Chapter 7

Linear Type Theory

The distinction between logic and type theory is not always clear in the literature. From the judgmental point of view, the principal judgments of logic are A *is a proposition* (A *prop*) and A *is true* (A *true*). This may be different for richer logics. For example, in temporal logic we may have a basic judgment A *is true at time* t. However, it appears to be a characteristic that the judgments of logic are concerned with the study or propositions and truth.

In type theory, we elevate the concept of *proof* to a primary concept. In constructive logic this is important because proofs give rise to computation under reduction, as discussed in the chapter on linear functional programming (Chapter 6). Therefore, our primary judgment has the form M *is a proof of* A ($M : A$). This has the alternative interpretation M *is an object of type* A. So the principal feature that distinguishes a type theory from a logic is the internal notion of proof. But proofs are programs, so right from the outset, type theory has an internal notion of program and computation which is lacking from logic.

The desire to internalize proofs and computation opens a rich design space for the judgments defining type theory. We will not survey the different possibilities, but we may map out a particular path that is appropriate for linear type theory. We may occasionally mention alternative approaches or possibilities.

7.1 Dependent Types

The foundation of linear type theory was already laid in Section 6.1 where we introduced a *propositional linear type theory* through the notion of proof terms. We call it propositional, because the corresponding logic does not allow quantification.

Propositions	A	$::=$	P	Atoms
			$\mid A_1 \multimap A_2 \mid A_1 \otimes A_2 \mid \mathbf{1}$	Multiplicatives
			$\mid A_1 \& A_2 \mid \top \mid A_1 \oplus A_2 \mid \mathbf{0}$	Additives
			$\mid A \supset B \mid\, !A$	Exponentials

We now reconsider the quantifiers, $\forall x.\ A$ and $\exists x.\ A$. In the first-order linear logic we developed, the quantifiers range over a single (unspecified) domain. We may thus think of first-order logic as the study of quantification independently of any particular domain. This is accomplished by not making any assumptions about the domain of quantification. In contrast, first-order arithmetic arises if we introduce natural numbers and allow quantifiers to range specifically over natural numbers. This suggests to generalize the quantifiers to $\forall x{:}\tau.\ A$ and $\exists x{:}\tau.\ A$, where τ is a type.

In type theory, we may identify types with propositions. Therefore, we may label a quantifier with A instead of inventing a new syntactic category τ of types. Data types, such as the natural numbers, then have to be introduced as new types A together with their introduction and elimination rules. We postpone the discussion of numbers and other data types to Section **??**. Here, we are most interested in understanding the nature of the quantifiers themselves once they are typed.

Universal Quantification. Before, the introduction rule for $\forall x.\ A$ required us to prove $[a/x]A$ for a new parameter a. This stays essentially the same, except that we are allowed to make a typing assumption for a.

$$\frac{\Gamma, a{:}B; \Delta \vdash [a/x]A\ \mathit{true}}{\Gamma; \Delta \vdash \forall x{:}B.\ A\ \mathit{true}}\ \forall\mathrm{I}^a$$

Note that the parameter a is treated in an unrestricted manner. In other words, the object a we assume to exist is *persistent*, it is independent of the state. This avoids the unpleasant situation where a proposition C may talk about an object that no longer exists in the current state as defined via the linear hypotheses. See Exercise **??** for an exploration of this issue.

The rule above is written as part of a logic, and not as part of a type theory. We can obtain the corresponding type-theoretical formulation by adding proof terms. In this case, the proof of the premise is a function that maps an object N of type B to a proof of $[N/x]A$. We write this function as an ordinary λ-abstraction. We will also now use the same name x for the parameter and the bound variable, to remain closer to the traditions functional programming and type theory.

$$\frac{\Gamma, x{:}B; \Delta \vdash M : A}{\Gamma; \Delta \vdash \lambda x{:}B.\ M : \forall x{:}B.\ A}\ \forall\mathrm{I}$$

It is implicit here that x must be new, that is, it may not already be declared in Γ or Δ. This can always be achieved via renaming of the bound variable x in the proof term and the type of the conclusion. Note that the variable x may occur in A. This represents a significant change from the propositional case, where the variables in Γ and Δ occur only in proof terms, but not propositions.

In the corresponding elimination rule we now need to check that the term we use to instantiate the universal quantifier has the correct type. The proof term for the result is simply application.

$$\frac{\Gamma; \Delta \vdash M : \forall x{:}B.\ A \qquad \Gamma; \cdot \vdash N : B}{\Gamma; \Delta \vdash M\, N : [N/x]A}\ \forall E$$

Because $x{:}B$ may be used in an unrestricted manner in the proof of A, the proof of B may not depend on any linear hypotheses.

