15-819K: Logic Programming

Lecture 18

# Proof Terms

Frank Pfenning

October 31, 2006

In this lecture we will substantiate an earlier claim that logic programming not only permits the representation of specifications and the implementation of algorithm, but also the realization of proofs of correctness of algorithms. We will do so by first showing how *deductions* can be represented as terms, so-called *proof terms*. We also discuss the problems of checking the validity of deductions, which amounts to type checking the terms that represent them.

## 18.1 Deductions as Terms

In logic programming we think of computation as proof search. However, so far search only returns either success together with an answer substitution, fails, or diverges. In the case of success it is reasonable to expect that the logic programming engine could also return a deduction of the instantiated goal. But this raises the question of how to represent deductions. In the traditional logic programming literature we will find ideas such as a list of the rules that have been applied to solve a goal. There is, however, a much better answer. We can think of an inference rule as a *function* which takes deductions of the premisses to a deduction of the conclusion. Such a function is a constructor for proof terms. For example, when interpreting

$$\frac{}{\mathsf{plus}(\mathsf{z}, N, N)}\ \mathsf{pz} \qquad \frac{\mathsf{plus}(M, N, P)}{\mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P))}\ \mathsf{ps}$$

we extract two constructors

$$\begin{array}{lcl} \mathsf{pz} & : & \mathsf{plus}(\mathsf{z}, N, N) \\ \mathsf{ps} & : & \mathsf{plus}(M, N, P) \rightarrow \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)). \end{array}$$

The only unusual thing about these constructors is their type. The type of $\mathsf{pz}$, for example, is a *proposition*.

The idea that *propositions* can be *types* is a crucial observation of the Curry-Howard isomorphism in functional programming that also identifies proofs (of a proposition) with programs (of the correspending type). Here, the correspondence of propositions with types is still perfect, but proofs are *not* programs (which, instead, or are given by inference rules).

As an example of how a complete proof is interpreted as a term, consider the computation of $2 + 2 = 4$.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{plus}(0,2,2)}\,\mathsf{pz}}{\mathsf{plus}(1,2,3)}\,\mathsf{ps}}{\mathsf{plus}(2,2,4)}\,\mathsf{ps}$$

Here we have abbreviated $\mathsf{z} = 0, \mathsf{s}(\mathsf{z}) = 1, \ldots$. This deduction becomes the very simple term

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})).$$

Our typing judgment should be such that

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(2,2,4)$$

so that a proposition acts as the type of its proofs.

As a slightly more complex example, consider multiplication

$$\dfrac{}{\mathsf{times}(\mathsf{z}, N, \mathsf{z})}\,\mathsf{tz} \qquad \dfrac{\mathsf{times}(M, N, P) \quad \mathsf{plus}(P, N, Q)}{\mathsf{times}(\mathsf{s}(M), N, Q)}\,\mathsf{ts}$$

which yields the following constructors

$$\begin{array}{lcl} \mathsf{tz} & : & \mathsf{times}(\mathsf{z}, N, \mathsf{z}) \\ \mathsf{ts} & : & \mathsf{times}(M, N, P), \mathsf{times}(P, N, Q) \to \mathsf{times}(M, N, Q). \end{array}$$

Now the deduction that $2 * 2 = 4$, namely

$$\dfrac{\dfrac{\dfrac{}{\mathsf{times}(0,2,0)}\,\mathsf{tz} \quad \dfrac{}{\mathsf{plus}(0,2,2)}\,\mathsf{pz}}{\mathsf{times}(1,2,2)}\,\mathsf{ts} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{plus}(0,2,2)}\,\mathsf{pz}}{\mathsf{plus}(1,2,3)}\,\mathsf{ps}}{\mathsf{plus}(2,2,4)}\,\mathsf{ps}}{\mathsf{times}(2,2,4)}\,\mathsf{ts}$$

becomes the term

$$\mathsf{ts}(\mathsf{ts}(\mathsf{tz}, \mathsf{pz}), \mathsf{ps}(\mathsf{ps}(\mathsf{pz}))) : \mathsf{times}(2, 2, 4).$$

The tree structure of the deduction is reflected in the corresponding tree structure of the term.

