type variable binding*

Well-formed kinds $\boxed{\Gamma \vdash \mathtt{K}}$

$$\Gamma \vdash * \qquad \text{(WF-STAR)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{T} \vdash \mathtt{K}}{\Gamma \vdash \Pi\mathtt{x}{:}\mathtt{T}.\mathtt{K}} \qquad \text{(WF-PI)}$$

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\mathtt{X} :: \mathtt{K} \in \Gamma \qquad \Gamma \vdash \mathtt{K}}{\Gamma \vdash \mathtt{X} :: \mathtt{K}} \qquad \text{(K-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{T}_1 :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{T}_2 :: *}{\Gamma \vdash \Pi\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2 :: *} \qquad \text{(K-PI)}$$

$$\frac{\Gamma \vdash \mathtt{S} :: \Pi\mathtt{x}{:}\mathtt{T}.\mathtt{K} \qquad \Gamma \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \mathtt{S}\ \mathtt{t} : [\mathtt{x} \mapsto \mathtt{t}]\mathtt{K}} \qquad \text{(K-APP)}$$

$$\frac{\Gamma \vdash \mathtt{T} :: \mathtt{K} \qquad \Gamma \vdash \mathtt{K} \equiv \mathtt{K}'}{\Gamma \vdash \mathtt{T} :: \mathtt{K}'} \qquad \text{(K-CONV)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma \qquad \Gamma \vdash \mathtt{T} :: *}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash \mathtt{S} :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \lambda\mathtt{x}{:}\mathtt{S}.\mathtt{t} : \Pi\mathtt{x}{:}\mathtt{S}.\mathtt{T}} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \Pi\mathtt{x}{:}\mathtt{S}.\mathtt{T} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{S}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{T}} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{T} \qquad \Gamma \vdash \mathtt{T} \equiv \mathtt{T}' :: *}{\Gamma \vdash \mathtt{t} : \mathtt{T}'} \qquad \text{(T-CONV)}$$

Figure 2-1: First-order dependent types (λLF)

`Vector :: Nat → *`. We may use type and term variables declared in a fixed initial context to simulate the built-in types and operators of a programming language.[3] Apart from variables, types may be dependent products or type family applications. The latter allow us to instantiate families, for example, to give types such as `Vector k` for `k:Nat`.

Kinds allow us to distinguish between proper types and type families. Proper types have kind $*$ while type families have dependent product kinds of the form $\Pi\mathtt{x}{:}\mathtt{T}.\mathtt{K}$.

Contexts may bind term variables and type variables.

3. Strictly speaking, we should consider a *signature* as a special form of context and consider the term and type variables declared in it to be the constants of the language. This isn't necessary when we move to richer type theories in which it is possible to define data types.

λLF

Kind Equivalence $\boxed{\Gamma \vdash \mathsf{K} \equiv \mathsf{K}'}$

$$\frac{\Gamma \vdash \mathsf{T}_1 \equiv \mathsf{T}_2 :: * \qquad \Gamma, \mathsf{x}{:}\mathsf{T}_1 \vdash \mathsf{K}_1 \equiv \mathsf{K}_2}{\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{T}_1.\mathsf{K}_1 \equiv \Pi\mathsf{x}{:}\mathsf{T}_2.\mathsf{K}_2} \quad \text{(QK-Pi)}$$

$$\frac{\Gamma \vdash \mathsf{K}}{\Gamma \vdash \mathsf{K} \equiv \mathsf{K}} \quad \text{(QK-Refl)}$$

$$\frac{\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2}{\Gamma \vdash \mathsf{K}_2 \equiv \mathsf{K}_1} \quad \text{(QK-Sym)}$$

$$\frac{\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_2 \qquad \Gamma \vdash \mathsf{K}_2 \equiv \mathsf{K}_3}{\Gamma \vdash \mathsf{K}_1 \equiv \mathsf{K}_3} \quad \text{(QK-Trans)}$$

Type Equivalence $\boxed{\Gamma \vdash \mathsf{S} \equiv \mathsf{T} :: \mathsf{K}}$

$$\frac{\Gamma \vdash \mathsf{S}_1 \equiv \mathsf{T}_1 :: * \qquad \Gamma, \mathsf{x}{:}\mathsf{T}_1 \vdash \mathsf{S}_2 \equiv \mathsf{T}_2 :: *}{\Gamma \vdash \Pi\mathsf{x}{:}\mathsf{S}_1.\mathsf{S}_2 \equiv \Pi\mathsf{x}{:}\mathsf{T}_1.\mathsf{T}_2 :: *} \quad \text{(QT-Pi)}$$

$$\frac{\Gamma \vdash \mathsf{S}_1 \equiv \mathsf{S}_2 :: \Pi\mathsf{x}{:}\mathsf{T}.\mathsf{K} \qquad \Gamma \vdash \mathsf{t}_1 \equiv \mathsf{t}_2 : \mathsf{T}}{\Gamma \vdash \mathsf{S}_1\ \mathsf{t}_1 \equiv \mathsf{S}_2\ \mathsf{t}_2 : [\mathsf{x} \mapsto \mathsf{t}_1]\mathsf{K}} \quad \text{(QT-App)}$$

$$\frac{\Gamma \vdash \mathsf{T} : \mathsf{K}}{\Gamma \vdash \mathsf{T} \equiv \mathsf{T} :: \mathsf{K}} \quad \text{(QT-Refl)}$$

$$\frac{\Gamma \vdash \mathsf{T} \equiv \mathsf{S} :: \mathsf{K}}{\Gamma \vdash \mathsf{S} \equiv \mathsf{T} :: \mathsf{K}} \quad \text{(QT-Sym)}$$

$$\frac{\Gamma \vdash \mathsf{S} \equiv \mathsf{U} :: \mathsf{K} \qquad \Gamma \vdash \mathsf{U} \equiv \mathsf{T} :: \mathsf{K}}{\Gamma \vdash \mathsf{S} \equiv \mathsf{T} :: \mathsf{K}} \quad \text{(QT-Trans)}$$

Term Equivalence $\boxed{\Gamma \vdash \mathsf{t}_1 {\equiv} \mathsf{t}_2 : \mathsf{T}}$

$$\frac{\Gamma \vdash \mathsf{S}_1 \equiv \mathsf{S}_2 :: * \qquad \Gamma, \mathsf{x}{:}\mathsf{S}_1 \vdash \mathsf{t}_1 \equiv \mathsf{t}_2 : \mathsf{T}}{\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{S}_1.\mathsf{t}_1 \equiv \lambda\mathsf{x}{:}\mathsf{S}_2.\mathsf{t}_2 : \Pi\mathsf{x}{:}\mathsf{S}_1.\mathsf{T}} \quad \text{(Q-Abs)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 \equiv \mathsf{s}_1 : \Pi\mathsf{x}{:}\mathsf{S}.\mathsf{T} \qquad \Gamma \vdash \mathsf{t}_2 \equiv \mathsf{s}_2 : \mathsf{S}}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 \equiv \mathsf{s}_1\ \mathsf{s}_2 : [\mathsf{x} \mapsto \mathsf{t}_2]\mathsf{T}} \quad \text{(Q-App)}$$

$$\frac{\Gamma, \mathsf{x}{:}\mathsf{S} \vdash \mathsf{t} : \mathsf{T} \qquad \Gamma \vdash \mathsf{s} : \mathsf{S}}{\Gamma \vdash (\lambda\mathsf{x}{:}\mathsf{S}.\mathsf{t})\ \mathsf{s} \equiv [\mathsf{x} \mapsto \mathsf{s}]\mathsf{t} : [\mathsf{x} \mapsto \mathsf{s}]\mathsf{T}} \quad \text{(Q-Beta)}$$

$$\frac{\Gamma \vdash \mathsf{t} : \Pi\mathsf{x}{:}\mathsf{S}.\mathsf{T} \qquad \mathsf{x} \notin FV(\mathsf{t})}{\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{T}.\mathsf{t}\ \mathsf{x}\ \equiv\ \mathsf{t} : \Pi\mathsf{x}{:}\mathsf{S}.\mathsf{T}} \quad \text{(Q-Eta)}$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{T}}{\Gamma \vdash \mathsf{t} \equiv \mathsf{t} : \mathsf{T}} \quad \text{(Q-Refl)}$$

$$\frac{\Gamma \vdash \mathsf{t} \equiv \mathsf{s} : \mathsf{T}}{\Gamma \vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{T}} \quad \text{(Q-Sym)}$$

$$\frac{\Gamma \vdash \mathsf{s} \equiv \mathsf{u} : \mathsf{T} \qquad \Gamma \vdash \mathsf{u} \equiv \mathsf{t} : \mathsf{T}}{\Gamma \vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{T}} \quad \text{(Q-Trans)}$$

Figure 2-2: First-order dependent types (λLF)—Equivalence rules

Typechecking Rules

The rules in Figure 2-1 define three judgment forms, for checking kind formation, kinding, and typing.

The characteristic typing rules of the system are the abstraction and application rules for terms, altered to use Π-types. The abstraction introduces a dependent product type, checking that the domain type S is well-formed:

$$\frac{\Gamma \vdash \mathsf{S} :: * \qquad \Gamma, \mathsf{x}{:}\mathsf{S} \vdash \mathsf{t} : \mathsf{T}}{\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{S}.\mathsf{t} : \Pi\mathsf{x}{:}\mathsf{S}.\mathsf{T}} \quad \text{(T-Abs)}$$

The term application rule eliminates a term with this type, substituting the operand in the Π-type:

$$\frac{\Gamma \vdash \mathsf{t}_1 : \Pi\mathsf{x}{:}\mathsf{S}.\mathsf{T} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{S}}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : [\mathsf{x} \mapsto \mathsf{t}_2]\mathsf{T}} \quad \text{(T-App)}$$

The well-formedness check in T-Abs uses the kinding rules to ensure that S is a type. Notice that this check may again invoke the typing rules, in the rule K-App, which checks the instantiation of a type family. The kind formation judgment also invokes the well-formedness of types (in the first premise of Wf-Pi), so the three judgment forms are in fact mutually defined. One consequence is that proofs of properties in this system typically proceed by simultaneous proofs for the different judgment forms, using derivation height as an overall measure or alternatively simultaneous structural induction.

There are two conversion rules, K-Conv and T-Conv, which allow us to replace a kind or type with another one that is equivalent.

Kinds have the general form $\Pi x_1{:}T_1 \ldots x_n{:}T_n . *$ but in the typing rules we only ever need to check for proper types with kind $*$. Nevertheless, we include the K-Conv to ensure that kinding is closed under conversion within the T_i. There is no mechanism for forming kinds by abstraction, so the only way to construct an object of a kind other than $*$ is by declaring it in the context.

Equivalence Rules

One of the main questions in any type system is when two types should be considered equivalent. Type equivalence is in particular needed in the application rules T-App and K-App. To show that some actual argument has an acceptable type for a function or type family, we may need to use the rule T-Conv to convert the type. In fact, the algorithmic typing rules introduced later on show that this is the only place where type equivalence is needed.

But what should our notion of type equivalence be? Without adding special equality axioms, we can consider natural notions of equality which arise from the type structure. With dependent types, a natural notion is to equate types that differ only in their term components, when those term components themselves should be considered equal. So the question is reduced to considering notions of term equality.

A first example is a type-family application containing a β-redex, since we consider β-equivalent λ-terms to be equal: $T\ ((\lambda x{:}S.x)\ z) \equiv T\ z$. A slightly different and more concrete example is two different applications of the Vector family: Vector (3 + 4) $\equiv$ Vector 7. It seems reasonable that a typechecker should accept each of these pairs of types as being equivalent. But we quickly come across more complex equivalences involving more computation, or even, requiring proof in the general case. For example, supposing x is an unknown value of type Nat and f:Nat → Nat is a function whose behavior is known. If it happens that f x=7 for all x then we have Vector (f x) $\equiv$ Vector 7, but this equality could be more difficult to add to an automatic typechecker.

The question of what form of type equivalence to include in a system of dependent types is therefore a central consideration in the design of the system. Many different choices have been studied, leading to systems with fundamentally different character. The most notable distinction between systems is whether or not typechecking is decidable.

In the first case, we may choose to include only basic equalities which are manifestly obvious. This is the viewpoint favored by Martin-Löf, who considers *definitional equality* to be the proper notion of equivalence. The first two equalities above are definitional: $3 + 4$ is definitionally equal to 7 by the rules of computation for addition. Alternatively, one may prefer to include as many equalities as possible, to make the theory more powerful. This is the approach followed, for example, in the type theory implemented by the NuPrl system (1986). This formulation of type theory includes type equivalences like the third example above, which may require arbitrary computation or proof to establish. In such a type system, typechecking is undecidable.