The local reduction is simply β-reduction, the local expansion is η-expansion. We write these out on the proof terms.

$$\begin{array}{lcl} (\lambda x{:}B.\ M)\, N & \longrightarrow_{\beta} & [N/x]M \\ M : \forall x{:}B.\ A & \longrightarrow_{\eta} & \lambda x{:}B.\ M\, x \end{array}$$

When viewed as a type, the universal quantifier is a *dependent function type*. The word "dependent" refers to the fact that the type of the the result of the function $M : \forall x{:}B.\ A$ *depends* on the actual argument N since it is $[N/x]A$. In type theory this is most often written as $\Pi x{:}B.\ A$ instead of $\forall x{:}B.\ A$. The dependent function type is a generalization of the ordinary function type (see Exercise **??**) and in type theory we usually view $B \supset A$ as an abbreviation for $\forall x{:}B.\ A$ for some x that does not occur free in A. This preserves the property that there is exactly one rule for each form of term. In this section we will also use the form $B \rightarrow A$ instead of $B \supset A$ to emphasize the reading of the propositions as types.

As a first simple example, consider a proof of $\forall x{:}i.\ P(x) \multimap P(x)$ for some arbitrary type i and predicate P on objects of type i.

$$\dfrac{\dfrac{\dfrac{}{x{:}i; u{:}P(x) \vdash u : P(x)}\ u}{x{:}i; \cdot \vdash \hat{\lambda} u{:}P(x).\ u : P(x) \multimap P(x)}\ {\multimap} I}{\cdot; \cdot \vdash \lambda x{:}i.\ \hat{\lambda} u{:}P(x).\ u : \forall x{:}i.\ P(x) \multimap P(x)}\ \forall I$$

This proof is very much in the tradition of first-order logic—the added expressive power of a type theory is not exploited. We will see more examples later, after we have introduced the existential quantifier.

Existential Quantification. There is also a dependent version of existential quantification. For reasons similar to the existential quantifier, the witness N for the truth of $\exists x{:}A.\ B$ is persistent and cannot depend on linear hypotheses.

$$\frac{\Gamma; \cdot \vdash N : A \qquad \Gamma; \Delta \vdash M : [N/x]B}{\Gamma; \Delta \vdash N\ !\!\otimes^{\exists x.\ B}\ M : \exists x{:}A.\ B}\ \exists I$$

Unfortunately, we need to annotate the new term constructor with most of its type in order to guarantee uniqueness. The problem is that even if we know N and $[N/x]B$, we cannot in general reconstruct B uniquely (see Exercise **??**). The notation $N\ !\!\otimes\ M$ is a reminder that pairs of this form are exponential, not additive. In fact, $\exists x{:}A.\ B$ is a dependent generalization of the operator $A \mathbin{!\otimes} B$ defined either by introduction and eliminations or via notational definition as $(!A) \otimes B$. We have already seen the reverse, $A \mathbin{\otimes!} B$ in Section 5.5 and Exercise 5.2.

The corresponding elimination is just a dependent version of the ordinary existential elimination.

$$\frac{\Gamma; \Delta \vdash M : \exists x{:}A.\ B \qquad \Gamma, x{:}A; \Delta', u{:}B \vdash N : C}{\Gamma; (\Delta, \Delta') \vdash \mathsf{let}\ x \mathbin{!\otimes} u = M\ \mathsf{in}\ N : C}\ \exists\mathrm{E}$$

Here, x and u must be new with respect to Γ and Δ'. In particular, they cannot occur in C. Also note that x is unrestricted while u is linear, as expected from the introduction rule.

Again, we write out the local reductions on proof terms.

$$\begin{array}{rcl}
\mathsf{let}\ x \mathbin{!\otimes} u = M_1 \mathbin{!\otimes} M_2\ \mathsf{in}\ N & \longrightarrow_\beta & [M_1/x][M_2/u]N \\
M : A \mathbin{!\otimes} B & \longrightarrow_\eta & \mathsf{let}\ x \mathbin{!\otimes} u = M\ \mathsf{in}\ x \mathbin{!\otimes} u
\end{array}$$

Here we have omitted the superscript on the $!\otimes$ constructor for the sake of brevity.

