

Metalevel Programming in Constructive Type Theory

Robert L. Constable*
Department of Computer Science
Cornell University
Ithaca, NY 14853
`rc@cs.cornell.edu`

Abstract

This paper describes a particular version of constructive type theory, a subset of the one underlying the Nuprl proof development system, and it examines its role as a programming environment. Special attention is given to the property of the theory to talk about itself. This reflective capability is the basis for automating proof development and for metalevel programming.

1 Introduction

A mature theory of computing will be a basis for a rigorous approach to programming. Already we see this in practice; the algorithms we use to solve problems are often those developed in the mathematical theory of algorithms, and they are chosen based on properties proven to hold for them, such as economy of space or time or robustness or generality. Moreover the algorithms are often expressed in a programming notation that is typed, and the programming environment will include a type checker to insure that the types are sensible. The type checker and the program evaluator are parts of the environment that were developed in the mathematical theory of semantics.

It may soon be the case that most parts of the environment will be based on systematic theoretical results. We see more concepts from logic appearing such as formal specification languages, and it is now a smaller step from these disconnected parts, programming notation, evaluator, type checker, specification language, to a full-blown theory whose axioms and rules of inference are as explicit and rigorous as the other components and yet is an integral part of the programming environment. In such a setting more results of computing theory can be *directly* applied in programming. It

*This research supported by NSF grant CCR86-16552 and ONR grant N00014-88K-0490.

seems that constructive type theory is one candidate for such a unified computing theory, perhaps the first candidate. In these notes we will examine certain aspects of programming in this setting.

It is interesting to note that such a theory might serve as a foundations for computer science in the sense that logicians searched to find such a foundational theory for mathematics. Many people are happy with Set Theory in that role. But for computer science the formal theory should offer a programming notation as well since computing involves programming in a central way; and a formal notation is needed for that. If the programming notation were independent of the foundational theory, then we could not directly use the theoretical insights in programming and vice versa. Once we are committed to a formalized theory for computing, we need one that can express arguments about algorithms and systems actually built, that is, it should be a programming logic as well as a foundational theory.

These two requirements place a large burden on such a theory. Simply adding a programming notation to Set Theory for example results in a costly and cumbersome solution. Constructive type theories have arisen as a more elegant and lean approach. In this article we will work in one such theory, but the ideas are applicable in most others being developed.

This article begins with remarks about the variety of type theories. Then it defines a small subset of the Nuprl type theory, which is an instance in the class of Martin-Löf type theories. The treatment is intuitive and informal since there reader can consult other sources for a complete and rigorous account [14, 31], and the book of Nordstrom *et al* [43] covers similar ground. After this there is a presentation of the syntax and deductive machinery of the theory in itself, following exactly the method in [2]. This serves as a basis for the applications of metaprogramming. The main example is a derivation of a term matching algorithm critical in theorem proving and metamathematics. This example comes from [15] and was implemented by Howe in Nuprl. It is the example used in the Marktoberdorf course. The article ends with a discussion of *logic variables* which was not part of the Marktoberdorf course in 1990 but is germane to the topic and is now an active concern of type theory research.

2 Choosing a Type Theory

I do not intend to survey all of the candidate type theories, but there are at least three main species: Martin-Löf's predicative type theories and their extensions [35, 36, 14, 43, 30, 38, 40, 17], Girard's impredicative theories and their extensions [24, 25, 18, 34] [3], and Fefeman's theories of functions and classes [22, 29, 9]. In some ways these are converging in that they are treating the same class of concepts, and none is clearly right at the expense of the others. Indeed all are not right, but they represent the best we can do at present. They all offer the following advantages over taking the union of Set Theory and Programming Logic (say over ones favorite programming language).

1. Simple connections between programs and specifications

These type theories all use a functional programming notation and assign types essentially as for the typed lambda calculus. The connection to logic in the case of the ML and Girard theories is via the propositions-as-types principle. These two mechanisms combine elegantly, and the connection is robust, supporting many extensions to both type and programs. It offers also a richer type system than any in existing functional programming language, so in principle the result is a richer programming notation.

2. Rich specification languages

These type theories can express a great deal of mathematics. One reason for this is that they allow quantification over propositions. This is possible in Martin-Löf style theories because the notion *Type* presented above is refined into a hierarchy of types called *universes*. A universe is a type closed under all the type forming operations, so it looks like the collection of all types. But it is itself a type and is not contained in itself. The first of these is called $U1$. There is a larger universe $U2$ which contains all of the types from $U1$ plus $U1$ itself. Indeed there is a sequence of universes, U_i with U_i contained in U_{i+1} . (We will not be concerned with universes here and simplify the whole account by taking *Type* to be $U1$ and omitting mention of $U2$ and higher universes for the most part.)