## 18.2 Indexed Types

Looking at our example signature,

$$\begin{array}{lcl} \mathsf{pz} & : & \mathsf{plus}(\mathsf{z}, N, N) \\ \mathsf{ps} & : & \mathsf{plus}(M, N, P) \rightarrow \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \\ \\ \mathsf{tz} & : & \mathsf{times}(\mathsf{z}, N, \mathsf{z}) \\ \mathsf{ts} & : & \mathsf{times}(M, N, P), \mathsf{times}(P, N, Q) \rightarrow \mathsf{times}(M, N, Q) \end{array}$$

we can observe a new phenomenon. The types of our constructors contain *terms*, both constants (such as $\mathsf{z}$) and variables (such as $M$, $N$, or $P$). We say that $\mathsf{plus}$ is a *type family indexed* by terms. In general, under the propositions-as-types interpretation, predicates are interpreted type families indexed by terms.

The free variables in the declarations are interpreted *schematically*, just like in inference rules. So $\mathsf{pz}$ is really a family of constants, indexed by $N$. This has some interesting consequences. For example, we found earlier that

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(2, 2, 4).$$

However, we can also check that

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(2, 3, 5).$$

In fact, our type system will admit a most general type:

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(\mathsf{s}(\mathsf{s}(\mathsf{z})), N, \mathsf{s}(\mathsf{s}(N))).$$

This schematic type captures all types of the term on the left, because any type for $\mathsf{ps}(\mathsf{ps}(\mathsf{pz}))$ is an instance of $\mathsf{plus}(\mathsf{s}(\mathsf{s}(\mathsf{z})), N, \mathsf{s}(\mathsf{s}(N)))$.

## 18.3 Typing Rules

In order to write down the typing rules, it is convenient to make quantification over schematic variables in a indexed type declaration explicit. We write $\forall x{:}\tau$ for quantification over a single variable, and following our general notational convention, $\forall \mathbf{x}{:}\boldsymbol{\tau}$ for a sequence of quantifiers. We have already introduced quantified propositions, so we emphasize its role as quantifying over the schematic variables of a proposition viewed as a type. The example of addition would be written as

$$\begin{array}{lcl} \mathsf{z} & : & \mathsf{nat} \\ \mathsf{s} & : & \mathsf{nat} \to \mathsf{nat} \\ \\ \mathsf{pz} & : & \forall N{:}\mathsf{nat}.\, \mathsf{plus}(\mathsf{z}, N, N) \\ \mathsf{ps} & : & \forall M{:}\mathsf{nat}.\, \forall N{:}\mathsf{nat}.\, \forall P{:}\mathsf{nat}.\, \mathsf{plus}(M, N, P) \to \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \end{array}$$

We call such explicitly quantified types *dependent types*. Unlike other formulations (for example, in the LF logical framework), but similarly to our treatment of polymorphism, the quantifiers do not affect the term language. We write '$\forall$' to emphasize the logical reading of the quantifiers; in a fully dependent type theory they would be written as '$\Pi$'.

There are a many of similarities between polymorphic and dependent types. We will see that in addition to the notation, also the typing rules are analogous. Nevertheless, they are different concepts: polymorphism quantifies over types, while dependency quantifies over terms. We review the rule for polymorphic typing.

$$\frac{\begin{array}{l}\mathrm{dom}(\hat{\theta}) = \boldsymbol{\alpha} \\ f : \forall \boldsymbol{\alpha}.\, \boldsymbol{\sigma} \to \tau \in \Sigma \quad\quad \Delta \vdash \mathbf{t} : \boldsymbol{\sigma}\hat{\theta}\end{array}}{\Delta \vdash f(\mathbf{t}) : \tau\hat{\theta}}$$

Recall that $\hat{\theta}$ is a substitution of types for type variables, and that $\Delta$ contains declarations $x{:}\tau$ as well as $\alpha$ *type*.

The rule for dependent types is analogous, using ordinary substitutions instead of type substitution.

$$\frac{\begin{array}{l}\mathrm{dom}(\theta) = \mathbf{x} \\ c : \forall \mathbf{x}{:}\boldsymbol{\tau}.\, \mathbf{Q} \to P \in \Sigma \quad\quad \Delta \vdash \mathbf{t} : \mathbf{Q}\theta\end{array}}{\Delta \vdash c(\mathbf{t}) : P\theta}$$

We have written $P$ and $\mathbf{Q}$ instead of $\tau$ and $\boldsymbol{\sigma}$ to emphasize the interpretation of the types as propositions.