Type Families. In order for a dependent type to be truly dependent, we must have occurrences of the quantified variables in the body of the type. In the example above we had $\forall x{:}i.\ P(x) \multimap P(x)$. But what is the nature of P? Logically speaking, it is a predicate on objects of type i. Type-theoretically speaking, it is a *type family*. That is, $P(M)$ is a type for every object M of type i. In such a type we call M the *index object*. It should now be clear that well-formedness of types is not longer a trivial issue the way it has been so far. We therefore augment our collections of judgments with a new one, *A is a type* written as $A : \text{type}$.

Since a type such as $P(x) \multimap P(x)$ may depend on some parameters, we consider a hypothetical judgment of the form

$$\Gamma; \cdot \vdash A : \text{type}.$$

There are no linear hypotheses because it not clear what such a judgment would mean, as hinted above. One possible answer has been given by Ishtiaq and Pym [IP98], but we restrict ourselves here to the simpler case.

Most rules for this judgment are entirely straightforward; we show only three representative ones.

$$\frac{\Gamma; \cdot \vdash A : \mathrm{type} \qquad \Gamma; \cdot \vdash B : \mathrm{type}}{\Gamma; \cdot \vdash A \multimap B : \mathrm{type}} \multimap\mathrm{F}$$

$$\frac{\Gamma; \cdot \vdash A : \mathrm{type} \qquad \Gamma, x{:}A; \cdot \vdash B : \mathrm{type}}{\Gamma; \cdot \vdash \forall x{:}A.\ B : \mathrm{type}} \forall\mathrm{F}$$

$$\frac{\Gamma; \cdot \vdash A : \mathrm{type} \qquad \Gamma, x{:}A; \cdot \vdash B : \mathrm{type}}{\Gamma; \cdot \vdash \exists x{:}A.\ B : \mathrm{type}} \exists\mathrm{F}$$

Atomic types $a\ M_1 \ldots M_n$ consist of a type family a indexed by objects $M_1, \ldots, M_n$. We introduce each such family with a separate formation rule. For example,

$$\frac{\Gamma; \cdot \vdash M : i}{\Gamma; \cdot \vdash P\ M : \mathrm{type}} P\mathrm{F}$$

would be the formation rule for the type family P considered above. Later it will be convenient to collect this information in a *signature* instead (see Section **??**).

In the next section we will see some examples of type families from functional programming.

7.2 Dependently Typed Data Structures

One potentially important application of dependent types lies functional programming. Here we can use certain forms of dependent types to capture data structure invariants concisely. As we will see, there are also some obstacles to the practical use of such type systems.

As an example we will use lists with elements of some type A, indexed by their length. This is of practical interest because dependent types may allow us to eliminate bounds checks statically, as demonstrated in [XP98].

Natural Numbers. First, the formation and introduction rules for natural numbers in unary form.

$$\frac{}{\Gamma; \cdot \vdash \mathsf{nat} : \mathrm{type}} \mathsf{nat}\mathrm{F}$$

$$\frac{}{\Gamma; \cdot \vdash 0 : \mathsf{nat}} \mathsf{nat}\mathrm{I}_0 \qquad \frac{\Gamma; \Delta \vdash M : \mathsf{nat}}{\Gamma; \Delta \vdash \mathsf{s}(M) : \mathsf{nat}} \mathsf{nat}\mathrm{I}_1$$

There a two destructors: one a case construct and one operator for iteration. We give these in the schematic form, as new syntax constructors, and as constants.

Schematically, f is defined by cases if it satisfies

$$\begin{array}{rcl} f(0) & = & N_0 \\ f(\mathsf{s}(n)) & = & N_1(n) \end{array}$$

Here, $N_1(n)$ indicates that the object N_1 may depend on n. However, this form does not express linearity conditions. If we write it as a new language constructor, we have (avoiding excessive syntactic sugar)

$$\mathsf{case}^{\mathsf{nat}}(M; N_0, n.\ N_1)$$

with the rule

$$\frac{\Gamma; \Delta \vdash M : \mathsf{nat} \qquad \Gamma; \Delta' \vdash N_0 : C \qquad \Gamma; \Delta', n : \mathsf{nat} \vdash N_1 : C}{\Gamma; \Delta, \Delta' \vdash \mathsf{case}^{\mathsf{nat}}(M; N_0, n.\ N_1) : C}\ \mathsf{nat}\mathrm{E}_{\mathrm{case}}$$