For λLF, we axiomatize definitional equality based on the type structure, which includes β and η equality on lambda terms. It is possible to define this using a relation of equality defined via compatible closure of untyped reduction (this is the approach followed by Pure Type Systems, see Section 2.7). Instead, we give a declarative, typed definition of equivalence, using rules which follow the same pattern as the typing rules. The advantage of this approach is that it is more extensible than the "untyped" approach and avoids the need to establish properties of untyped reduction. See Chapter 6 in this volume for further explanation of the issues here.

The rules for equivalence are shown in Figure 2-2. Again there are three judgments, for equivalence of each of the syntactic categories of terms, types, and kinds. The only interesting rules are Q-Beta and Q-Eta which introduce β and η-equivalence on terms; the remaining rules are purely structural and express that equivalence is a congruence.

2.3 Properties

In this section we mention some basic properties of λLF. We don't go very far: in Section 2.4 we introduce an algorithmic presentation of λLF which allows us to establish further properties indirectly but rather more easily.

Basic Properties

The following properties use some additional notation. Inclusion is defined between contexts as $\Gamma \subseteq \Delta$ iff $\mathtt{x}{:}\mathsf{T} \in \Gamma$ implies $\mathtt{x}{:}\mathsf{T} \in \Delta$, in other words,

$\Gamma \subseteq \Delta$ means that Δ is a permutation of an extension of Γ. We write $\Gamma \vdash J$ for an arbitrary judgment, amongst the six defined in Figures 2-1 and 2-2. We write $\Gamma \vdash K, K'$ to stand for both $\Gamma \vdash K$ and $\Gamma \vdash K'$, and similarly for other judgments.

2.3.1 Lemma [Permutation and Weakening]: Suppose $\Gamma \subseteq \Delta$. Then $\Gamma \vdash J$ implies $\Delta \vdash J$. □

2.3.2 Lemma [Substitution]: If Γ, $\mathtt{x{:}S}$, $\Delta \vdash J$ and $\Gamma \vdash \mathtt{s : S}$, then Γ, $[\mathtt{x} \mapsto \mathtt{s}]\Delta \vdash [\mathtt{x} \mapsto \mathtt{s}]J$. □

2.3.3 Lemma [Agreement]: Judgments in the system are in agreement, as follows:

1. If $\Gamma \vdash \mathtt{T :: K}$ then $\Gamma \vdash \mathtt{K}$.
2. If $\Gamma \vdash \mathtt{t : T}$ then $\Gamma \vdash \mathtt{T} :: *$.
3. If $\Gamma \vdash \mathtt{K} \equiv \mathtt{K'}$ then $\Gamma \vdash \mathtt{K, K'}$.
4. If $\Gamma \vdash \mathtt{T} \equiv \mathtt{T'} :: \mathtt{K}$ then $\Gamma \vdash \mathtt{T, T'} :: \mathtt{K}$.
5. If $\Gamma \vdash \mathtt{t} \equiv \mathtt{t'} : \mathtt{T}$ then $\Gamma \vdash \mathtt{t, t'} : \mathtt{T}$. □

2.3.4 Exercise [★★, ↛]: Prove the lemmas above. □

Strong Normalization

As an auxiliary device for the soundness and completeness of algorithmic typechecking we will now introduce general beta reduction which permits reductions within the scope of abstractions.

We define beta reduction on λLF terms by the four rules:

$$\frac{\mathtt{t_1} \longrightarrow_\beta \mathtt{t_1'}}{\lambda\mathtt{x{:}T_1.t_1} \longrightarrow_\beta \lambda\mathtt{x{:}T_1.t_1'}} \quad \text{(Beta-Abs)}$$

$$\frac{\mathtt{t_1} \longrightarrow_\beta \mathtt{t_1'}}{\mathtt{t_1\ t_2} \longrightarrow_\beta \mathtt{t_1'\ t_2}} \quad \text{(Beta-App1)}$$

$$\frac{\mathtt{t_2} \longrightarrow_\beta \mathtt{t_2'}}{\mathtt{t_1\ t_2} \longrightarrow_\beta \mathtt{t_1\ t_2'}} \quad \text{(Beta-App2)}$$

$$(\lambda\mathtt{x{:}T_1.t_1})\ \mathtt{t_2} \longrightarrow_\beta [\mathtt{x} \mapsto \mathtt{t_2}]\mathtt{t_1} \quad \text{(Beta-AppAbs)}$$

Notice that this reduction does not go inside the type labels of λ abstractions.

The following central result is required to ensure completeness and termination of typechecking, proved in the next section.

2.3.5 Theorem [Strong normalization]: The relation $\longrightarrow_\beta$ is strongly normalizing on well-typed terms. More precisely, if $\Gamma \vdash \mathtt{t}:\mathtt{T}$ then there is no infinite sequence of terms $(\mathtt{t}_i)_{i\geq 1}$ such that $\mathtt{t} = \mathtt{t}_1$ and $\mathtt{t}_i \longrightarrow_\beta \mathtt{t}_{i+1}$ for $i \geq 1$. □

Proof: This can be proved by defining a reduction-preserving translation from λLF to the simply-typed lambda-calculus as follows. First, for every type variable X, no matter of what kind, we introduce a simple type variable $\mathtt{X}^\natural$. Second, for each type expression T, no matter of what kind, we define a simple type expression $\mathtt{T}^\natural$ by $\Pi\mathtt{x}:\mathtt{S}.\mathtt{T}^\natural = \mathtt{S}^\natural \to \mathtt{T}^\natural$ and $(\mathtt{T}\ \mathtt{t})^\natural = \mathtt{T}^\natural$. Finally, the mapping $-^\natural$ is extended to terms and contexts by applying it to all type expressions occurring within.

Now we can show by induction on typing derivations in λLF that $\Gamma \vdash \mathtt{t}:\mathtt{T}$ implies $\Gamma^\natural \vdash \mathtt{t}^\natural:\mathtt{T}^\natural$, from which the result follows by the strong normalization theorem for β-reduction of the simply typed lambda calculus. □

Since $\longrightarrow_\beta$ is finitely branching, this implies that for each term t there exists a number $\mu(\mathtt{t})$ such that if $(\mathtt{t}_i)_{1\leq i\leq k}$ is a reduction sequence starting from t, that is, $\mathtt{t}=\mathtt{t}_1$ and $\mathtt{t}_i \longrightarrow_\beta \mathtt{t}_{i+1}$ for $1 \leq i < k$ then $k \leq \mu(\mathtt{t})$. A term $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow_\beta{}^* \mathtt{t}'$ and $\mathtt{t}' \not\longrightarrow_\beta$ is called a (β) *normal form* of t. Since $\longrightarrow_\beta$ is confluent, see below, normal forms are unique and we may write $\mathtt{t}' = \mathrm{nf}(\mathtt{t})$.

2.3.6 Theorem: The relation $\longrightarrow_\beta$ is confluent. □

2.3.7 Exercise [★★★, ↛]: Prove the theorem in the following way: first show that $\longrightarrow_\beta$ is *locally confluent* in the sense that if $\mathtt{t} \longrightarrow_\beta \mathtt{t}_1$ and $\mathtt{t} \longrightarrow_\beta{}^* \mathtt{t}_2$ then $\mathtt{t}_1 \longrightarrow_\beta{}^* \mathtt{t}'$ and $\mathtt{t}_2 \longrightarrow_\beta \mathtt{t}'$ for some $\mathtt{t}'$. Then conclude confluence using the fact that $\longrightarrow_\beta$ is strongly normalizing. This last part is a standard result from term rewriting known as Newman's Lemma.

Alternatively, you can prove confluence directly using Tait–Martin-Löf's method of parallel reduction, see *TAPL*, Chapter 30. □

2.4 Algorithmic Typing and Equality

To implement λLF, we need to find a formulation of the system that is closer to an algorithm. As usual, we follow the strategy of reformulating the rules to be *syntax-directed*, so that they can be used to define an algorithm going from premises to conclusion (see the description of the implementation in Section 2.9) . We also need an algorithm for deciding type equivalence.

The algorithmic presentation of λLF is shown in Figures 2-3 and 2-4. The judgments mirror the defining presentation, with the addition of a context checking judgment. (This is used only to check an initial context: the rules otherwise maintain context well-formation when extending contexts going from conclusions to premises.)

The non-syntax-directed rules K-CONV and T-CONV are removed. To replace T-CONV, we add equivalence testing in the algorithmic rules for applications, KA-APP and TA-APP.

The equivalence testing rules in Figure 2-4 assume that they are invoked on well-typed phrases. We show these rules with contexts Γ to facilitate extensions to type-dependent equalities or definitions in the context (used in the implementation), although in the rules for pure λLF, the context plays no role in equivalence testing.

The equivalence testing algorithm on terms that is suggested by these rules is similar to the one described in Chapter 6, but we do not make use of type information. (Similarly, the type equivalence rules do not record kinds.) The algorithmic judgment $\Gamma \mapsto \mathtt{s} \equiv \mathtt{t}$ for arbitrary terms is defined mutually with $\Gamma \mapsto \mathtt{s} \equiv_{\mathrm{wh}} \mathtt{t}$ which is defined between *weak head normal forms*. Weak head reduction is a subset of the β reduction $\longrightarrow_{\beta}$, defined by the rules:

$$\frac{\mathtt{t}_1 \longrightarrow_{\mathrm{wh}} \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow_{\mathrm{wh}} \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(WH-APP1)}$$

$$(\lambda \mathtt{x}{:}\mathtt{T}_1.\mathtt{t}_1)\ \mathtt{t}_2 \longrightarrow_{\mathrm{wh}} [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}_1 \qquad \text{(WH-APPABS)}$$

Weak head reduction only applies β-reduction in the head position. The implementation described in Section 2.9 adds expansion of definitions to this reduction; see Chapter 9 for a thorough treatment of how to do this.

2.4.1 THEOREM [WEAK HEAD NORMAL FORMS]: If $\Gamma \vdash \mathtt{t}{:}\mathtt{T}$ then there exists a unique term $\mathtt{t}' = \mathrm{whnf}(\mathtt{t})$ such that $t \longrightarrow_{\mathrm{wh}}{}^{*} \mathtt{t}' \not\longrightarrow_{\mathrm{wh}}$. □

The theorem is a direct consequence of Theorem 2.3.5 and of the fact that $\longrightarrow_{\mathrm{wh}}$ is deterministic (a partial function).

Correctness of the Algorithm

We will now show that the typechecking algorithm defined by the algorithmic rules is sound, complete, and terminates on all inputs.

Since the algorithm checks the context only as it is extended, and (for efficiency) does not check the type of variables in the leaves, we can only expect to show soundness for contexts which are well-formed. The soundness lemma makes use of an auxiliary algorithmic judgment for context formation:

$$\mapsto \varnothing \qquad \text{(WFA-EMPTY)}$$

$$\frac{\mapsto \Gamma \qquad \Gamma \mapsto \mathtt{T} :: *}{\mapsto \Gamma, \mathtt{x}{:}\mathtt{T}} \qquad \text{(WFA-TM)}$$

$$\frac{\mapsto \Gamma \qquad \Gamma \mapsto \mathtt{K}}{\mapsto \Gamma, \mathtt{X}{::}\mathtt{K}} \qquad \text{(WFA-TY)}$$

Algorithmic kind formation $\boxed{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{K}}$

$$\Gamma \mathrel{\mapstochar\!\blacktriangleright} * \qquad \text{(WFA-Star)}$$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{T} :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{T} \mathrel{\mapstochar\!\blacktriangleright} \mathtt{K}}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \Pi \mathtt{x}{:}\mathtt{T}.\mathtt{K}} \qquad \text{(WFA-Pi)}$$

Algorithmic kinding $\boxed{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{T} :: \mathtt{K}}$

$$\frac{\mathtt{X} :: \mathtt{K} \in \Gamma}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{X} :: \mathtt{K}} \qquad \text{(KA-Var)}$$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{T}_1 :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \mathrel{\mapstochar\!\blacktriangleright} \mathtt{T}_2 :: *}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \Pi \mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2 :: *} \qquad \text{(KA-Pi)}$$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{S} :: \Pi \mathtt{x}{:}\mathtt{T}_1.\mathtt{K} \qquad \Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{t} : \mathtt{T}_2 \qquad \Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{T}_1 \equiv \mathtt{T}_2}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{S}\ \mathtt{t} : [\mathtt{x} \mapsto \mathtt{t}]\mathtt{K}} \qquad \text{(KA-App)}$$

Algorithmic typing $\boxed{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{x} : \mathtt{T}} \qquad \text{(TA-Var)}$$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{S} :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{S} \mathrel{\mapstochar\!\blacktriangleright} \mathtt{t} : \mathtt{T}}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \lambda \mathtt{x}{:}\mathtt{S}.\mathtt{t} : \Pi \mathtt{x}{:}\mathtt{S}.\mathtt{T}} \qquad \text{(TA-Abs)}$$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{t}_1 : \Pi \mathtt{x}{:}\mathtt{S}_1.\mathtt{T} \qquad \Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{t}_2 : \mathtt{S}_2 \qquad \Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{S}_1 \equiv \mathtt{S}_2}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{t}_1\ \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{T}} \qquad \text{(TA-App)}$$