If we require $\theta$ to be a ground substitution (that is, $\mathrm{cod}(\theta) = \emptyset$), then we can use this rule to determine ground typings such as

$$\mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(2, 2, 4).$$

If we allow free variables, that is, $\mathrm{cod}(\theta) = \mathrm{dom}(\Delta)$, then we can write out schematic typings, such as

$$n{:}\mathsf{nat} \vdash \mathsf{ps}(\mathsf{ps}(\mathsf{pz})) : \mathsf{plus}(\mathsf{s}(\mathsf{s}(\mathsf{z})), n, \mathsf{s}(\mathsf{s}(n))).$$

In either case we want the substitution to be well-typed. In the presence of dependent types, the formalization of this would lead us a bit far afield, so we can think of it just as before: we always substitute a term of type $\tau$ for a variable of type $\tau$.

## 18.4 Type Checking

Ordinarily in logic programming a query is simply a proposition and the result is an answer substitution for its free variables. When we have proof terms we can also ask if a given term constitutes a proof of a given proposition. We might write this as

```
?- t : P.
```

where $t$ is a term representing a purported proof and $P$ is a goal proposition. From the typing rule we can see that type-checking comes down to unification. We can make this more explicit by rewriting the rule:

$$\frac{\begin{array}{l}\mathrm{dom}(\rho) = \mathbf{x} \\ c : \forall \mathbf{x}{:}\boldsymbol{\tau}.\, \mathbf{Q} \to P' \in \Sigma \qquad P'\rho \doteq P \mid \theta \quad \Delta \vdash \mathbf{t} : \mathbf{Q}\rho\theta\end{array}}{\Delta \vdash c(\mathbf{t}) : P}$$

Here $\rho$ is a renaming substituting generating fresh logic variables for the bound variables $\mathbf{x}$.

Because a constant has at most one declaration in a signature, and unification returns a unique most general unifier, the type-checking process is entirely deterministic and will always either fail (in which case there is no type) or succeed. We can even leave $P$ as a variable and obtain the most general type for a given term $t$.

Checking that a given term represents a valid proof can be useful in a number of situations. Some practical scenarios where this has been applied is *proof-carrying code* and *proof-carrying authorization*. Proof-carrying code is the idea that we can equip a piece of mobile code with a proof that it is safe to execute. A code recipient can check the validity of the proof against the code and then run the code without further run-time checks. Proof-carrying authorization is a similar idea, except that the proof is used to convince a resource monitor that a client is authorized for access. Please see the section on historical notes for some references on these applications of logic programming.

## 18.5 Proof Search

Coming back to proof search: we would like to instrument our interpreter so it returns a proof term (as well as an answer substitution) when it succeeds. But the exact rule for type-checking with a slightly different interpretation on modes, will serve that purpose.

$$\frac{\begin{array}{l}\mathrm{dom}(\rho) = \mathbf{x} \\ c : \forall \mathbf{x}{:}\boldsymbol{\tau}.\, \mathbf{Q} \to P' \in \Sigma \qquad P'\rho \doteq P \mid \theta \quad \Delta \vdash \mathbf{t} : \mathbf{Q}\rho\theta\end{array}}{\Delta \vdash c(\mathbf{t}) : P}$$

Above, we thought of $c(\mathbf{t})$ and therefore $\mathbf{t}$ as given input, so this was a rule for type-checking. Now we think of $\mathbf{t}$ as an output, produced by proof search for the premiss, which then allows us to construct $c(\mathbf{t})$ as an output in the conclusion. Now the rule is non-deterministic since we do not know which rule for a given atomic predicate to apply, but for a given proof we will be able to construct a proof term as an output.

We have not addressed here if ordinary untyped unification will be sufficient for program execution (or, indeed, type-checking), or if unification needs to be changed in order to take typing into account. After a considerable amount of technical work, we were able to show in the case of polymorphism that function symbols needed to be type preserving and clause heads parametric for untyped unification to suffice. If we explicitly stratify our language so that in a declaration $c : \forall \mathbf{x}{:}\boldsymbol{\tau}.\, \mathbf{Q} \to P'$ all the types $\boldsymbol{\tau}$ have no variables then the property still holds for well-typed queries; otherwise it may not (see Exercise 18.1).

## 18.6 Meta-Theoretic Proofs as Relations

We now take a further step, fully identifying types with propositions. This means that quantifiers in clauses can now range over *deductions*, and we can specify relations between deductions. Deductions have now become first-class.