Note that the elimination is additive in character, since exactly one branch of the case will be taken. The local reductions and expansion are simple:

$$\begin{array}{rcl} \mathsf{case}^{\mathsf{nat}}(0; N_0, n.\ N_1) & \longrightarrow_L & N_0 \\ \mathsf{case}^{\mathsf{nat}}(\mathsf{s}(M); N_0, n.\ N_1) & \longrightarrow_L & [M/n]N_1 \\ M : \mathsf{nat} & \longrightarrow_\eta & \mathsf{case}^{\mathsf{nat}}(M; 0, n.\ \mathsf{s}(n)) \end{array}$$

We could also introduce $\mathsf{case}^{\mathsf{nat}}$ as a constant with linear typing. This would violate our orthogonality principle. However, for any given type C we can define a "canonical" higher-order function implementing a case-like construct with the expected operational behavior.

$$\begin{array}{rcl} \mathsf{casenat}_C & : & \mathsf{nat} \multimap (C \,\&\, (\mathsf{nat} \multimap C)) \multimap C \\ & = & \hat{\lambda} n{:}\mathsf{nat}.\ \hat{\lambda} c{:}(C \,\&\, (\mathsf{nat} \multimap C)).\ \mathsf{case}^{\mathsf{nat}}(n; \mathsf{fst}\ c, n.\ \mathsf{snd}\ c\,\hat{}\,n) \end{array}$$

Schematically, f is defined by iteration if it satisfies

$$\begin{array}{rcl} f(0) & = & N_0 \\ f(\mathsf{s}(n)) & = & N_1(f(n)) \end{array}$$

Here, N_1 can refer to the value of f on the predecessor, but not to n itself. As a language constructor:

$$\mathsf{it}^{\mathsf{nat}}(M; N_0, r.\ N_1)$$

with the rule

$$\frac{\Gamma; \Delta \vdash M : \mathsf{nat} \qquad \Gamma; \cdot \vdash N_0 : C \qquad \Gamma; r{:}C \vdash N_1 : C}{\Gamma; \Delta \vdash \mathsf{it}^{\mathsf{nat}}(M, N_0, r.\ N_1) : C}\ \mathsf{nat}\mathrm{E}_{\mathrm{it}}$$

Note that N_1 will be used as many times as M indicates, and can therefore not depend on any linear variables. We therefore also do not allow the branch for 0 to depend on linear variables. In this special data type this would be possible, since each natural number contains exactly one base case (see Exercise **??**).

$$\begin{array}{rcl} \mathsf{it}^{\mathsf{nat}}(0; N_0, r.\, N_1) & \longrightarrow_L & N_0 \\ \mathsf{it}^{\mathsf{nat}}(\mathsf{s}(M); N_0, r.\, N_1) & \longrightarrow_L & [\mathsf{it}^{\mathsf{nat}}(M; N_0, r.\, N_1)/r]N_1 \\ M : \mathsf{nat} & \longrightarrow_\eta & \mathsf{it}^{\mathsf{nat}}(M; 0, r.\, s(r)) \end{array}$$

From this we can define a canonical higher-order function for each result type C implementing iteration.

$$\begin{array}{rcl} \mathsf{itnat}_C & : & \mathsf{nat} \multimap (C \& (C \multimap C)) \to C \\ & = & \hat{\lambda} n{:}\mathsf{nat}.\ \lambda c{:}C \& (C \multimap C).\ \mathsf{it}^{\mathsf{nat}}(n; \mathsf{fst}\, c, r.\ \mathsf{snd}\, c \,\hat{}\, r) \end{array}$$

More interesting is to define an operator for primitive recursion. Note the linear typing, which requires that at each stage of iteration we either use the result from the recursive call or the predecessor, but not both.

$$\mathsf{recnat}_C \quad : \mathsf{nat} \multimap (C \& ((\mathsf{nat} \& C) \multimap C)) \to C$$

The definition of this recursor is subject of Exercise **??**.

Lists. Next we define the data type of lists. We leave the type of elements open, that is, list is a *type constructor*. In addition, lists are indexed by their length. We therefore have the following formation rule

$$\frac{\Gamma; \cdot \vdash A : \mathrm{type} \qquad \Gamma; \cdot \vdash M : \mathsf{nat}}{\Gamma; \cdot \vdash \mathsf{list}_A(M) : \mathrm{type}}\ \mathsf{listF}$$

There are two introduction rules for lists: one for the empty list (nil) and one for a list constructor (cons). Note that we have to take care to construct the proper index objects. In order to simplify type-checking and the description of the operational semantics, we make the length of the list explicit as an argument to the constructor. In practice, this argument can often be inferred and in many cases does not need to be carried when a program is executed. We indicate this informally by writing this dependent argument as a subscript.

$$\frac{}{\Gamma; \cdot \vdash \mathsf{nil}^A : \mathsf{list}_A(0)}\ \mathsf{list}I_0$$

$$\frac{\Gamma; \cdot \vdash N : \mathsf{nat} \qquad \Gamma; \Delta \vdash H : A \qquad \Gamma; \Delta' \vdash L : \mathsf{list}_A(N)}{\Gamma; \Delta, \Delta' \vdash \mathsf{cons}_N\ H\ L : \mathsf{list}_A(\mathsf{s}(N))}\ \mathsf{list}I_1$$

Again, there are two elimination constructs: one for cases and one for iteration. A function for primitive recursion can be defined.