Figure 2-3: Algorithmic presentation of λLF

Algorithmic kind equivalence $\boxed{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{K} \equiv \mathtt{K}'}$

$$\Gamma \mathrel{\mapstochar\!\blacktriangleright} * \equiv * \qquad \text{(QKA-Star)}$$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{T}_1 \equiv \mathtt{T}_2 \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \mathrel{\mapstochar\!\blacktriangleright} \mathtt{K}_1 \equiv \mathtt{K}_2}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \Pi \mathtt{x}{:}\mathtt{T}_1.\mathtt{K}_1 \equiv \Pi \mathtt{x}{:}\mathtt{T}_2.\mathtt{K}_2} \qquad \text{(QKA-Pi)}$$

Algorithmic type equivalence $\boxed{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{S} \equiv \mathtt{T}}$

$$\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{X} \equiv \mathtt{X} \qquad \text{(QTA-Var)}$$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{S}_1 \equiv \mathtt{T}_1 \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \mathrel{\mapstochar\!\blacktriangleright} \mathtt{S}_2 \equiv \mathtt{T}_2}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \Pi \mathtt{x}{:}\mathtt{S}_1.\mathtt{S}_2 \equiv \Pi \mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2} \qquad \text{(QTA-Pi)}$$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{S}_1 \equiv \mathtt{S}_2 \qquad \Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{t}_1 \equiv \mathtt{t}_2}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{S}_1\ \mathtt{t}_1 \equiv \mathtt{S}_2\ \mathtt{t}_2} \qquad \text{(QTA-App)}$$

Algorithmic term equivalence $\boxed{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{s} \equiv \mathtt{t}}$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathrm{whnf}(\mathtt{s}) \equiv_{\mathrm{wh}} \mathrm{whnf}(\mathtt{t})}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{s} \equiv \mathtt{t}} \qquad \text{(QA-WH)}$$

$$\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{x} \equiv_{\mathrm{wh}} \mathtt{x} \qquad \text{(QA-Var)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{S} \mathrel{\mapstochar\!\blacktriangleright} \mathtt{t}_1 \equiv \mathtt{t}_2}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \lambda \mathtt{x}{:}\mathtt{S}.\mathtt{t}_1 \equiv_{\mathrm{wh}} \lambda \mathtt{x}{:}\mathtt{S}.\mathtt{t}_2} \qquad \text{(QA-Abs)}$$

$$\frac{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{s}_1 \equiv_{\mathrm{wh}} \mathtt{s}_2 \qquad \Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{t}_1 \equiv_{\mathrm{wh}} \mathtt{t}_2}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{s}_1\ \mathtt{t}_1 \equiv_{\mathrm{wh}} \mathtt{s}_2\ \mathtt{t}_2} \qquad \text{(QA-App)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{S} \mathrel{\mapstochar\!\blacktriangleright} \mathtt{s}\ \mathtt{x} \equiv \mathtt{t} \qquad \mathtt{s} \text{ not a } \lambda}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \mathtt{s} \equiv_{\mathrm{wh}} \lambda \mathtt{x}{:}\mathtt{S}.\mathtt{t}} \qquad \text{(QA-NAbs1)}$$

$$\frac{\Gamma, \mathtt{x}{:}\mathtt{S} \mathrel{\mapstochar\!\blacktriangleright} \mathtt{s} \equiv \mathtt{t}\ \mathtt{x} \qquad \mathtt{t} \text{ not a } \lambda}{\Gamma \mathrel{\mapstochar\!\blacktriangleright} \lambda \mathtt{x}{:}\mathtt{S}.\mathtt{s} \equiv_{\mathrm{wh}} \mathtt{t}} \qquad \text{(QA-NAbs2)}$$

Figure 2-4: Algorithmic presentation of λLF—Equivalence rules

2.4.2 LEMMA [SOUNDNESS OF ALGORITHMIC λLF]: Each of the algorithmic judgments is sound, in the following sense:

1. If $\Gamma \mapsto \mathsf{K}$ then $\Gamma \vdash \mathsf{K}$.
2. If $\Gamma \mapsto \mathsf{T} :: \mathsf{K}$ then $\Gamma \vdash \mathsf{T} :: \mathsf{K}$.
3. If $\Gamma \mapsto \mathtt{t} : \mathsf{T}$ then $\Gamma \vdash \mathtt{t} : \mathsf{T}$.
4. If $\Gamma \mapsto \mathsf{K}, \mathsf{K}'$ and $\Gamma \mapsto \mathsf{K} \equiv \mathsf{K}'$, then $\Gamma \vdash \mathsf{K} \equiv \mathsf{K}'$.
5. If $\Gamma \mapsto \mathsf{T}, \mathsf{T}' :: \mathsf{K}$ and $\Gamma \mapsto \mathsf{T} \equiv \mathsf{T}'$ then $\Gamma \vdash \mathsf{T} \equiv \mathsf{T}' :: \mathsf{K}$.
6. If $\Gamma \mapsto \mathtt{t}, \mathtt{t}' : \mathsf{T}$ and $\Gamma \mapsto \mathtt{t} \equiv \mathtt{t}'$ then $\Gamma \vdash \mathtt{t} \equiv \mathtt{t}' :: \mathsf{K}$.

where in each case, we additionally assume $\mapsto \Gamma$. □

Proof: By induction on algorithmic derivations. □

To establish completeness of algorithmic subtyping and later on termination we need to induct on the length of normalization sequences which we formalize as follows.

Recall that $\mu(\mathsf{s})$ denotes an upper bound on the length of any $\longrightarrow_\beta$ reduction sequence starting from s. We write $|\mathsf{s}|$ for the size of the term s.

2.4.3 DEFINITION: We associate an ω^2-valued weight to each judgment arising in a possible derivation of an equality judgment by

$$w(\Delta \mapsto \mathsf{s}_1 \equiv \mathsf{s}_2) = w(\Delta \mapsto \mathsf{s}_1 \equiv_{\mathrm{wh}} \mathsf{s}_2) + 1$$
$$w(\Delta \mapsto \mathsf{s}_1 \equiv_{\mathrm{wh}} \mathsf{s}_2) = \omega \cdot (\mu(\mathsf{s}_1) + \mu(\mathsf{s}_2)) + |\mathsf{s}_1| + |\mathsf{s}_2| + 1.$$ □

2.4.4 LEMMA [COMPLETENESS OF ALGORITHMIC λLF]: Each of the algorithmic judgments is complete, in the following sense:

1. If $\Gamma \vdash \mathsf{K}$ then $\Gamma \mapsto \mathsf{K}$.
2. If $\Gamma \vdash \mathsf{T} : \mathsf{K}$ then for some K', we have $\Gamma \mapsto \mathsf{T} : \mathsf{K}'$ and $\Gamma \mapsto \mathsf{K} \equiv \mathsf{K}'$ and $\Gamma \mapsto \mathsf{K}'$.
3. If $\Gamma \vdash \mathtt{t} : \mathsf{T}$ then for some T', we have $\Gamma \mapsto \mathtt{t} : \mathsf{T}'$ and $\Gamma \mapsto \mathsf{T} \equiv \mathsf{T}'$ and $\Gamma \mapsto \mathsf{T}' :: *$.
4. If $\Gamma \vdash \mathtt{t}_1 \equiv \mathtt{t}_2 : \mathsf{T}$ then $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_2$.
5. If $\Gamma \vdash \mathsf{T}_1 \equiv \mathsf{T}_2 :: \mathsf{K}$ then $\Gamma \mapsto \mathsf{T}_1 \equiv \mathsf{T}_2$. □

Proof: One first proves that each of the declarative rules is admissible in the algorithmic system. The result then follows by induction on derivations in the declarative system. The only rules that are not immediately obvious are the transitivity rules for equivalence of kinds, types, and terms, and the rule Q-App. These two are left as exercises with solutions. □

2.4.5 Exercise [★★★]: Show that rule QT-Trans is admissible for the algorithmic system in the sense that whenever $\Gamma \vdash \mathtt{t}_i\mathtt{:T}$ for $i = 1,2,3$ and $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_2$ and $\Gamma \mapsto \mathtt{t}_2 \equiv \mathtt{t}_3$ then $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_3$. □

2.4.6 Exercise [★★★]: Show that rule Q-App is admissible for the algorithmic system in the sense that whenever $\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}$ and $\Gamma \vdash \mathtt{s}_1\ \mathtt{s}_2 : \mathtt{T}$ and $\Gamma \mapsto t_1 \equiv \mathtt{s}_1$ and $\Gamma \mapsto \mathtt{t}_2 \equiv \mathtt{s}_2$ then $\Gamma \mapsto \mathtt{t}_{1t2} \equiv \mathtt{s}_{1s2}$. □

Given soundness and completeness, we also want to know that our algorithm terminates on all inputs. This also demonstrates the decidability of the original judgments.

2.4.7 Theorem [Termination of typechecking]: The algorithmic presentation yields a terminating algorithm for typechecking. □

We highlight the crucial ideas of the proof of Theorem 2.4.7 here; the details are left to the diligent reader (Exercise 2.4.9 below). The equivalence judgment $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_2$ introduces a possible source of nontermination when invoked on non-well-typed terms (for example, $\Omega = \Delta\Delta$ where $\Delta = \lambda\mathtt{x{:}A.x\ x}$). Here, computation of weak head normal form runs into an infinite loop. We must be careful that equivalence testing is called only on well-typed terms.

The crucial termination property for term equality that we need is captured by the following lemma.

2.4.8 Lemma: Suppose that $\Gamma \vdash \mathtt{t}_1\mathtt{:T}_1$ and $\Gamma \vdash \mathtt{t}_2\mathtt{:T}_2$. Then the syntax-directed backwards search for a derivation of $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_2$ always terminates. Equivalently, there is no infinite derivation ending in $\Gamma \mapsto \mathtt{t}_1 \equiv \mathtt{t}_2$. □

Proof: We claim that the weight of any premise of a rule is always less than the weight of the conclusion which excludes any infinite derivation tree. In other words we argue by induction on the length of reduction sequences and, subordinately, on the size of terms. This property is obviously satisfied for QA-WH, QA-Abs, QA-App. To deal with rule QA-Nabs1 (and similarly, QA-Nabs2) note that $\mathtt{s}$ must be of the form $\mathtt{y\ u}_1\ \mathtt{...}\ \mathtt{u}_n$ whereby $\mu(\mathtt{s\ x}) = \mu(\mathtt{s})$. The size, however, goes down, as the λ-symbol has disappeared. □

2.4.9 Exercise [★★★, ↛]: Complete the proof of 2.4.2, 2.4.4, and 2.4.7. □

Properties of λLF

We can use the algorithmic presentation of λLF to prove additional properties enjoyed by the main definition. We just mention one example: type preservation under β-reduction.

2.4.10 THEOREM [PRESERVATION]: If $\Gamma \vdash \mathtt{t} : \mathsf{T}$ and $\mathtt{t} \longrightarrow_\beta \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathsf{T}$ also. □

Proof: We show a slightly restricted form of the theorem, for well-formed contexts Γ. More precisely, well-formed contexts are those which can be built using the same rules as for $\mapsto \Gamma$ (page 57), but in the declarative system; the corresponding assertion is written $\vdash \Gamma$. It is easy to extend the completeness lemma to show that $\vdash \Gamma$ implies $\mapsto \Gamma$.

The crucial case is that of an outermost β-reduction (BETA-APPABS), when $\mathtt{t} = (\lambda \mathtt{x}{:}\mathsf{T}_1.\mathtt{t}_1)\ \mathtt{t}_2$ for some T_1, $\mathtt{t}_1$, $\mathtt{t}_2$.

By Lemma 2.4.4, we know that $\Gamma \mapsto (\lambda \mathtt{x}{:}\mathsf{T}_1.\mathtt{t}_1)\ \mathtt{t}_2 : \mathsf{T}'$ for some T' with $\Gamma \mapsto \mathsf{T} \equiv \mathsf{T}'$ and $\Gamma \mapsto \mathsf{T}' :: *$. The first judgment must have been derived with TA-APP preceded by TA-ABS, so we have the derivability of

$$\Gamma \mapsto \mathsf{T}_1 :: * \qquad \Gamma \mapsto \mathsf{T}_1 \equiv \mathsf{S}_1 \qquad \Gamma, \mathtt{x}{:}\mathsf{T}_1 \mapsto \mathtt{t}_1 : \mathsf{S}_2 \qquad \Gamma \mapsto \mathtt{t}_2 : \mathsf{S}_1$$

in the algorithmic system, with $\mathsf{T}' = [\mathtt{x} \mapsto \mathtt{t}_2]\mathsf{S}_2$.