These universes allow the user to define recursive types, and with recursive types one can define the natural numbers and lists and trees and other inductively defined data types. This is how we proceed here, but in Nuprl the integers are taken as primitive since we want a very efficient implementation, including a fast decision procedure for a subset of quantifier free arithmetic over plus, minus and times. In Nuprl the list forming operator is also primitive, but this is for historical reasons.

With recursive types one can even define infinitary versions of the logical connectives and obtain a version of infinitary logic. It is also possible to represent dynamic logic and other programming logics. In the case of the type theories with partial objects [48, 17] [3] this can be done directly taking programs as partial functions on state.

Of course Set Theory allows all of these concepts as well and can represent higher-order logic by taking propositions to be boolean valued functions. It is simple to define sets inductively and to provide a denotational semantics for dynamic logic and programming logics. However, the encoding of these ideas in Set Theory does not help us keep track of what can be computed and what can't be. So typically although we can for example define a model of a programming language, the partial functions used in it will not be computable, so we cannot actually execute the semantic definition. Also the definition of a mathematical concept in Set Theory is often not the one we want to compute with. Let us examine two examples of this phenomenon, first consider the definition of *multi-sets*, say as in [Manna & Waldinger/89book].

A multi-set is defined axiomatically as a set with the operations of adding an element, say $add(a, S)$. There is an empty multi-set, say $[]$. There is an axiom about equality,

saying that $add(x, add(y, S)) = add(y, add(x, S))$, and there is a principle of induction for proving things about such sets. It has the form that if we can prove P on $[]$ and if for every multi-set S and any element a we can prove that P holding on S implies that it holds on $add(a, S)$, then we know P on any multi-set. From this basis Manna & Waldinger show that we can define a choice function, *choose*, from any nonempty multi-set S to an element, $choose(S)$. They also show how to define the function $count(a, S)$ which denotes the number of times that the element a appears in S . Now a constructive account of multi-sets requires a more delicate approach. A complete account occurs in [4]. First these sets must be defined over some underlying type, say A . Then some concrete representation must be defined, say the sets of Lists over A . An equality relation is imposed on these representations, the same as above, and an induction combinator must be defined. It is like the induction principle stated above except that one must explicitly say that the method of proving P on $add(x, S)$ from P on S must respect the defined equality. It turns out that this restriction prevents us from defining the choice function, $choose(x, S)$, which is a good thing since it is not in general computable for multi-sets over any base A .

As a second example, consider the definition of the real numbers in Set Theory. There one can choose from among many equivalent definitions, say Dedekind cuts versus Cauchy sequences versus nested intervals and so forth. But one cannot compute with any of these definitions, and some of them are difficult to modify to make them computable. Dedekind cuts do not lead to a notion of computable reals in a natural way whereas the Cauchy definition does. Namely, following Bishop, [6], we say that a constructive real number is a sequence of rational numbers, x_1, x_2, \dots such that $|x_i - x_j| < 1/i + 1/j$. Two of them, x_1, x_2, \dots and y_1, y_2, \dots are equal iff for all i , $|x_i - y_i| < 2/i$. These are examples of two very basic mathematical concepts which arise frequently in computer science and for which the set theoretic definitions do not necessarily lead to good data structures for practical computation. So new computational definitions must be introduced and related to the other definitions. In most cases this process requires an elaborate definition of computability prior to the definitions. In type theory we can start with the computationally sensible definition right at the start because the right computing concepts are part of the basic theory.

3. A range of programming styles

functional programming The terms of constructive type theories include those that constitute a functional programming language, so they support directly the functional programming style. In fact in a system like Nuprl we expect to be able to achieve the performance of a language like Standard ML on the subset of programs common to both languages, which is essentially the functional subset of Standard ML. This is sufficient performance for a wide range of algorithms and systems, including by the way the implementation of Nuprl itself [41].

object oriented programming The richness of these type theories allows them to express directly concepts such as modules and algebraic data types, ADT's. So it

is possible to program in the style of the so-called object oriented languages to some extent. For instance in [4] there are many examples including lists, multi-sets, formulas and proofs all formulated as ADT's using the constructors *fun* and *prod*. However, no one has explored in detail the adequacy with which type theory can represent the various operations on algebraic structures that make this kind of programming so attractive. I would conjecture that many of these operations will have a pleasant formulation.

logic programming In a system like Nuprl it is possible to use the metalanguage to enable a logic programming style. Lipton [33] has actually built a Prolog engine in Nuprl in just this way. But we can see the main idea by looking at an example of a Prolog style computation done using just the basic types of a Martin-Löf theory plus the Nuprl-style tactics. This will be done in section 3 where we also discuss logic variables.

imperative programming There is good evidence to believe that the mechanisms being used in functional programming to attain the performance levels of imperative programming languages will be available in these type theories, either in the implementation or in some cases in the logic as well. For example the work of Griffin and Murthy [28, 41] shows how to type the control operator call/cc (the “go to's” of functional programming). Moreover this work reveals a deep connection between these evaluation strategies in functional programming and certain transformations important in proof theory.