Schematically, f is defined by cases over a list l if it satisfies:

$$\begin{array}{rcl} f(\mathsf{nil}) & = & N_0 \\ f(\mathsf{cons}_N\, H\, L) & = & N_1(N, H, L) \end{array}$$

Here we supplied an additional argument to N_1 since we cannot statically predict the length of the list L. This also complicates the typing rule for the case construct, in addition to the linearity conditions.

$$\mathsf{case}^{\mathsf{list}}(M; N_0, n.\ h.\ l.\ N_1)$$

with the rule

$$\frac{\begin{array}{l}\Gamma; \Delta \vdash M : \mathsf{list}_A(N) \\ \Gamma; \Delta' \vdash N_0 : C(0) \\ \Gamma, n{:}\mathsf{nat}; \Delta', h{:}A, l{:}\mathsf{list}_A(n) \vdash N_1 : C(\mathsf{s}(n))\end{array}}{\Gamma; \Delta, \Delta' \vdash \mathsf{it}^{\mathsf{list}}(M; N_0, n.\ h.\ l.\ N_1) : C(N)}\ \mathsf{nat}\mathrm{E}_{\mathsf{case}}$$

The novelty in this rule is that C is normally just a type; here it is a type family indexed by a natural number. This is necessary so that, for example

$$\begin{array}{rcl} id_A & : & \forall n{:}\mathsf{nat}.\ \mathsf{list}_A(n) \multimap \mathsf{list}_A(n) \\ & = & \lambda n{:}\mathsf{nat}.\ \lambda l{:}\mathsf{list}_A(n).\ \mathsf{case}^{\mathsf{list}}(M; 0, n.\ h.\ l'.\ \mathsf{cons}_n\, h\, l') \end{array}$$

type-checks. Note that here the first branch has type $\mathsf{list}_A(0)$ and the second branch has type $\mathsf{list}_A(\mathsf{s}(n))$, where n is the length of the list l' which stands for the tail of l. Local reduction and expansion are straightforward, given the intuition above.

$$\begin{array}{rcl} \mathsf{case}^{\mathsf{list}}(\mathsf{nil}; N_0; n.\ h.\ l.\ N_1) & \longrightarrow_L & N_0 \\ \mathsf{case}^{\mathsf{list}}(\mathsf{cons}_N\, H\, L; N_0; n.\ h.\ l.\ N_1) & \longrightarrow_L & [N/n, H/h, L/l]N_1 \\ M : \mathsf{list}_A(N) & \longrightarrow_\eta & \mathsf{case}^{\mathsf{list}}(M; \mathsf{nil}; n.\ h.\ l.\ \mathsf{cons}_n\, h\, l) \end{array}$$

If we write a higher-order case function it would have type

$$\mathsf{caselist}_{A,C} : \forall m{:}\mathsf{nat}.\ \mathsf{list}_A(m) \multimap (C(0) \,\&\, (\forall n{:}\mathsf{nat}.\ A \otimes \mathsf{list}_A(n) \multimap C(\mathsf{s}(n)))) \multimap C(m)$$

The iteration constructs for lists follows a similar idea. Schematically, f is defined by iteration over a list if it satisfies

$$\begin{array}{rcl} f(\mathsf{nil}) & = & N_0 \\ f(\mathsf{cons}_N\, H\, L) & = & N_1(N, H, f(L)) \end{array}$$

As a language constructor:

$$\mathsf{it}^{\mathsf{list}}(M; N_0, n.\ h.\ r.\ N_1)$$

with the rule

$$\frac{\begin{array}{l}\Gamma; \Delta \vdash M : \mathsf{list}_A(N) \\ \Gamma; \cdot \vdash N_0 : C(0) \\ \Gamma, n{:}\mathsf{nat}; h{:}A, r{:}C(n) \vdash N_1 : C(\mathsf{s}(n))\end{array}}{\Gamma; \Delta \vdash \mathsf{it}^{\mathsf{list}}(M; N_0, n.\ h.\ r.\ N_1) : C(N)}\ \mathsf{nat}\mathrm{E}_{\mathsf{it}}$$