By the above and the assumptions about Γ, we have $\mapsto \Gamma, \mathtt{x}{:}\mathsf{T}_1$. Hence by Lemma 2.4.2, we can go back to get analogs of the statements above in the declarative system. For the last case, to establish the equivalence $\Gamma \vdash \mathsf{T}_1 \equiv \mathsf{S}_1 :: *$ we use Lemma 2.3.3 to get $\Gamma \vdash \mathsf{S}_1 :: *$ and then $\Gamma \mapsto \mathsf{S}_1 :: *$.

Now by T-CONV we have $\Gamma \vdash \mathtt{t}_2 : \mathsf{S}_2$ and so with substitution, Lemma 2.3.2, we get $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_2]\mathtt{t}_1 : [\mathtt{x} \mapsto \mathtt{t}_2]\mathsf{S}_2$ and then the result follows using T-CONV again, with another hop between the systems and Lemma 2.3.3, to show the equivalence $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{t}_2]\mathsf{S}_2 \equiv \mathsf{T} :: *$. □

2.4.11 EXERCISE [★★★, ↛]: Generalize the proof above to arbitrary contexts Γ. □

2.5 Dependent Sum Types

Figure 2-5 shows extensions to λLF to add dependent sum (or "Sigma") types, written $\Sigma \mathtt{x}{:}\mathsf{T}_1.\mathsf{T}_2$. Dependent sums were motivated briefly in the introduction. They generalize ordinary product types in a similar way to the way that dependent products generalize ordinary function spaces. The degenerate non-dependent case, when x does not appear free in T_2, amounts to the ordinary product, written as $\mathsf{T}_1 \times \mathsf{T}_2$.

We extend the terms and types of λLF given in Figure 2-1 with pairs, projection operations, and the type constructor itself. Notice that the pair $(\mathtt{t}_1,\mathtt{t}_2)$

Extends λLF (2-1 and 2-2)

New syntax

$\mathtt{t}$ ::=	...	*terms:*
	$(\mathtt{t},\ \mathtt{t}{:}\Sigma\mathtt{x}{:}\mathtt{T}.\mathtt{T})$	*typed pair*
	$\mathtt{t.1}$	*first projection*
	$\mathtt{t.2}$	*second projection*
$\mathtt{T}$ ::=	...	*types:*
	$\Sigma\mathtt{x}{:}\mathtt{T}.\mathtt{T}$	*dependent sum type*

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\frac{\Gamma \vdash \mathtt{S} :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{T} :: *}{\Gamma \vdash \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T} :: *} \qquad \text{(K-Sigma)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T} :: * \\ \Gamma \vdash \mathtt{t}_1 : \mathtt{S} \qquad \Gamma \vdash \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{T}\end{array}}{\Gamma \vdash (\mathtt{t}_1,\mathtt{t}_2{:}\Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}) : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}} \qquad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}}{\Gamma \vdash \mathtt{t.1} : \mathtt{S}} \qquad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}}{\Gamma \vdash \mathtt{t.2} : [\mathtt{x} \mapsto \mathtt{t.1}]\mathtt{T}} \qquad \text{(T-Proj2)}$$

Term Equivalence $\boxed{\Gamma \vdash \mathtt{t}_1 \equiv \mathtt{t}_2 : \mathtt{T}}$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T} :: * \\ \Gamma \vdash \mathtt{t}_1 : \mathtt{S} \qquad \Gamma \vdash \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{T}\end{array}}{\Gamma \vdash (\mathtt{t}_1,\mathtt{t}_2{:}\Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T})\mathtt{.1} \equiv \mathtt{t}_1 : S} \qquad \text{(Q-Proj1)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T} :: * \\ \Gamma \vdash \mathtt{t}_1 : \mathtt{S} \qquad \Gamma \vdash \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{T}\end{array}}{\Gamma \vdash (\mathtt{t}_1,\mathtt{t}_2{:}\Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T})\mathtt{.2} \equiv \mathtt{t}_2 : [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{T}} \qquad \text{(Q-Proj2)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}}{\Gamma \vdash (\mathtt{t.1},\ \mathtt{t.2}{:}\Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}) \equiv \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{S}.\mathtt{T}} \qquad \text{(Q-SurjPair)}$$

Figure 2-5: Dependent sum types

is annotated explicitly with a type $\Sigma\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2$ in the syntax. The reason for this is that the type of such a pair cannot be reconstructed from the types of $\mathtt{t}_1$ and $\mathtt{t}_2$ alone. For example, if $\mathtt{S}{:}\mathtt{T}{\to}*$ and $\mathtt{x}{:}\mathtt{T}$ and $\mathtt{y}{:}\mathtt{S}\ \mathtt{x}$ the pair $(\mathtt{x},\mathtt{y})$ could have both $\Sigma\mathtt{z}{:}\mathtt{T}.\mathtt{S}\ \mathtt{z}$ and $\Sigma\mathtt{z}{:}\mathtt{T}.\mathtt{S}\ \mathtt{x}$ as a type.

The most cluttered typing rule is the one which introduces a dependent pair, T-Pair. It must check first that the Σ-type itself is allowed, and then that each component has the requested type. The projection rules are straightforward: compare the second projection with the rule T-App in Figure 2-1.

The equality relation on terms is extended to Σ-types by three rules. The first two define the elimination behavior of projections on a pair (compare with the beta rule for Π-types). The third rule, Q-SurjPair, is known as *surjective pairing.* This rule is a form of eta rule for Σ-types: it states that every pair can be formed using the pair constructor.

Algorithmic Typing with Dependent Sum Types

To extend the algorithm to deal with Σ-types, we first extend the notions of beta and weak-head reduction. In both, the main clause is projection on a

Extends λLF algorithm (2-3 and 2-4)

Algorithmic kinding $\boxed{\Gamma \mapsto \mathtt{T} :: \mathtt{K}}$

$$\frac{\Gamma \mapsto \mathtt{T}_1 :: * \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \mapsto \mathtt{T}_2 :: *}{\Gamma \mapsto \Sigma\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2 :: *} \quad \text{(KA-Sigma)}$$

Algorithmic typing $\boxed{\Gamma \mapsto \mathtt{t} : \mathtt{T}}$

$$\frac{\begin{array}{l}\Gamma \mapsto \Sigma\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2 :: * \\ \Gamma \mapsto \mathtt{t}_1 : \mathtt{T}_1' \qquad \Gamma \mapsto \mathtt{T}_1' \equiv \mathtt{T}_1 \\ \Gamma \mapsto \mathtt{t}_2 : \mathtt{T}_2' \qquad \Gamma \mapsto \mathtt{T}_2' \equiv [\mathtt{x} \mapsto \mathtt{t}_1]\mathtt{T}_2\end{array}}{\Gamma \mapsto (\mathtt{t}_1,\mathtt{t}_2{:}\Sigma\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2) : \Sigma\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2} \quad \text{(TA-Pair)}$$

$$\frac{\Gamma \mapsto \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2}{\Gamma \mapsto \mathtt{t.1} : \mathtt{T}_1} \quad \text{(TA-Proj1)}$$

$$\frac{\Gamma \mapsto \mathtt{t} : \Sigma\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2}{\Gamma \mapsto \mathtt{t.2} : [\mathtt{x} \mapsto \mathtt{t.1}]\mathtt{T}_2} \quad \text{(TA-Proj2)}$$

Algorithmic type equivalence $\boxed{\Gamma \mapsto \mathtt{S} \equiv \mathtt{T}}$

$$\frac{\Gamma \mapsto \mathtt{S}_1 \equiv \mathtt{T}_1 \qquad \Gamma, \mathtt{x}{:}\mathtt{T}_1 \mapsto \mathtt{S}_2 \equiv \mathtt{T}_2}{\Gamma \mapsto \Sigma\mathtt{x}{:}\mathtt{S}_1.\mathtt{S}_2 \equiv \Sigma\mathtt{x}{:}\mathtt{T}_1.\mathtt{T}_2} \quad \text{(QTA-Sigma)}$$

Algorithmic term equivalence $\boxed{\Gamma \mapsto \mathtt{t} \equiv_{\mathrm{wh}} \mathtt{t}'}$

$$\frac{\Gamma \mapsto \mathtt{t}_i \equiv \mathtt{t'}_i}{\Gamma \mapsto (\mathtt{t}_1,\mathtt{t}_2{:}\mathtt{T}) \equiv_{\mathrm{wh}} (\mathtt{t'}_1,\mathtt{t'}_2{:}\mathtt{T}')} \quad \text{(QA-Pair)}$$

$$\frac{\Gamma \mapsto \mathtt{t}_i \equiv \mathtt{t}.i \qquad \mathtt{t} \text{ not a pair}}{\Gamma \mapsto (\mathtt{t}_1,\mathtt{t}_2{:}\mathtt{T}) \equiv_{\mathrm{wh}} \mathtt{t}} \quad \text{(QA-Pair-NE)}$$

$$\frac{\Gamma \mapsto \mathtt{t}.i \equiv \mathtt{t}_i \qquad \mathtt{t} \text{ not a pair}}{\Gamma \mapsto \mathtt{t} \equiv_{\mathrm{wh}} (\mathtt{t}_1,\mathtt{t}_2{:}\mathtt{T})} \quad \text{(QA-NE-Pair)}$$

Figure 2-6: Algorithmic typing for Σ-types

pair. Beta reduction also allows reduction inside the components of a pair.

$$(\mathtt{t}_1,\mathtt{t}_2{:}\mathtt{T}).i \longrightarrow_\beta \mathtt{t}_i \quad \text{(Beta-ProjPair)}$$

$$\frac{\mathtt{t} \longrightarrow_\beta \mathtt{t}'}{\mathtt{t}.i \longrightarrow_\beta \mathtt{t}'.i} \quad \text{(Beta-Proj)}$$

$$\frac{\mathtt{t}_1 \longrightarrow_\beta \mathtt{t}_1'}{(\mathtt{t}_1,\mathtt{t}_2{:}\mathtt{T}) \longrightarrow_\beta (\mathtt{t}_1',\mathtt{t}_2{:}\mathtt{T})} \quad \text{(Beta-Pair1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow_\beta \mathtt{t}_2'}{(\mathtt{t}_1,\mathtt{t}_2{:}\mathtt{T}) \longrightarrow_\beta (\mathtt{t}_1,\mathtt{t}_2'{:}\mathtt{T})} \quad \text{(Beta-Pair2)}$$

Weak head reduction just has two new cases:

$$(\mathtt{t}_1,\mathtt{t}_2{:}\mathtt{T}).i \longrightarrow_{\mathrm{wh}} \mathtt{t}_i \quad \text{(WH-ProjPair)}$$

$$\frac{\mathtt{t} \longrightarrow_{\mathrm{wh}} \mathtt{t}'}{\mathtt{t}.i \longrightarrow_{\mathrm{wh}} \mathtt{t}'.i} \quad \text{(WH-Proj)}$$

Using the weak head reduction, the algorithmic typing and equality judgments are extended with the rules in Figure 2-6 to deal with Σ-types.

2.5.1 Exercise [★★★, ↛]: Extend Lemmas 2.4.2, 2.4.4 and 2.4.7 to Σ-types. (No surprises are to be expected.) □

Extends λLF (2-1 and 2-2)

New syntax

`t ::=`	`...`	*terms:*
	`all x:T.t`	*universal quantification*
`T ::=`	`...`	*types:*
	`Prop`	*propositions*
	`Prf`	*family of proofs*

Kinding $\boxed{\Gamma \vdash \mathtt{T} :: \mathtt{K}}$

$$\Gamma \vdash \mathtt{Prop} :: * \qquad \text{(K-PROP)}$$

$$\Gamma \vdash \mathtt{Prf} :: \Pi\mathtt{x}{:}\mathtt{Prop}.\ * \qquad \text{(K-PRF)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \vdash \mathtt{T} :: * \qquad \Gamma, \mathtt{x} : \mathtt{T} \vdash \mathtt{t} : \mathtt{Prop}}{\Gamma \vdash \mathtt{all\ x{:}T.t} : \mathtt{Prop}} \qquad \text{(T-ALL)}$$

Type Equivalence $\boxed{\Gamma \vdash \mathtt{S} \equiv \mathtt{T} :: K}$

$$\frac{\Gamma \vdash \mathtt{T} :: * \qquad \Gamma, \mathtt{x{:}T} \vdash \mathtt{t} : \mathtt{Prop}}{\Gamma \vdash \mathtt{Prf\ (all\ x{:}T.t)} \equiv \Pi\mathtt{x{:}T.Prf\ t} :: *} \qquad \text{(QT-ALL)}$$

Figure 2-7: The Calculus of Constructions (CC)

2.6 The Calculus of Constructions

The Calculus of Constructions (CC), one of the most famous systems of dependent types, was introduced by Coquand and Huet (1988) as a setting for all of constructive mathematics. While it has turned out that CC needs to be extended with certain features (in particular inductive types [Mohring, 1986]), its simplicity in relationship to its expressivity is unprecedented.