There are several uses for first-class deductions. One is that we can now implement theorem provers or decision procedures in a way that intrinsically guarantees the validity of generated proof objects.

Another application is the implementation of proofs *about* the predicates that make up logic programs. To illustrate this, we consider the proof that the sum of two even numbers is even. We review the definitions:

$$\frac{}{\mathsf{even}(\mathsf{z})}\,\mathsf{ez} \qquad \frac{\mathsf{even}(N)}{\mathsf{even}(\mathsf{s}(\mathsf{s}(N)))}\,\mathsf{ess}$$

so that the type declarations for proof constructors are

$$\begin{array}{lcl} \mathsf{ez} & : & \mathsf{even}(\mathsf{z}) \\ \mathsf{ess} & : & \mathsf{even}(N) \rightarrow \mathsf{even}(\mathsf{s}(\mathsf{s}(N))) \\ \\ \mathsf{pz} & : & \mathsf{plus}(\mathsf{z}, N, N) \\ \mathsf{ps} & : & \mathsf{plus}(M, N, P) \rightarrow \mathsf{plus}(\mathsf{s}(M), N, \mathsf{s}(P)) \end{array}$$

**Theorem 18.1** *For any $m$, $n$, and $p$, if* $\mathsf{even}(m)$, $\mathsf{even}(n)$, *and* $\mathsf{plus}(m, n, p)$ *then* $\mathsf{even}(p)$.

**Proof:** By induction on the structure of the deduction $\mathcal{D}$ of $\mathsf{even}(m)$.

**Case:** $\mathcal{D} = \dfrac{}{\mathsf{even}(\mathsf{z})}$ where $m = \mathsf{z}$.

| | |
|---|---|
| $\mathsf{even}(n)$ | Assumption |
| $\mathsf{plus}(\mathsf{z}, n, p)$ | Assumption |
| $n = p$ | By inversion |
| $\mathsf{even}(p)$ | Since $n = p$ |

**Case:** $\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ \mathsf{even}(m')\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(m')))}$ where $m = \mathsf{s}(\mathsf{s}(m'))$.

| | |
|---|---|
| $\mathsf{plus}(\mathsf{s}(\mathsf{s}(m')), n, p)$ | Assumption |
| $\mathsf{plus}(\mathsf{s}(m'), n, p')$ with $p = \mathsf{s}(p')$ | By inversion |
| $\mathsf{plus}(m', n, p'')$ with $p' = \mathsf{s}(p'')$ | By inversion |
| $\mathsf{even}(p'')$ | By ind. hyp. |
| $\mathsf{even}(\mathsf{s}(\mathsf{s}(p'')))$ | By rule |

□

The theorem and its proof involves four deductions:

$$\begin{array}{cccc} \mathcal{D} & \mathcal{E} & \mathcal{F} & \mathcal{G} \\ \mathsf{even}(m) & \mathsf{even}(n) & \mathsf{plus}(m,n,p) & \mathsf{even}(p) \end{array}$$

The theorem states that for any derivations $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{F}$ there exists a deduction $\mathcal{G}$. Using our newfound notation for proof terms we can write this as

> *For every $m$, $n$, and $p$, and for every $D : \mathsf{even}(m)$, $E : \mathsf{even}(n)$, and $F : \mathsf{plus}(m,n,p)$ there exists $G : \mathsf{even}(p)$.*

If this course were about functional programming, we would ensure that this theorem holds by exhibing a *total function*

$$\mathsf{eee} : \mathsf{even}(M) \to \mathsf{even}(N) \to \mathsf{plus}(M,N,P) \to \mathsf{even}(P).$$

It is important that the function be total so that it is guaranteed to generate an output deduction of $\mathsf{even}(P)$ for any combination of input deductions, thereby witnessing its truth.

In logic programming, such functions are not at our disposal. But we can represent the same information as a *total relation*

$$\mathsf{eee} : \mathsf{even}(M), \mathsf{even}(N), \mathsf{plus}(M,N,P), \mathsf{even}(P) \to o.$$

The predicate symbol eee represents a four-place relation between deductions, where we consider the first three deductions as inputs and the last one as an output.