Local reduction and expansion present no new ideas.

$$\begin{array}{rcl}
\mathsf{it}^{\mathsf{list}}(\mathsf{nil}; N_0; n.\ h.\ r.\ N_1) & \longrightarrow_L & N_0 \\
\mathsf{it}^{\mathsf{list}}(\mathsf{cons}_N\ H\ L; N_0; n.\ h.\ r.\ N_1) & \longrightarrow_L & [N/n, H/h, \mathsf{it}^{\mathsf{list}}(L; N_0; n.\ h.\ r.\ N_1)/r]N_1 \\
M : \mathsf{list}_A(N) & \longrightarrow_\eta & \mathsf{it}^{\mathsf{list}}(M; \mathsf{nil}; n.\ h.\ r.\ \mathsf{cons}_n\ h\ r)
\end{array}$$

We can then define the following higher-order constant.

$$\mathsf{itlist}_{A,C} \quad : \quad \forall m{:}\mathsf{nat}.\ \mathsf{list}_A(m) \multimap (C(0) \& (\forall n{:}\mathsf{nat}.\ C(n) \multimap C(\mathsf{s}(n)))) \to C(m)$$

Note that, once again, C is a type family indexed by a natural number.

We now consider some functions on lists and their types. When we append two lists, the length of the resulting list will be the sum of the lengths of the two lists. To express this in our language, we first program addition.

$$\begin{array}{rcl}
\mathit{plus} & : & \mathsf{nat} \multimap \mathsf{nat} \multimap \mathsf{nat} \\
& = & \hat{\lambda} x{:}\mathsf{nat}.\ \mathsf{it}^{\mathsf{nat}}(x; \hat{\lambda} y{:}\mathsf{nat}.\ y; r.\ \hat{\lambda} y{:}\mathsf{nat}.\ \mathsf{s}(r\ \hat{}\ y))
\end{array}$$

Note that the iteration is at type $\mathsf{nat} \multimap \mathsf{nat}$ so that $r : \mathsf{nat} \multimap \mathsf{nat}$ in the second branch.

$$\begin{array}{rcl}
\mathit{append}_A & : & \forall n{:}\mathsf{nat}.\ \forall m{:}\mathsf{nat}.\ \mathsf{list}_A(n) \multimap \mathsf{list}_A(m) \multimap \mathsf{list}_A(\mathit{plus}\ n\ m) \\
& = & \lambda n{:}\mathsf{nat}.\ \lambda m{:}\mathsf{nat}.\ \hat{\lambda} l{:}\mathsf{list}_A(n). \\
& & \quad \mathsf{it}^{\mathsf{list}}(l;\ \hat{\lambda} k{:}\mathsf{list}_A(m).\ k, \\
& & \qquad\qquad p.\ h.\ r.\ \hat{\lambda} k{:}\mathsf{list}_A(m).\ \mathsf{cons}_p\ h\ (r\ \hat{}\ k))
\end{array}$$

This example illustrates an important property of dependent type systems: the type of the function append_A contains a defined function (in this case plus). This means we need to compute in the language in order to obtain the type of an expression. For example

$$\mathit{append}_A\ 0\ 0\ \mathsf{nil}_A\ \mathsf{nil}_A \quad : \mathsf{list}_A(\mathit{plus}\ 0\ 0)$$

by the rules for dependent types. But the result of evaluating this expression will be nil_A which has type $\mathsf{list}_A(0)$. Fortunately, $\mathit{plus}\ 0\ 0$ evaluates to 0. In general, however, we will not be able to obtain the type of an expression purely by computation, but we have to employ equality reasoning. This is because the arguments to a function will not be known at the time of type-checking. For example, to type-check the definition of append_A above, we obtain the type

$$\mathsf{list}_A(\mathsf{s}(\mathit{plus}\ p\ m))$$

for N_1 (the second branch) for two parameters p and m. The typing rules for $\mathsf{it}^{\mathsf{nat}}$ require this branch to have type

$$\mathsf{list}_A(\mathit{plus}\ (\mathsf{s}(p))\ m)$$

since the type family $C(n) = \mathsf{list}_A(\mathit{plus}\ n\ m)$ and the result of the second branch should have type $C(\mathsf{s}(n))$.