In our framework CC can be formulated as an extension of λLF which has a new basic type `Prop` and a new type family `Prf`. Elements of the type `Prop` represent propositions, and also "datatypes" such as the type of natural numbers (we use the term "datatype" to refer to usual programming language types, as opposed to types of proofs of a proposition). Propositions and datatypes are identified in CC by taking the Curry-Howard isomorphism as an identity. The type family `Prf` assigns to each proposition or datatype `p : Prop` the type `Prf p` of its proofs, or, in the case of datatypes, its members. CC has one new term former `all x:T.t`, and two rules which relate it to `Prf`. The additions to λLF are shown in Figure 2-7.

In most presentations and implementations of CC the type `Prf t` is notationally identified with the term `t`. This is convenient and enhances readability, however, we will not adopt it for the sake of compatibility. The original formulation of CC went as far as using the same notation, namely `(x:A)` for all three binders: Π,`all`,λ. That did not enhance readability at all and was thus given up after some time!

CC contains F^{ω} as a subsystem by an obvious translation. For example, here is the type of an encoding of natural numbers in CC:

```
nat = all a:Prop.all z:Prf a.all s:Prf a →Prf a. a
```

Recall that A→B abbreviates Πx:A.B.

Notice that nat is a member of type Prop. The natural numbers inhabit the type Prf nat. Accordingly, we have

```
zero = λa:Prop.λz:Prf a.λs:Prf a → Prf a.z : Prf nat

succ = λn:Prf nat.λa:Prop.λz:Prf a.
          λs:Prf a → Prf a.s(n a z s) : Prf nat → Prf nat


add = λm:Nat.λn:Nat.m nat n succ : Prf nat → Prf nat → Prf nat
```

Regarding higher-order polymorphism here is how we define existential types in CC:

```
exists = λf:A→Prop.all c:Prop.all m:(Πx:Prop.Prf (f x)→Prf c).c
```

Here A is any type; we obtain System F's existential types with A=Prop; we obtain existential quantification over natural numbers with A=Nat.

2.6.1 Exercise [★, ↛]: Define the term corresponding to existential introduction of type: Πf:A→Prop.Πa:Prop.Πi:Prf (f a).Prf (exists f). □

Conversely, existential elimination corresponds to applying a term of type exists f to an appropriately typed argument.

2.6.2 Exercise [★★★, ↛]: Formalize the translation from F^{ω} into CC. □

The combination of type dependency and impredicativity à la System F yields an astonishing expressive power. For example, we can define Leibniz equality as follows:

```
eq = λa:Prop.λx:Prf a.λy:Prf a.
          all p:Prf a→Prop.all h:Prf (p x).p y
   : Πa:Prop.Prf a → Prf a → Prop
```

We can now prove reflexivity of equality by exhibiting an inhabitant of the type Πa:Prop. Πx:Prf a. Prf (eq a x x). Indeed,

```
eqRefl = λa:Prop. λx:Prf a. λp:Prf a → Prop. λh:Prf (p x).h
```

is such a term.

2.6.3 Exercise [★★, ↛]: State and prove symmetry and transitivity of equality. □

In a similar way, we can define other logical primitives such as boolean connectives and quantifiers and then prove mathematical theorems. Occasionally we have to assume additional axioms. For example, induction for the natural numbers can be stated, but not proved; it is thus convenient to work under the assumption:

```
natInd : Πp:Prf nat →Prop.Prf (p zero)
               → (Πx:Prf nat.Prf (p x) → Prf (p(succ x)))
                → Πx:Prf nat.Prf (p x)
```

With that assumption in place we can for example prove associativity of addition in the form of a term of type:

```
Πx:Prf nat.Πy:Prf nat.Πz:Prf nat.
                Prf (eq nat (add x (add y z)) (add (add x y) z))
```

2.6.4 EXERCISE [★★★]: Find such a term. □

The task of finding proof terms inhabiting types is greatly simplified by an interactive goal-directed theorem prover such as LEGO (Luo and Pollack, 1992; Pollack, 1994) or Coq (Barras et al., 1997), or a structure-driven text editor for programming, such as Agda or Alfa (Coquand, 1998; Hallgren and Ranta, 2000).

Algorithmic Typing and Equality for CC

We will now consider algorithmic typechecking for the pure CC. The beta reduction relation is extended with a clause for `all`:

$$\frac{\texttt{t} \longrightarrow_\beta \texttt{t}'}{\texttt{all x:T.t} \longrightarrow_\beta \texttt{all x:T.t}'} \qquad \text{(BETA-ALL)}$$

2.6.5 THEOREM: The relation $\longrightarrow_\beta$ is strongly normalizing on well-typed terms of CC. □

Proof: One can prove this directly using Tait's reducibility method; see, for example, Coquand and Huet (1988) or Luo and Pollack (1992). Alternatively, we can define a reduction-preserving mapping from CC into F^ω by "forgetting" type dependency—e.g., by mappingeq a $\texttt{t}_1$ $\texttt{t}_2$ to $\forall P.P \to P$. Therefore, an alleged infinite reduction sequence in CC would entail an infinite reduction sequence in F^ω. The details are beyond the scope of this survey. □

With this result in place it is now possible to establish soundness, completeness, and termination of algorithmic typing. The additional rules for the algorithm (extending those for λLF) are presented in Figure 2-8.

The Calculus of Inductive Constructions

The fact that induction cannot be proved is a flaw of the impredicative encoding of datatypes. Not only is it aesthetically unappealing to have to make

Extends λLF algorithm (2-3 and 2-4)

Algorithmic kinding $\boxed{\Gamma \mapsto \mathtt{T} :: \mathtt{K}}$

$$\Gamma \mapsto \mathtt{Prop} :: * \qquad \text{(KA-Prop)}$$

$$\frac{\Gamma \mapsto \mathtt{t}:\mathtt{Prop}}{\Gamma \mapsto \mathtt{Prf}\ \mathtt{t} :: *} \qquad \text{(KA-Prf)}$$

Algorithmic typing $\boxed{\Gamma \mapsto \mathtt{t} : \mathtt{T}}$

$$\frac{\Gamma \mapsto \mathtt{T} :: * \qquad \Gamma, \mathtt{x}:\mathtt{T} \mapsto \mathtt{t} : \mathtt{Prop}}{\Gamma \mapsto \mathtt{all}\ \mathtt{x}:\mathtt{T}.\mathtt{t} : \mathtt{Prop}} \qquad \text{(QT-All-E)}$$

Algorithmic type equivalence $\boxed{\Gamma \mapsto \mathtt{S} \equiv \mathtt{T}}$

$$\frac{\begin{array}{c}\mathtt{t} \longrightarrow_{\text{wh}} \mathtt{all}\ \mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 \\ \Gamma \mapsto \mathtt{S}_1 \equiv \mathtt{T}_1 \qquad \Gamma, \mathtt{x}:\mathtt{S}_1 \mapsto \mathtt{S}_2 \equiv \mathtt{Prf}\ \mathtt{t}_2\end{array}}{\Gamma \mapsto \Pi\mathtt{x}:\mathtt{S}_1.\mathtt{S}_2 \equiv \mathtt{Prf}\ \mathtt{t}} \qquad \text{(QKA-Pi-Prf)}$$

$$\frac{\Gamma \mapsto \Pi\mathtt{x}:\mathtt{S}_1.\mathtt{S}_2 \equiv \mathtt{Prf}\ \mathtt{t}}{\Gamma \mapsto \mathtt{Prf}\ \mathtt{t} \equiv \Pi\mathtt{x}:\mathtt{S}_1.\mathtt{S}_2} \qquad \text{(QKA-Prf-Pi)}$$

$$\frac{\Gamma \mapsto \mathtt{s} \equiv \mathtt{t}}{\Gamma \mapsto \mathtt{Prf}\ \mathtt{s} \equiv \mathtt{Prf}\ \mathtt{t}} \qquad \text{(QKA-Prf)}$$

Algorithmic term equivalence $\boxed{\Gamma \mapsto \mathtt{t} \equiv_{\text{wh}} \mathtt{t}'}$

$$\frac{\Gamma \mapsto \mathtt{S}_1 \equiv \mathtt{T}_1 \qquad \Gamma, \mathtt{x}:\mathtt{S}_1 \mapsto \mathtt{s} \equiv \mathtt{t}}{\Gamma \mapsto \mathtt{all}\ \mathtt{x}:\mathtt{S}.\mathtt{s} \equiv_{\text{wh}} \mathtt{all}\ \mathtt{x}:\mathtt{T}.\mathtt{t}} \qquad \text{(QA-All-E)}$$

Figure 2-8: Algorithmic typing for CC

assumptions on an encoding; more seriously, the assumption of `natInd` destroys the analog of the progress theorem (see *TAPL*, §8.3). For example, the following term does not reduce to a canonical form:

```
natInd (λx:Prf nat.nat) zero (λx:Prf nat.λy:Prf nat.zero) zero
```

For these reasons, Mohring (1986) and subsequent authors (Werner, 1994; Altenkirch, 1993) have combined CC with inductive definitions as originally proposed (for a predicative system) by Martin-Löf (1984). In the thus obtained *Calculus of Inductive Constructions* (CIC) as implemented in the Coq theorem prover (Barras et al., 1997) we can declare the type `nat:Prop` as an inductive type with constructors `zero:Prf nat` and `succ:Prf nat→Prf nat`. This generates a constant:

```
natInd : Πp:Prf nat→Prop.Prf (p zero) →
            (Πx:Prf nat.Prf (p x) → Prf (p(succ x))) →
             Πx:Prf nat.Prf (p x)
```

which obeys the following equality rules:

```
natInd p hz hs zero ≡ hz
natInd p hz hs (succ n) ≡ hs n (natInd p hz hs n)
```

This clearly provides induction, but it also allows us to define primitive recursive functions such as addition by

```
add = λx:Prf nat.λy:Prf nat.natInd (λx:nat.nat)
        y (λy:nat.λr:nat.succ r) x
```

Notice that we instantiated `natInd` with the constant "predicate" `λx:nat.nat`.

The mechanism of inductive definitions is not restricted to simple inductive types such as `nat`. CIC, as well as Martin-Löf's predicative systems (as implemented in ALF [Magnusson and Nordström, 1994]) admit the inductive definition of type families as well. For example, with `nat` already in place we may define an inductive family

```
vector : Prf nat → Prop
```

with constructors `nil : Prf (vector zero)` and

```
cons : Πx:Prf nat. Prf nat →
        Prf (vector x) → Prf (vector(succ x))
```

The (automatically generated) induction principle then has the typing

```
vecInd : Πp:Πx:nat.Prf (vector x) → Prop.
   Prf (p zero nil) →
   (Πx:Prf nat.Πy:Prf (vector x).
        Πa:Prf nat.Prf (p y)→Prf (cons x a y)) →
   Πx:Prf nat.Πy:Prf (vector x).Prf (p x y)
```

2.6.6 Exercise [★★, ↛]: What are the equality rules for this induction principle by analogy with the equations for `natInd`? □

Let us see how we can define the exception-free `first` function from the introduction for these vectors. We first define an auxiliary function `first′` that works for arbitrary vectors by

```
first' = vecInd (λx:Prf nat.λv:Prf (vector x).nat)
  zero
  (λx:Prf nat.λy:Prf (vector x).
       λa:Prf nat.λprev:Prf nat.a) :
   Πx:Prf nat.Πv:Prf (vector x).Prf nat
```

This function obeys the equations:

```
first' zero nil = zero
first' (succ x) (cons x a y) = a
```

We obtain the desired function `first` by instantiation

```
first = λx:Prf nat.λy:Prf (vector (succ x)).
              first' (succ x) y
```

The default value zero can be omitted in a system like ALF which allows the definition of dependently-typed functions by pattern matching. In that system one would merely declare the type of first and write down the single pattern

```
first x (cons x a y) = a
```

ALF can then work out that this defines a total function. The extension of pattern matching to dependent types was introduced in Coquand (1992) which also contains beautiful examples of proofs (as opposed to programs) defined by pattern matching. McBride (2000) has studied translations of such pattern matching into traditional definitions using recursion/induction principles like vecInd.