In the next lecture we consider in some details what is required to *verify* that this relation represents a meta-theoretic proof of the property that the sum of two even number is even. Before we get to that, let us examine how our careful, but informal proof is translated into a relation. We will try to construct clauses for

$$\mathsf{eee}(D,E,F,G)$$

where $D : \mathsf{even}(M)$, $E : \mathsf{even}(N)$, $F : \mathsf{plus}(M,N,P)$, and $G : \mathsf{even}(P)$. We repeat the proof, analyzing the structure of $D$, $E$, $F$, and $G$. We highlight the incremental construction of clauses for eee in interspersed boxes.

**Case:** $D = \dfrac{}{\mathsf{even}(\mathsf{z})}\,\mathsf{ez}$ where $m = \mathsf{z}$.

At this point we start to construct a clause

$$\mathsf{eee}(\mathsf{ez}, E, F, G)$$

because $D = \mathsf{ez}$, and we do not yet know $E$, $F$, or $G$.

| | |
|---|---|
| $E : \mathsf{even}(n)$ | Assumption |
| $F : \mathsf{plus}(\mathsf{z}, n, p)$ | Assumption |
| $F = \mathsf{pz}$ and $n = p$ | By inversion |

At this point we have some more information, namely $F = \mathsf{pz}$. So the partially constructed clause now is

$$\mathsf{eee}(\mathsf{ez}, E, \mathsf{pz}, G).$$

| | |
|---|---|
| $G = E : \mathsf{even}(p)$ | Since $n = p$ |

Now we see that the output $G$ is equal to the second input $E$.

$$\mathsf{eee}(\mathsf{ez}, E, \mathsf{pz}, E)$$

This completes the construction of this clause. The second argument $E$ is not analyzed and simply returned as $G$.

**Case:** $D = \dfrac{\begin{array}{c}\mathcal{D}' \\ \mathsf{even}(m')\end{array}}{\mathsf{even}(\mathsf{s}(\mathsf{s}(m')))}\ \mathsf{ess}$ where $m = \mathsf{s}(\mathsf{s}(m'))$.

The partially constructed second clause now looks like

$$\mathsf{eee}(\mathsf{ess}(D'), E, F, G).$$

| | |
|---|---|
| $F : \mathsf{plus}(\mathsf{s}(\mathsf{s}(m')), n, p)$ | Assumption |
| $F = \mathsf{ps}(F')$ where $F' : \mathsf{plus}(\mathsf{s}(m'), n, p')$ with $p = \mathsf{s}(p')$ | By inversion |

Now we have

$$\mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(F'), G)$$

replacing $F$ above by $\mathsf{ps}(F')$.

| | |
|---|---|
| $F' = \mathsf{ps}(F'')$ where $F'' : \mathsf{plus}(m', n, p'')$ with $p' = \mathsf{s}(p'')$ | By inversion |

In this step, the third argument has been even further refined to $\mathsf{ps}(\mathsf{ps}(F''))$ which yields

$$\mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(\mathsf{ps}(F'')), G).$$

$G' : \mathsf{even}(p'')$ By ind. hyp. on $D'$, $E$, and $F''$

An appeal to the induction hypothesis corresponds to a recursive call in the definition of eee.

$$\mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(\mathsf{ps}(F'')), G) \leftarrow \mathsf{eee}(D', E, F'', G')$$

The first three arguments of the recursive call correspond to the deductions on which the induction hypothesis is applied, the fourth argument is the returned deduction $G'$. The question of how we construct the $G$ still remains.

$G = \mathsf{ess}(G') : \mathsf{even}(\mathsf{s}(\mathsf{s}(p'')))$ By rule ess applied to $G'$

Now we can fill in the last missing piece by incorporating the definition of $G$.

$$\mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(\mathsf{ps}(F'')), \mathsf{ess}(G')) \leftarrow \mathsf{eee}(D', E, F'', G')$$

In summary, the meta-theoretic proof is represented as the relation eee shown below. We have named the rules defining eee for consistency, even though it seems unlikely we will want to refer to these rules by name.

$$\begin{array}{lcl}
\mathsf{eee} & : & \mathsf{even}(M), \mathsf{even}(N), \mathsf{plus}(M, N, P), \mathsf{even}(P) \to o. \\
\\
\mathsf{eeez} & : & \mathsf{eee}(\mathsf{ez}, E, \mathsf{pz}, E). \\
\mathsf{eeess} & : & \mathsf{eee}(\mathsf{ess}(D'), E, \mathsf{ps}(\mathsf{ps}(F'')), \mathsf{ess}(G')) \leftarrow \mathsf{eee}(D', E, F'', G').
\end{array}$$

Each case in the definition of eee corresponds to a case in the inductive proof, a recursive call corresponds to an appeal to the induction hypothesis. A constructed term on an input argument of the clause head corresponds to a case split on the induction variable or an appeal to inversion. A constructed term in an output position of the clause head corresponds to a rule application to generate the desired deduction.