Fortunately, in this case, we can obtain the first type from the second essentially by some local reductions. In order for dependent type theories to be useful for functional programming we therefore need the rule of *type conversion*

$$\frac{\Gamma; \Delta \vdash M : A \qquad \Gamma; \cdot \vdash A = B : \text{type}}{\Gamma; \Delta \vdash M : B}\,\mathsf{conv}$$

where the $A = B$: type is a new judgment of *definitional equality*. At the very least the example above suggests that if M and N can be related by a sequence of reduction steps applied somewhere in M and N, then they should be considered equal. If this question is decidable is not at all clear and has to be reconsidered for each type theory in detail.

In an *extensional type theory*, such as the one underlying Nuprl [C+86], we allow essentially arbitrarily complex reasoning (including induction) in order to prove that $A = B$: type. This means that conversion and also type-checking in general are undecidable—the judgment of the type theory are no longer analytic, but synthetic. This cast some doubt on the use of this type theory as a functional programming language.

In an *intensional type theory*, such as later type theories developed by Martin-Löf [ML80, NPS90, CNSvS94], definitional equality is kept weak and thereby decidable. The main judgment of the type theory remains analytic, which is desirable in the design of a programming language.

However, there is also a price to pay for a weak notion of equality. It means that sometimes we will be unable to type-check simple (and correct) functions, because the reason for their correctness requires inductive reasoning. A simple example might be

$$\mathit{rev}_A \quad : \quad \forall n{:}\mathsf{nat}.\ \mathsf{list}_A(n) \multimap \mathsf{list}_A(n)$$

which reverses the elements of a list. Depending on its precise formulation, we may need to know, for example, that

$$n{:}\mathsf{nat}, m{:}\mathsf{nat}; \cdot \vdash \mathsf{list}_A(\mathit{plus}\, n\, m) = \mathsf{list}_A(\mathit{plus}\, m\, n) : \text{type}$$

which will not be the case in an intensional type theory.

We circumvent this problem by introducing a new proposition (or type, depending on one's point of view) $\mathit{eq}\, N\, M$ for index objects N and M of type nat. Objects of this types are explicit proofs of equality for natural numbers and should admit induction (which is beyond the scope of these notes). The type of *rev* would then be rewritten as

$$\mathit{rev}_A \quad : \quad \forall n{:}\mathsf{nat}.\ \mathsf{list}_A(n) \multimap \exists m{:}\mathsf{nat}.\ \exists p{:}\mathit{eq}\, n\, m.\ \mathsf{list}_A(m)$$

In other words, in order to have decidable type-checking, we sometimes need to make correctness proofs explicit. This is not surprising, given the experience that correctness proofs can often be difficult.

In practical languages such as ML, it is therefore difficult to include dependent types. The approach taken in [Xi98] is to restrict index objects to be drawn from a domain with a decidable equality theory. This appears to be a reasonable compromise that can make the expressive power of dependent types available to the programmer without sacrificing decidable and efficient type-checking.

Bibliography

[ABCJ94] D. Albrecht, F. Bäuerle, J. N. Crossley, and J. S. Jeavons. Curry-Howard terms for linear logic. In ??, editor, *Logic Colloquium '94*, pages ??–?? ??, 1994.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[ACS98] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchrounous π-calculus. *Theoretical Computer Science*, 195(2):291–423, 1998.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[And92] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):197–347, 1992.

[AP91] J.-M. Andreoli and R. Pareschi. Logic programming with sequent systems: A linear logic approach. In P. Schröder-Heister, editor, *Proceedings of Workshop to Extensions of Logic Programming, Tübingen, 1989*, pages 1–30. Springer-Verlag LNAI 475, 1991.

[AS01] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. *Submitted*, 2001. A previous version presented at LICS'00.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.

[Bib86] Wolfgang Bibel. A deductive solution for plan generation. *New Generation Computing*, 4:115–132, 1986.

[Bie94] G. Bierman. On intuitionistic linear logic. Technical Report 346, University of Cambridge, Computer Laboratory, August 1994. Revised version of PhD thesis.

[BS92] G. Bellin and P. J. Scott. On the π-calculus and linear logic. Manuscript, 1992.

[C⁺86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[Cer95] Iliano Cervesato. Petri nets and linear logic: a case study for logic programming. In M. Alpuente and M.I. Sessa, editors, *Proceedings of the Joint Conference on Declarative Programming (GULP-PRODE'95)*, pages 313–318, Marina di Vietri, Italy, September 1995. Palladio Press.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic.* North-Holland, Amsterdam, 1958.