2.6.7 Exercise [★★★, ↛]: Define using vecInd a function

```
concat : Πx:Prf nat.Πy:Prf nat.Prf (vector x) →
                                 Prf (vector y) →
                                 Prf (vector (add x y))
```

How does it typecheck? □

As a matter of fact, the CIC goes beyond the type system sketched here in that it allows quantification over *kinds*, so, for example, the "predicate" p in natInd may be an arbitrary type family. This means that using the constant family p = λx:nat.Prop we can define a function eqZero: Prf nat → Prop which equals true when applied to zero and false on all other arguments. This facility turns out to be useful to define the exception-free first function on vectors which was presented in the introduction.

Another additional feature of the CIC is the separation of propositions and datatypes into two disjoint universes Prop and Set. Both behave like our Prop, the difference lies in a program extraction feature that maps developments in the CIC to programs in an extension of F^ω with inductive types and general recursion. Types and terms residing in Prop are deleted by this translation; only types and terms in Set are retained. In this way, it is possible to extract correct programs from formal correctness proofs. Details can be found in Paulin-Mohring (1989).

Sigma Types in CC

It is unproblematic and useful to combine CC with Σ-types as described in Section 2.5 and Figure 2-5. This allows one to form types of algebraic structures, for instance

```
Semigrp = Σa:Prop.Σop:Prf a → Prf a → Prf a.
            Πx:Prf a.Πy:Prf a.Πz:Prf a.
             Prf (eq a (op x (op y z)) (op (op x y) z));
```

This system is contained in Luo's *Extended Calculus of Constructions* (ECC) (1994) which additionally permits Π and Σ quantification over kinds. For consistency reasons which we will briefly describe next this requires an infinite hierarchy of ever higher kinds $*_0$, $*_1$, $*_2$, ... For instance, in ECC one has

$$\Sigma X{:}*_3.\ X : *_4$$

ECC has been implemented in the LEGO system (Luo and Pollack, 1992).

It is quite another matter to ask for a reflection of Σ-types into the universe `Prop` of datatypes and propositions, by analogy with the way `all` is treated. The temptation is to introduce a term former `ex y:T.t : Prop` when `x:T` ⊢ `t:Prop`, together with an equality rule asserting that

$$\texttt{Pr (ex y:T.t)} \equiv \Sigma\texttt{y:T.Prf t}.$$

Coquand (1986) has shown that the resulting system is unsound in the sense that all types are inhabited and strong normalization fails. Intuitively, the reason is that in this system we are able to define

```
prop = ex x:Prop.nat
```

and now have a mapping `i:Prop→Prf prop` defined by

```
i = λx:Prop.(x,zero:prop)
```

as well as a left inverse `j:Prf prop →Prop` given by

```
j = λx:Prf prop.x.1.
```

Thus, we have reflected the universe `Prop` into one of its members, which allows one to encode (after some considerable effort) one of the set-theoretic paradoxes showing that there cannot be a set of all sets.

This must be contrasted with the impredicative existential quantifier `exists` defined on page 65. The difference between `exists` and the hypothetical term former `ex` is that `exists` does not allow one to project out the existential witness in case it is of type `Prop`.

An existential quantifier which does not provide first and second projections, but only the impredicative elimination rule known from System F is called a *weak sum*, *weak Σ-type*, or *existential*. In contrast, the Σ-types with projections are sometimes called *strong*.

We conclude this section by remarking that it is unproblematic to have "small" strong Σ-types in the CC, that is, if $\texttt{t}_1\texttt{:Prop}$ and $\texttt{x:Prf t}_1 \vdash \texttt{t}_2\texttt{:Prop}$ then $\sigma\texttt{x:Prf t}_1\texttt{.t}_2\texttt{:Prop}$ with the equivalence

$$\texttt{Prf}(\sigma\ \texttt{x:Prf t}_1\texttt{.t}_2) \equiv \Sigma\texttt{x:Prf t}_1\texttt{.Prf t}_2.$$

2.6.8 EXERCISE [★★★, ↛]: An "approximation" for σ `x:Prf t₁.t₂` is given by

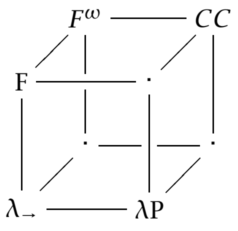
```
exists = all c:Prop.all b:Πx:Prf t₁.Prf t₂ → Prf c.c.
```

Define pairing and first projection for `exists`. Unfortunately, it is not possible to define a second projection. □

2.7 Relating Abstractions: Pure Type Systems

The Calculus of Constructions is a very expressive system, but at first sight, somewhat difficult to understand because of the rich mix of different "levels" of typing (especially in its original formulation with `Prf` implicit). Given a lambda term $\lambda x : S.t$, we cannot tell without (possibly lengthy) further analysis of S and t whether this is a term-level function, a type abstraction, a type family, a type operator, or something else.

Partly as an attempt to explain the fine structure of CC, Barendregt introduced the *lambda cube* of typed calculi (briefly introduced in *TAPL*, Chapter 30), illustrated below:



The cube relates previously known typed lambda calculi (recast within a uniform syntax) to CC, by visualizing three "dimensions" of abstraction. In the bottom left corner, we have $\lambda_\to$with ordinary term-term abstraction. Moving rightwards, we add the type-term abstraction characteristic of dependent types: λP is the Lambda Cube's version of our λLF. Moving upwards, we add the term-type abstraction of System F, capturing polymorphism. Finally, moving towards the back plane of the cube, we add the higher-order type-type abstraction characteristic of F^ω.

Pure Type Systems

The type systems of the Lambda Cube, and many others besides, can be described in the setting of *pure type systems* (Terlouw, 1989; Berardi, 1988; Barendregt, 1991, 1992; Jutting, McKinna, and Pollack, 1994; McKinna and Pollack, 1993; Pollack, 1994). There is an simple and elegant central definition of Pure Type System (PTS) using just six typing rules, which captures a

λP

Syntax

`t`	::=		*terms:*
		s	*sort*
		`x`	*variable*
		`λx:t.t`	*abstraction*
		`t t`	*application*
		`Πx:t.t`	*dependent product type*
s	::=		*sorts:*
		$*$	*sort of proper types*
		$\square$	*sort of kinds*
Γ	::=		*contexts:*
		$\emptyset$	*empty context*
		$\Gamma, x{:}T$	*variable binding*

Typing $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash * : \square \qquad \text{(T-Star)}$$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \qquad \text{(T-Var)}$$

$$\frac{\Gamma \vdash S : * \qquad \Gamma, x{:}S \vdash t : T}{\Gamma \vdash \lambda x{:}S.t : \Pi x{:}S.T} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash t_1 : \Pi x{:}S.T \qquad \Gamma \vdash t_2 : S}{\Gamma \vdash t_1\ t_2 : [x \mapsto t_2]T} \qquad \text{(T-App)}$$

$$\frac{\Gamma \vdash S : s_i \qquad \Gamma, x{:}S \vdash T : s_j}{\Gamma \vdash \Pi x{:}S.T : s_j} \qquad \text{(T-Pi)}$$

$$\frac{\Gamma \vdash t : T \qquad T \equiv T' \qquad \Gamma \vdash T' : s}{\Gamma \vdash t : T'} \qquad \text{(T-Conv)}$$

where $(s_i, s_j) \in \{(*, *), (*, \square)\}$.

Figure 2-9: First-order dependent types, PTS-style (λP)

large family of systems constructed using Π-types. This uniform presentation allows one to establish basic properties for many systems at once, and also to consider mappings between type systems (so-called *PTS morphisms*).

A presentation of λLF as a Pure Type System is given in Figure 2-9.

The first thing to notice about PTSs is that there is a single syntactic category of terms, used to form types, terms, and abstractions and applications of different varieties. Although formally there is a single syntactic category, we use the same meta-variables as before, to aid intuition. (So the letters T and K and also range over the syntactic category of terms, but the system will determine that they are types and kinds, respectively).

To allow levels of types and kinds to be distinguished, the PTS framework uses tokens called *sorts* to classify different categories of term, within the formal system itself. The system λP requires two sorts: first, $*$, which is the kind of all proper types, as used before, and second, $\square$, which is the sort that classifies well-formed kinds. Judgments of the form $\Gamma \vdash T : *$ replace $\Gamma \vdash T :: *$ from Figure 2-1, and judgments $\Gamma \vdash K : \square$ replace $\Gamma \vdash K$.

The rule T-Pi controls formation of Π-types, by restricting which sorts we are allowed to quantify over. In turn, this restricts which λ-abstractions can be introduced by T-Abs. For λLF, there are two instances of λ-abstraction and two instances of Π-formation. In the PTS presentation, these are captured by the two pairs of sorts allowed in T-Pi. When $s_i = s_j = *$, we have the first-order dependent product type, and when $s_j = \square$ we have the kind of type families, corresponding respectively to K-Pi and Wf-Pi in Figure 2-1.

The conversion rule is the main point of departure. The equivalence relation `s` ≡ `t` in Pure Type Systems is defined between untyped terms, as the compatible closure of β-reduction. This has a strong effect on the metatheory.

2.7.1 Exercise [★★★★]: Using the obvious mapping from the syntax of λLF into the syntax of λP, give a proposition stating a connection between the two presentations. Try to prove your proposition. □

Systems of the Lambda-Cube and Beyond

The other systems of the Lambda Cube can be expressed using the same rules as in Figure 2-9, with the single difference of changing the combinations of pairs of sorts (s_i, s_j) allowed in T-Pi. This controls which kind of abstractions we can put into the context. The table below characterises the systems of the Lambda Cube:

System	PTS formation rules				
$\lambda_{\rightarrow}$	{ (∗,∗)				}
λP	{ (∗,∗),	(∗,□)			}
F	{ (∗,∗),		(□,∗)		}
F^{ω}	{ (∗,∗),		(□,∗),	(□, □)	}
CC	{ (∗,∗),	(∗,□),	(□,∗),	(□, □)	}

Further PTSs are given by adjusting the axiom T-Star of Figure 2-9, which is another parameter in the formal definition of PTS. For example, if we take the axiom to be

$$\Gamma \vdash * : * \qquad \text{(T-TypeType)}$$

(together with the T-Pi restriction of $\{(*, *)\}$), we obtain a system where $*$ is the sort of all types including itself. In this system, all types are inhabited and there are non-normalizing terms (as in the result of Coquand, 1986 mentioned on page 70). Though this renders the logical interpretation of the system meaningless, it is debatable whether such systems may nonetheless be useful in some situations as type systems for programming languages.

For further details of Pure Type Systems, we refer the reader to the references given at the end of the chapter.

2.8 Programming with Dependent Types

The task of building practical programming languages with dependent types is a topic of current research. Early languages include Pebble (Lampson and Burstall, 1988) and Cardelli's Quest (Cardelli and Longo, 1991). Programming in Martin-Löf's type theory is described in the monograph (Smith, Nordström, and Petersson, 1990). More recently, Augustsson introduced a language called Cayenne (1998), with a syntax based on the functional programming language Haskell, and Xi and Pfenning studied the language Dependent ML, based around a fragment of Standard ML (1998; 1999). The difference between Cayenne and Dependent ML goes beyond the choice of underlying language, and serves to illustrate a fundamental design decision for practical programming with dependent types.

Languages with Undecidable Typechecking

Given the expressivity of dependent types as illustrated in previous sections it is natural and tempting to add them to a programming language. The price for this expressivity is, however, the complexity of typechecking. As we have explained, typechecking dependent types requires deciding equality of terms as a subtask which in turn requires the underlying term language to be strongly normalizing. On the other hand, most practical programming languages provide general recursion with possible nontermination. Simply adding dependent types to a Turing-complete term language invariably leads to undecidable typechecking.

Of course, typechecking remains semi-decidable, so one can simply wait for an answer for a reasonable amount of time before giving up and turning the typechecker off. This is basically the (surprisingly successful) approach undertaken in Cayenne. Another example is the theorem prover PVS (1996) which includes a dependently-typed programming language (at the time of writing, in an experimental stage), and also has semi-decidable typechecking. In PVS, however, it is possible to resort to interactive theorem proving to aid the type checker.

Undecidablef typechecking is not to the taste of all programming language designers, and for reasons such as scalability, may not be suitable for general application. The alternative is to consider dependently typed languages built around standard programming language features, yet with low-complexity typechecking algorithms. To achieve this one must sacrifice some of the generality of dependent types. Dependent ML (DML) is a proposal which follows this approach, which we will investigate in more detail in the remainder of this section. A type system closely related to that of DML, but aimed at Haskell, was studied by Zenger, under the name *indexed types* (1997).

Exactly because this class of type systems have the desirable feature that they provide “static” typechecking independently from execution or equivalence checking of terms, some authors prefer not to call them “dependent” at all. The definition of dependent types given in Chapter 8 is slightly stricter than ours, and contrasts statically typed languages like DML and Indexed Types with languages where there is a lack of *phase distinction* between the compilation and execution of a program (see page 305).