It is remarkable how compact the representation of the informal proof has become: just one line declaring the relation and two lines defining the relation. This is in contrast to the informal proof which took up 11 lines.

## 18.7 Verifying Proofs of Meta-Theorems

In the previous section we showed how to represent a proof of a theorem about deductions as a relation between proofs. But what does it take to verify that a given relation indeed represents a meta-theoretic proof of a proposed theorem? A full treatment of this question is beyond the scope of this lecture (and probably this course), but meta-logical frameworks such as Twelf can indeed verify this. Twelf decomposes this checking into multiple steps, each making its own contribution to the overall verification.

1. Type checking. This guarantees that if $\mathsf{eee}(D, E, F, G)$ for deductions $D$, $E$, $F$, and $G$ is claimed, all of these are valid deductions. In particular, the output $G$ will be acceptable evidence that $P$ is even because $G : \mathsf{even}(P)$.

2. Mode checking. The mode $\mathsf{eee}(+, +, +, -)$ guarantees that if the inputs are all complete (ground) deductions and the query succeeds, then the output is also a complete (ground) deduction. This is important because a free variable in a proof term represents an unproven subgoal. Such deductions would be insufficient as evidence that $P$ is indeed even.

3. Totality checking. If type checking and mode checking succeeds, we know that $P$ is even if a query $\mathsf{eee}(D, E, F, G)$ succeeds for ground $D$, $E$, $F$ and free variable $G$. All that remains is to show that *all* such queries succeed. We decompose this into two properties.

   (a) Progress checking. For any given combination of input deductions $D$, $E$, and $F$ there is an applicable clause and we either succeed or at least can proceed to a subgoal. Because of this, $\mathsf{eee}(D, E, F, G)$ can never fail.

   (b) Termination checking. Any sequence of recursive calls will terminate. Since queries can not fail (by progress) the only remaining possibility is that they succeed, which is what we needed to verify.

We have already extensively discussed type checking and mode checking. In the next lecture we sketch progress checking. For termination checking we refer to the literature in the historical notes below.

## 18.8 Polymorphism and Dependency

Because of the similarity of polymorphic and dependent types, and because both have different uses, it is tempting to combine the two in a single language. This is indeed possible. For example, if we would like to index a polymorphic list with its length we could define

```
nat : type.
z : nat.
s : nat -> nat.

list : type,nat -> type.
nil : list(A,z).
cons : A,list(A,N) -> list(A,s(N)).
```

In the setting of functional programming, such combinations have been thoroughly investigated, for examples, as fragments of the Calculus of Constructions. In the setting here we are not aware of a thorough study of either type checking or unification. A tricky issue seems to be how much type information must be carried at run-time, during unification, especially if a type variable can be instantiated by a dependent type.

## 18.9 Historical Notes

The notion of proof term originates in the so-called Curry-Howard isomorphism, which was noted for combinatory logic by Curry and Feys in 1956 [8] and for natural deduction by Howard in 1969 [10]. The emphasis in this work is on functional computation, by combinatory reduction or $\beta$-reduction, respectively. The presentation here is much closer to the LF logical framework [9] in which only canonical forms are relevant, and which is entirely based on dependent types. We applied one main simplification: because we restricted ourselves to the Horn fragment of logic, proof terms contain no $\lambda$-abstractions. This in turn allows us to omit $\Pi$-quantifiers without any loss of decidability of type checking, since type inference can be achieved just by ordinary unification.

The combination of polymorphism and dependency is present in the Calculus of Constructions [6]. Various fragments were later analyzed in detail by Barendregt [2], including the combination of polymorphism with dependent types. A modern version of the Calculus of Constructions is the basis for the Coq theorem proving system.[1]

[1] `http://coq.inria.fr/`

The use of explicit proof terms to certify the safety of mobile code goes back to Necula and Lee [13, 12]. They used the LF logical framework with some optimizations for proof representation. Proof-carrying authorization was proposed by Appel and Felten [1] and then realized for the first time by Bauer, Schneider, and Felten [5, 3]. It is now one of the cornerstones of the Grey project [4] for universal distributed access control at Carnegie Mellon University.