[CHP00] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. *Theoretical Computer Science*, 232(1–2):133–163, February 2000. Special issue on Proof Search in Type-Theoretic Languages, D. Galmiche and D. Pym, editors.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion.* Princeton University Press, Princeton, New Jersey, 1941.

[CNSvS94] Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994.

[Doš93] Kosta Došen. A historical introduction to substructural logics. In Peter Schroeder-Heister and Kosta Došen, editors, *Substructural Logics*, pages 1–30. Clarendon Press, Oxford, England, 1993.

[Gen35] Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. Translated under the title *Investigations into Logical Deductions* in [Sza69].

[Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir93] J.-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[Her30] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la Société des Sciences et de Lettres de Varsovic*, 33, 1930.

[HM94] Joshua Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. A preliminary version appeared in the Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science, pages 32–42, Amsterdam, The Netherlands, July 1991.

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[Hof00a] Martin Hofmann. Linear types and non-size increasing polynomial time computation. *Theoretical Computer Science*, 2000. To appear. A previous version was presented at LICS'99.

[Hof00b] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, November 2000. To appear. A previous version was presented as ESOP'00.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980. Hitherto unpublished note of 1969, rearranged, corrected, and annotated by Howard.

[HP97] James Harland and David Pym. Resource-distribution via boolean constraints. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (CADE-14)*, pages 222–236, Townsville, Australia, July 1997. Springer-Verlag LNAI 1249.

[HT91] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, pages 133–147, Geneva, Switzerland, July 1991. Springer-Verlag LNCS 512.

[Hue76] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[IP98] Samin Ishtiaq and David Pym. A relevant analysis of natural deduction. *Journal of Logic and Computation*, 8(6):809–838, 1998.

[Kni89] Kevin Knight. Unification: A multi-disciplinary survey. *ACM Computing Surveys*, 2(1):93–124, March 1989.

[Lin92] P. Lincoln. Linear logic. *ACM SIGACT Notices*, 23(2):29–37, Spring 1992.

[Mil92] D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the Workshop on Extensions of Logic Programming*, pages 242–265. Springer-Verlag LNCS 660, 1992.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML94] Per Martin-Löf. Analytic and synthetic judgements in type theory. In Paolo Parrini, editor, *Kant and Contemporary Epistemology*, pages 87–99. Kluwer Academic Publishers, 1994.

[ML96] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996.

[MM76] Alberto Martelli and Ugo Montanari. Unification in linear time and space: A structured presentation. Internal Report B76-16, Istituto di Elaborazione delle Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy, July 1976.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MOM91] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic through categories: A survey. *Journal on Foundations of Computer Science*, 2(4):297–399, December 1991.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

[PD01] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11:511–540, 2001. Notes to an invited talk at the *Workshop on Intuitionistic Modal Logics and Applications* (IMLA'99), Trento, Italy, July 1999.

[Pra65] Dag Prawitz. *Natural Deduction.* Almquist & Wiksell, Stockholm, 1965.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Rob71] J. A. Robinson. Computational logic: The unification computation. *Machine Intelligence*, 6:63–72, 1971.

[Sce93] A. Scedrov. A brief guide to linear logic. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 377–394. World Scientific Publishing Company, 1993. Also in Bulletin of the European Association for Theoretical Computer Science, volume 41, pages 154–165.

[SHD93] Peter Schroeder-Heister and Kosta Došen, editors. *Substructural Logics*. Number 2 in Studies in Logic and Computation. Clarendon Press, Oxford, England, 1993.

[Sta85] Richard Statman. Logical relations and the typed λ-calculus. *Information and Control*, 65:85–97, 1985.

[Sza69] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*. North-Holland Publishing Co., Amsterdam, 1969.

[Tai67] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal Of Symbolic Logic*, 32:198–212, 1967.

[Tro92] A. S. Troelstra. *Lectures on Linear Logic*. CSLI Lecture Notes 29, Center for the Study of Language and Information, Stanford, California, 1992.

[Tro93] A. S. Troelstra. Natural deduction for intuitionistic linear logic. Prepublication Series for Mathematical Logic and Foundations ML-93-09, Institute for Language, Logic and Computation, University of Amsterdam, 1993.

[WW01] David Walker and Kevin Watkins. On linear types and regions. In *Proceedings of the International Conference on Functional Programming (ICFP'01)*. ACM Press, September 2001.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Department of Mathematical Sciences, Carnegie Mellon University, December 1998.

[XP98] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98)*, pages 249–257, Montreal, Canada, June 1998. ACM Press.