A Simplified Version of Dependent ML

The crucial idea behind DML is that type dependency on terms is not allowed for arbitrary types, but only for certain *index sorts*. Typechecking gives rise to well-behaved constraint systems on terms belonging to index sorts. Typechecking and even (to an extent) type inference can then be reduced to a constraint-solving problem over the index sorts, which is decidable.

In this presentation we fix the index sorts to be integer numbers and linear subsets thereof, although Pfenning and Xi consider richer possibilities. We also base the language on the lambda-calculi we have studied so far, rather than a form of Standard ML.

Before going into details we will look at some simple examples concerning vectors. We write `int` for the index sort of integers and assume a basic type `data` and a basic type family `Vector : int→∗` where `Vector[n]` denotes arrays over `data` of length `n` as usual. Note that, for example, `Vector[-1]` will be empty. Next, we introduce the constants

```
nil  : Vector[0]
cons : Πn:int.data → Vector[n] → Vector[n+1]
```

and a construct for pattern matching obeying the following typing rule:

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{t}_1 : \mathtt{Vector[i]} \qquad \Gamma, \mathtt{i=0} \vdash \mathtt{t}_2 : \mathtt{T} \\ \Gamma, \mathtt{n{:}int}, \mathtt{x{:}data}, \mathtt{l{:}Vector[n]}, \mathtt{i=n+1} \vdash \mathtt{t}_3 : \mathtt{T}\end{array}}{\Gamma \vdash \mathtt{match}\ \mathtt{t}_1\ \mathtt{with}\ \mathtt{nil} \to \mathtt{t}_2 \mid \mathtt{cons[n](x,l)} \to \mathtt{t}_3 : \mathtt{T}} \quad \text{(MATCH-VECTOR)}$$

There are several things to notice here. Types distinguish between ordinary non-dependent function spaces $T_1 \to T_2$ and type families indexed by index sorts, $\Pi x{:}I.T$. Application for Π-types is written using square brackets. Contexts contain bindings of index variables to index sorts, type variables to types, and constraints over terms of index sort. Here the constraints are equations; in general they may be propositions of some restricted form so as to retain decidability.

In our setting, `nil`, `cons`, and `match` are just interesting for their typing behaviors. We might postulate the obvious conversion rules for instances of `match`, to define a term equality judgment as studied earlier. But it is important to realize that we needn't do this for the purpose of typechecking, since for DML-style systems term equality is completely decoupled from typing.

In examples we will allow the definition of recursive functions by first declaring them with their type and then giving an accordingly typed implementation which may involve calls to the function being defined.[4]

Example: Appending Vectors

We want to define a function for appending two vectors. It should obey the following typing:

```
append : Πm:int.Πn:int.Vector[m] → Vector[n] → Vector[m+n]
```

To do this we define the body of `append` as follows:

```
append-body = λm:int.λn:int.λl:Vector[m].λt:Vector[n].
    match l with
            nil → t
          | cons[r](x,y) → cons[r+n](x,append[r][n](y,t)
```

We should prove that `append-body` has the same type as `append`. Let Γ = `m:int, n:int, l:Vector[m], t:Vector[n]`. After applying the rule Match-Vector backwards we are required to show that

$$\Gamma, \texttt{m=0} \vdash \texttt{t : Vector[m+n]}$$

and

$$\Gamma, \texttt{r:int}, \texttt{x:data}, \texttt{y:Vector[r]}, \texttt{m=r+1} \vdash \\ \texttt{cons[r+n](x,append[r][n](y,t) : Vector[m+n]}$$

For the first requirement, we notice that Γ, `m=0` $\vdash$ `n=m+n:int` from which the claim will follow by the type conversion rule and the obvious type equivalence which equates instances of `Vector` indexed by equal index terms:

$$\frac{\Gamma \vdash \texttt{i=j}}{\Gamma \vdash \texttt{Vector[i]=Vector[j]}}$$

This rule is an instance of QT-App for DML families.

For the second requirement, we first notice that, given the purported typing of `append`, the `append`-subterm has type `Vector[r+n]`, thus, by the typing of `cons` the term itself has type `Vector[r+n+1]`, but in the given context, this is equal to `Vector[m+n]` hence the required typing follows by type conversion again.

4. One can achieve this effect with a constant $\texttt{fix}_T$ `: (T→T) → T` for any type `T`.

Example: Splitting a Vector

This example illustrates DML's restricted form of Σ-types. Again, we have both dependent sums indexed by index sorts, and non-dependent sums (i.e., ordinary cartesian products). We will use the following type abbreviation:

```
T(m) = Σp:int.Σq:{ i | p+i=m }.Vector[p] * Vector[q]
```

The type `T(m)` has elements of the form `(p,(q,(k,l)))`, which we shall write as `(p,q,k,l)` to avoid excessive parentheses. The terms `p` and `q` are integer indices, obeying the constraint $p + q = m$.

Now we can define a `split` function that partitions a vector into two pieces of given lengths:

```
split : Πm:int.Vector[m] → T(m)

split-body = λm:int.λl:Vector[m].
    match l with
       nil ⇒ (0,0,nil,nil) : T(0)
     | cons[r](x,y) ⇒ let (p,q,u,v) = split[r](y) in
             if test(x) then (p+1, q, cons[p](x,u), v) : T(r+1)
                        else (p, q+1, u, cons[q](x,v)) : T(r+1)
```

where `test(x)` is some unspecified boolean-valued term. The typing of `split` guarantees that the result vectors could be appended to form a vector with the same length as the input. Notice that we can say that there is *some* pair `p` and `q` such that `p+q=m` where `m` is the length of the input, but with the restricted form of predicates in DML, we cannot say that `p` is equal to the number of elements `x` from the input for which `test(x)` is true.

To see how `split` is typed, let Γ = `m:int, l:Vector[m]`. We have Γ, `m=0` $\vdash$ `T(0)=T(m)` which deals with the first case of the match. For the second case, we need to show

$$\begin{array}{l}\Gamma, \texttt{p:int}, \texttt{q:int}, \texttt{p+q=r}, \texttt{u:Vector[p]}, \texttt{v:Vector[q]}, \texttt{r+1=m} \vdash \\ \quad \texttt{(p+1, q, cons[p](x,u), v) : T(r+1)} = \texttt{T(m)}\end{array}$$

and similarly for the `else`-branch of the `if` statement. Again this follows from trivial equational reasoning, and the expected rules for sum types.

Definition of Simplified DML

Figure 2-10 summarizes the syntax of our simplified form of DML. Most of the typing rules are routine, so we leave completing the system to exercises.

The definition of DML is closely related to λLF with Σ-types, except that dependencies are restricted to special terms of index sorts, so there is a partitioning of the syntax. Index sorts comprise the integers and subsets of index

DML

I	::=		*index sorts:*
		int	*index sort of integers*
		$\{x{:}I \mid P\}$	*subset sort*
P	::=		*propositions:*
		P ∧ P	*conjunction*
		i<=i	*index inequality*
i	::=		*index terms:*
		x	*variable*
		q	*constant* $q \in \mathbb{Z}$
		qi	*multiplication by* $q \in \mathbb{Z}$
		i+i	*addition*
t	::=		*terms:*
		x	*variable*
		λx:I.t	*index abstraction*
		t[i]	*index application*
		λx:T.t	*abstraction*
		t t	*application*
		(i, t)	*index pairing*
		(t, t)	*term pairing*
		let (x, y)=t in t	*projection*
T	::=		*types:*
		X	*type/family variable*
		Πx:I.T	*indexed product*
		Σx:I.T	*indexed sum*
		T[i]	*type family application*
		$T_1 \to T_2$	*function type*
		$T_1 * T_2$	*cartesian product*
K	::=		*kinds:*
		∗	*kind of proper types*
		Πx:I.K	*kind of type families*
Γ	::=		*contexts:*
		∅	*empty context*
		Γ, x:T	*term variable binding*
		Γ, x:I	*index variable binding*
		Γ, P	*constraint*

Figure 2-10: Simplified Dependent ML (DML)

sorts. Subset formation is permitted only with respect to a restricted set of predicates. In our case, these are conjunctions of linear inequalities (equality of two indices, $i_1=i_2$, can be defined as $i_1<=i_2 \wedge i_2<=i_1$). Index terms themselves are restricted to variables, constants, addition of terms and multiplication by constants. Given an index sort, proposition, or index term $\mathcal{I}$, we write *FIV*($\mathcal{I}$) to stand for the free (index) variables of $\mathcal{I}$. We use the same category of variables for index variables and ordinary variables, but we can tell from a typing context whether a variable ranges over index terms or ordinary terms. Given a context Γ, let *IV*(Γ) stand for the set of index variables declared in Γ. A term $\mathcal{I}$ in the index syntax is well-formed in Γ just in case *FIV*($\mathcal{I}$) ⊆ *IV*(Γ); no typing rules are needed to check well-formedness in the index syntax. For contexts, we assume as usual that no variable is bound more than once, and moreover, that the free variables appearing in declarations x:I and constratints P are declared earlier in the context.

Ordinary terms include index terms in application position and in the first component of pairs. There are types depending on index terms, but there are no types depending on ordinary terms. As a result, function space and cartesian product cannot be defined as special cases of Π and Σ-types, but must be included as primitives. Kinds are just as in LF, except that dependency is restricted to index sorts `I`.

In the typing rules we assume given two semantically defined judgments:

$\Gamma \models \mathtt{P}$	`P` *is a consequence of* Γ
$\Gamma \models \mathtt{i} : \mathtt{I}$	`i:I` *follows from the assumptions of* Γ

These judgments depend only on the index assumptions and propositions in Γ, and their intention should be clear. For example, we have:

$$\mathtt{x:\{y:int \mid y>=8\},\ z:int,\ z>=9} \models \mathtt{x+z >= 13}$$

The judgments can be defined formally using the obvious interpretation of the index syntax in $\mathbb{Z}$ (see Exercise 2.8.1).

In practice we are of course interested an algorithm for deriving the two judgments. In our simplified version of DML, both judgments $\Gamma \models \mathtt{P}$ and $\Gamma \models \mathtt{i:I}$ are decidable, and there are well-known methods which we can use for handling linear equalities over the integers. In the case of a more complicated index language the judgments might both be undecidable; for instance, if we allow multiplication of index terms and existential quantification in propositions then undecidability follows from the undecidability of Hilbert's 10th problem.

In the typing rules, the semantic judgment is used whenever we need to check that an index term belongs to an index sort. For example, the rule for type family application becomes:

$$\frac{\Gamma \vdash \mathtt{S} :: \Pi\mathtt{x{:}I.K} \qquad \Gamma \models \mathtt{i} : \mathtt{I}}{\Gamma \vdash \mathtt{S[i]} : [\mathtt{x} \mapsto \mathtt{i}]\mathtt{K}} \qquad \text{(DML-K-App)}$$

The typing rules for the remainder of the language are defined similarly to λLF and the simply-typed lambda calculus. For instance, we have the following rule for index abstraction:

$$\frac{\Gamma, \mathtt{x{:}I} \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \lambda\mathtt{x{:}I.t} : \Pi\mathtt{x{:}I.T}} \qquad \text{(DML-I-Abs)}$$

but for ordinary abstraction we introduce the arrow:

$$\frac{\Gamma, \mathtt{x{:}S} \vdash \mathtt{t} : \mathtt{T}}{\Gamma \vdash \lambda\mathtt{x{:}S.t} : \mathtt{S} \to \mathtt{T}} \qquad \text{(DML-T-Abs)}$$

There are similarly two rules for pairing and for projections. For the projection of an indexed pair, we have the dependent case:

$$\frac{\Gamma \vdash \mathtt{t} : \Sigma\mathtt{x{:}I.T} \qquad \Gamma, \mathtt{x{:}I}, \mathtt{y{:}T} \vdash \mathtt{t'} : \mathtt{T'}}{\Gamma \vdash \mathtt{let\ (x,y)=t\ in\ t'} : \mathtt{T'}} \qquad \text{(DML-I-Proj)}$$

We can also follow the same procedure as for λLF to formulate an algorithmic version of typing; the difference is that algorithmic type equality amounts to checking of index constraints which can be performed semantically by constraint solving, without any normalization. In particular, equality of terms is not intertwined with typechecking at all. The crucial rule for algorithmic equality is

$$\frac{\Gamma \mapsto \mathtt{S}_1 \equiv \mathtt{S}_2 \qquad \Gamma \models \mathtt{i}_1 = \mathtt{i}_2}{\Gamma \mapsto \mathtt{S}_1\ \mathtt{i}_1 \equiv \mathtt{S}_2\ \mathtt{i}_2} \qquad \text{(DML-QIA-App)}$$

where the second judgment is an instance of the semantic consequence judgment $\Gamma \models \mathtt{P}$.