The technique of representing proofs of meta-theorems by relations is my own work [14, 15], eventually leading to the Twelf system [17] which has many contributors. Type reconstruction for Twelf was already sketched in the early work [15], followed the first foray into mode checking and termination checking [19] later extended by Pientka [18]. Totality checking [21] was a further development of meta-proofs that were correct by construction proposed by Schürmann [20]. The example in this lecture can be easily verified in Twelf. For more on Twelf see the Twelf home page.[2]

Relational representation of meta-theory has recently been shown to be sufficiently powerful to mechanize the theory of full-scale programming languages (Typed Assembly Language [7] and Standard ML [11]). For further references and a more general introduction to logical frameworks, see [16].

## 18.10 Exercises

**Exercise 18.1** *We can translate the conditions on polymorphism, namely that that term constructors be type-preserving and predicates be parametric, to conditions on dependency. Give such a parallel definition.*

*Further, give examples that show the need for type information to be carried during unification to avoid creating ill-typed terms if these conditions are violated.*

*Finally, discuss which of these, if any, would be acceptable in the case of dependent types.*

**Exercise 18.2** *Revisit the proof that the empty list is the right unit for* append *(Exercise 3.5 from Assignment 2) and represent it formally as a relation between deductions.*

**Exercise 18.3** *Revisit the proof that addition is commutative (Exercise 3.4 from Assignment 2) and represent it formally as a relation between deductions.*

**Exercise 18.4** *Revisit the proof that append is associative (Exercise 3.6 from Assignment 2) and represent it formally as a relation between deductions.*

[2] `http://www.twelf.org/`

**Exercise 18.5** *If we do not want to change the logic programming engine to produce proof objects, we can transform a program globally by adding an additional argument to every predicate, denoting a proof term.*

*Formally define this transformation and prove its correctness.*

## 18.11 References

[1] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the 6th Conference on Computer and Communications Security*, pages 52–62, Singapore, November 1999. ACM Press.

[2] Henk P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, chapter 2, pages 117–309. Oxford University Press, 1992.

[3] Lujo Bauer. *Access Control for the Web via Proof-Carrying Authorization*. PhD thesis, Princeton University, November 2003.

[4] Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Proceedings of the 8th Information Security Conference (ISC'05)*, pages 431–445, Singapore, September 2005. Springer Verlag LNCS 3650.

[5] Lujo Bauer, Michael A. Schneider, and Edward W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, California, August 2002.

[6] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[7] Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. *ACM Transactions on Computational Logic*, 2006. To appear.

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

[9] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[10] 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.

[11] Daniel K. Lee, Karl Crary, and Robert Harper. Mechanizing the metatheory of Standard ML. Technical Report CMU-CS-06-138, Carnegie Mellon University, 2006.

[12] George C. Necula. Proof-carrying code. In Neil D. Jones, editor, *Conference Record of the 24th Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, January 1997. ACM Press.

[13] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI'96)*, pages 229–243, Seattle, Washington, October 1996.

[14] Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *Fourth Annual Symposium on Logic in Computer Science*, pages 313–322, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[15] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[16] Frank Pfenning. Logical frameworks. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 17, pages 1063–1147. Elsevier Science and MIT Press, 2001.

[17] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[18] Brigitte Pientka and Frank Pfenning. Termination and reduction checking in the logical framework. In Carsten Schürmann, editor, *Workshop on Automation of Proofs by Mathematical Induction*, Pittsburgh, Pennsylvania, June 2000.

[19] Ekkehard Rohwedder and Frank Pfenning. Mode and termination checking for higher-order logic programs. In Hanne Riis Nielson, edi-

tor, *Proceedings of the European Symposium on Programming*, pages 296–310, Linköping, Sweden, April 1996. Springer-Verlag LNCS 1058.

[20] Carsten Schürmann. *Automating the Meta Theory of Deductive Systems*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2000. Available as Technical Report CMU-CS-00-146.

[21] Carsten Schürmann and Frank Pfenning. A coverage checking algorithm for LF. In D. Basin and B. Wolff, editors, *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, pages 120–135, Rome, Italy, September 2003. Springer-Verlag LNCS 2758.