2.8.1 Exercise [★★]: Give a semantic interpretation of DML index syntax. Considering only the index variables in Γ, an *index environment* η is a function from index variables to integers. Given this notion, we can define $\Gamma \models \mathtt{P}$ as $\forall \eta.\ \eta \models \Gamma. \Longrightarrow \eta \models \mathtt{P}$. Complete the definition. □

2.8.2 Exercise [★★★, ↛]: Complete the presentation of DML by defining the typechecking judgments and give an algorithm for typechecking. □

Closing Example: Certifying Parameters

Several motivating application examples have been given for DML in the literature, including eliminating array bounds checks and unnecessary cases from pattern matches. Rather than repeat those examples, we give a slightly different kind of example to illustrate the use of DML-style typing to certify that constraints are met on parameters of functions.[5]

The scenario is that we are programming for an embedded system which is providing safety features for an onboard computer in an automobile. We are provided with a system call:

```
brake : int * int → unit
```

where it is safety critical that whenever `brake` is called with parameters `(x,y)` then some proposition $P(x,y)$ must be satisfied, for example, a conjunction of linear inequalities describing some "safe window."

To guarantee this, we should try to type our main program under the following assumed typing for `brake`. Notice that `brake` is provided as a system call, so we can assume an arbitrary typing for it.

```
brake : {(x,y) : int * int | P} → unit
```

5. This example is taken from the project *Mobile Resource Guarantees* (EU IST-2001-33149); see `http://www.lfcs.inf.ed.ac.uk/mrg`.

where P encodes $P(x, y)$. Unfortunately, this typing does not quite fit into the DML-framework since it unduly mixes index sorts and ordinary types. To repair this, we introduce a type family Int : int → ∗ with the intuition that Int(x) is a singleton type containing just the integer x, as a "run-time" integer rather than an index term. We also need special typings for run-time integers:

```
0 : Int(0)
1 : Int(1)
plus : Πx,y:int.Int(x) → Int(y) → Int(x+y)
times_q : Πx:int.Int(x) → Int(qx)
```

where q is a fixed integer. These typings allow us to reflect the index terms in ordinary terms. Moreover, we need a type family Bool:int→∗ with the intuition that Bool(x) contains true if 1<=x and Bool(x) contains false if x<=0. Now we can suppose constants:

```
true  : Πx:int|1<=x. Bool(x)
false : Πx:int|x<=0. Bool(x)
leq   : Πx,y:int. Int(x) → Int(y) → Bool(1+y-x)
```

(where we write Πx:int|P.T as an abbreviation of Πx:{x:int | P}. T).

We also need a construct for case distinction obeying the following typing rule:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool(i)} \qquad \Gamma, \mathtt{1<=i} \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma, \mathtt{i<=0} \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}}$$

Notice that if we define boolean negation in terms of if-then-else then we would obtain the typing:

```
not : Πx:int. Bool(x) → Bool(1-x)
```

because 1<=x $\models$ 1-x<=0 and x<=0 $\models$ 1<=1-x. Unfortunately, the derived typings for conjunction and disjunction are rather weak:

```
andalso,orelse : Πx,y:int.Bool(x)→Bool(y) → Σz:int.Bool(z)
```

Xi introduces a separate index sort of booleans with the usual operations on the index level. This gives tighter typings for the boolean operations like

$$\mathtt{orelse} : \Pi \mathtt{x,y{:}bool.Bool(x)} \to \mathtt{Bool(y)} \to \mathtt{Bool(x} \wedge \mathtt{y)}.$$

The price is that constraint solving for such a combined theory is much more complex.

Returning to the example with the system call, let us suppose that the linear constraint P simply states x+y<=10 and that the main function is just a wrapper around brake that makes sure the constraint is actually met, i.e.

```
main(x,y) = if x+y<=10 then call brake(x,y) else call brake(0,0)
```

Here is the corresponding DML version. We assume that the system call brake satisfies the typing

```
brake : Πx,y:int|x+y<=10.Int(x) → Int(y) → unit

main : Πx,y:int.Int(x) → Int(y) → unit
main-body = λx,y:int.λxx:Int(x).λyy:Int(y).
                 if leq[x+y,10](plus[x,y](xx,yy))
                       then brake[x,y](xx,yy)
                       else brake[0,0](0,0)
```

Although this example is rather simple, it illustrates the general technique for connecting index sort constraints to function calls. The fact that this definition is type correct guarantees that the required safe window for calls to brake is indeed always obeyed.

Summary and Outlook

We have shown the theory of a simplified fragment of Pfenning and Xi's DML demonstrating the important feature that typechecking amounts to constraint solving, for example, in the domain of integers, rather than normalizing terms. In this way, it becomes possible to retain decidability of typechecking in the presence of general recursion.

The DML examples show that index annotations are quite heavy. Fortunately, most can be inferred automatically by a process known as *elaboration*. It is plausible that in the examples we can reconstruct the annotations by replacing them by indeterminate linear terms in the index variables in scope and then solving for the coefficients. In Xi's thesis (1998), elaboration is presented in detail as a logic program in the style of our algorithmic subtyping.

One of the design criteria behind the original DML was to allow ordinary Standard ML programs to be extended with additional type annotations. Current research in dependent type systems for programming seeks further advances at the programming language level. The aim is to provide more comfortable high-level notations and new programming language abstractions for applying dependent type theory. One example of this is by enriching pattern matching, see McBride (2000) and McBride and McKinna (2004).

Underlying type theories such as CIC are amply expressive for this purpose; the challenge lies in making these systems more convenient to use, by adding programming language constructions, notational conveniences and advanced inference techniques. Present implementations, oriented towards mathematical interactive proof development, need to be adapted to programming language settings. These exciting developments leave much to be expected for the future of programming with dependent types.

2.9 Implementation of Dependent Types

In this final section we describe an OCaml implementation of the dependent type theory described in preceding sections. The implementation allows declarations and definitions of both terms and types. Typechecking occurs as soon as a declaration or definition is given. A term may be given with a type, which will be checked, or without, in which case one will be inferred. Similarly for kinds. Finally, we can ask to normalise well-typed terms.

The typechecking algorithm proceeds by evaluating the rules in Figures 2-4 and 2-3 and the later tables extending these judgments. More precisely, we have (simultaneously defined) functions:

```
val whnf : term → term
val typeof : context → term → ty
val kindof : context → ty → kind
val checkkind :  context → kind → unit
val tyeqv : context → ty → ty → bool
val kindeqv : context → ty → ty → bool
val tmeqv : context → ty → ty → bool
```

These functions are implemented by encoding the algorithmic rules using pattern matching. For example, the definition of `tmeqv` begins like so:

```
  tmeqv ctx tm1 tm2 =
    let tm1' = whred true ctx tm1 in
    let tm2' = whred true ctx tm2 in
    match (tm1',tm2') with
      (TmVar(fi,i,j), TmVar(fi',i',j')) → i=i'
   |  (TmAbs(_,x,tyS1,tmS2),TmAbs(_,y,tyT1,tmT2))→
        let ctx' = addbinding ctx x (VarBind(tyS1)) in
        tmeqv ctx'  tmS2 tmT2
...
```

(the first argument of `whred` is a flag indicating whether to allow definitions in the context to be expanded).

We stress that the implementation is a direct rendition of the syntax and rules described earlier. It does not include any of the numerous desirable features that make programming with dependent types more convenient, such as argument synthesis (Harper and Pollack, 1991) or interactive, goal-directed construction of terms. Conversely, because the implementation is simple, it should be straightforward to experiment with extensions. The program is built on the F^{ω} implementation from *TAPL* and uses the same design and data structures (see *TAPL*, Chapters 6, 7, and 30).

We illustrate the use of the implementation by way of some examples. Notice that the ASCII input to the system to produce a type like `Πx:A.B` is `Pi x:A.B`.

Examples

With the commands

```
A : ∗;
Nat : ∗;
zero : Nat;
succ : Πn:Nat.Nat;
Vector : Πn:Nat.∗;
```

we declare variables `A,Nat`, constants `zero` and `succ` intended to denote zero and successor on the natural numbers, and a type `Vector` depending on type `Nat`. Note that the implementation does not support →; we must use Π-types throughout. Next, we declare functions to form vectors by

```
nil : Vector zero
cons : Πn:Nat. Πx:A. Πv:Vector n. Vector (succ n)
```

allowing us to define a function for forming vectors of length three:

```
one = succ zero;two = succ one;
mkthree = λx:A.λy:A.λz:A.
        cons two z (cons one y (cons zero x nil));
```

The implementation will respond by inferring the type of `mkthree`:

```
mkthree : Πx:A. Πy:A. Πz:A. Vector (succ two)
```

We can now partially apply `mkthree` to two elements of type `A` by

```
a:A; b:A;
mkthree a b;
```

resulting in the response

```
λz:A.
 cons (succ (succ zero)) z
 (cons (succ zero) b (cons zero a nil)) :
      Πz:A.Vector(succ (succ (succ zero)))
```

This response exhibits two weaknesses of the implementation. First, definitions are always expanded in results; this will in practice almost always lead to unreadable outpt. Second, the first arguments to `cons` must be given explicitly and are printed out while they could be inferred from the types of the second arguments. Practical implementations of dependent types overcome both these problems. For instance, in LEGO (Luo and Pollack, 1992), `mkthree` would be defined (in our notation) as

```
mkthree = λx:A.λy:A.λz:A. cons z (cons y (cons x nil));
```

and the response to `mkthree a b` would be

```
λz:A.cons z (cons  b (cons a nil))) :  Πz:A.Vector three
```

For LEGO to know that the first argument to `cons` is implicit we must declare `cons` by

```
Πn|Nat. Πx:A. Πv:Vector n. Vector (succ n)
```

where the bar indicates implicitness for argument synthesis.

Returning to our experimental checker, let us illustrate Σ-types. We declare three types

```
A:∗; B:Πx:A.∗; C:Πx:A.Πy:B x.∗;
```

and define

```
S = Σx:A.Σy:B x.C x y;
```

Supposing

```
a:A; b:B a; c: C a b;
```

then we can form

```
 (a,(b,c:Σy:B a.C a y):S);
```

which is an element of `S`. The first type annotation is actually redundant and the implementation allows one to abbreviate the above by

```
(a,b,c:S)
```

If we declare

```
Q : Πx:S.∗;
x:S; y:Q x;
```

Then the following typecast succeeds

```
y:Q (x.1,x.2.1,x.2.2:S);
```

thus illustrating the built-in surjective pairing.

Here, finally, is the definition of natural numbers in CC:

```
nat = all a:Prop.all z:Prf a.all s:Πx:Prf a.Prf a. a;
```

Note that `Prf` always requires parentheses in the implementation.

We also remark that by default the implementation prints the weak-head normal form of input terms. The β-normal form of a term `t` is printed with the command `Normal t`.

2.10 Further Reading

Dependent type theories have been widely investigated, much of the development building on the pioneering work of Per Martin-Löf. This is not the place for a comprehensive overview of the literature; rather we provide a few pointers into work related to the developments in this chapter.

The Edinburgh Logical Framework and its type system are described in Harper, Honsell, and Plotkin (1993). Our definition of λLF has the same type structure, but omits signatures, and includes declarative equality judgments rather than an untyped equivalence relation. A more complete recent development which also includes equality judgments is in Harper and Pfenning (2004).

Richer type theories than LF are considered in many places. The calculus of constructions was introduced in Coquand and Huet (1988) and further developed in related type theories (Mohring, 1986; Luo, 1994; Pollack, 1994). Algorithms for typechecking with dependent types were first considered by Coquand (1991), Cardelli (1986; 1988b), and also within the closely related AUTOMATH system of de Bruijn (1980).

The best survey of Pure Type Systems remains Barendregt's handbook article (1992), which includes a description of the λ-Cube. Although the definition of PTS is elegant and short, developing the meta-theory for PTSs has been surprisingly challenging. Several important results and improvements have been made since Barendregt's article. For example, Pollack (1994), studied formalization of the meta-theory of PTSs in type theory itself, Poll (1998) established the *expansion postponement* property for the class of normalizing PTSs, and Zwanenburg (1999) studied the addition of subtyping to PTSs.

Type theories which combine inductive types and quantification over kinds, such as CIC, do not permit an easy normalization proof by translation into a simply-typed normalizing sytem as was the case for the pure CC. Therefore, strong normalization must be proved from scratch for those systems. So far only partial proofs have been published for CIC as it is implemented in Coq; the closest work so far is in the recent PhD thesis of Miquel (2001). For UTT as implemented in LEGO, a strong normalization proof is given in Goguen (1994), which introduces the idea of a *typed operational semantics* as a more controlled way of managing reduction in a typed setting.

A topic we have not considered here is the semantics of dependent types. There has been considerable study; see Hofmann (1997b) for a particular approach and some comparison with the literature. Notable contributions include Cartmell (1986), Erhard (1988), Streicher (1991), and Jacobs (1999).