4. Expressing basic ideas from computer science

The constructive type theories are able to explain and generalize some of the concepts basic to computer science. In the first place they provide a rigorous and general notion of data type which generalizes many of the earlier notions. For example, variant records become dependent products, function spaces can be seen as special cases of dependent function spaces, set types can be defined over any type, modules are instances of parameterized dependent products (as are abstract types) and recursive types can be generalized to include the constructive ordinals. In addition new relationships are revealed by the definitions as in the case of lazy data types being the co-inductive types [37].

The underlying deductive mechanism of a system such as Nuprl can be used to unify many important basic concepts. For instance there is one uniform notion of binding and scope which applies to the entire system. Also the notion of a transformation tactic gives as well the concept of a program transformation. The use of tactics to define the idea of a high-level proof applies as well to defining high-level programs.

One of the most important unifications provided by constructive type theories occurs at the interface between mathematics and computer science. Here we see that

many of the concepts from constructive mathematics are exactly what is needed in computing, e.g. the real numbers, and conversely many of the algorithms needed in constructive mathematics can be developed by techniques from computer science. An illustration of this later point occurs in Howe’s library of basic real analysis, [12], the proof of the Intermediate Value Theorem is based on a simple binary search algorithm whose structure is directly apparent from the proof. In contrast the same proof in Bishop&Bridges hides the algorithm. In another field, the constructive proof of Higman’s lemma provided by Murthy and Russell [42] gives an algorithm based on regular expressions which is much simpler to understand than the one which can be obtained from the classical proofs, [41].

5. Naturally automated

One of the most successful and interesting approaches to automating reasoning is the method of tactics pioneered by the LCF system [27]. This approach fits very well with type theory because it involves using a richly typed programming language as a metalanguage for proof construction. These type theories can use themselves as programming notations for automating reasoning in them. So for example it is possible to write and prove correct a theorem proving procedure for first-order matching, in the programming styles suggested above and then use this for facilitating formal reasoning in the type theory. An interesting example of this is the match theorem presented in this article.

3 A Simple Type Theory

Here is a simple type theory that illustrates the basic concepts needed in the rest of the paper. The basic syntactic concept is a *term*.

Definition 1 *The terms are divided into two groups, the canonical ones and the noncanonical ones. Here are all of the canonical terms: $prod(A; x.B)$, $fun(A; x.B)$, $union(A; B)$, $void$, $rec(x.b)$, $Type$, $pair(a; b)$, $\lambda(x.b)$, $inl(a)$, $inr(b)$, $any(b)$, $eq(t; e1; e2)$, $axiom$.*

Here are the noncanonical ones: $spread(p; u, v.t)$, $ap(f; a)$, $decide(d; u.t1; v.t2)$, $rec - ind(t; h, x.t)$.

The notions of *binding variable*, *bound variable*, *free variable*, *scope* and *closed term* are carried over to these terms from the lambda calculus in a natural way. For instance, for the lambda term $\lambda(x.b)$ we say that x before the dot is a binding occurrence of x whose scope is b . This occurrence of x along with the dot forms a binding operator which binds all free occurrences of x in b . A variable is free in an expression if it does not occur in the scope of a finding operator. These concepts are defined precisely for the reflected concept of terms, denoted *Term*, to come later.

There are *computation rules* telling how to *reduce* the noncanonical terms; the canonical ones reduce to themselves. In these rules the notation $exp[a/x]$ denotes the term obtained from exp by substituting a for each occurrence of x in exp .

Definition 2 *To reduce $spread(p; u, v.t)$, first reduce p . If the result is $pair(a; b)$, then continue by reducing $t[a/u, b/v]$.*

To reduce $ap(f; a)$, first reduce f ; if the result is $\lambda(x.b)$, then continue by reducing $b[a/x]$.

To reduce $decide(d; u.t1; v.t2)$, first reduce d ; if the result is $inl(a)$, then continue by reducing $t1[a/u]$; if the result is $inr(b)$, then continue by reducing $t2[b/v]$.

To reduce $rec - ind(t; h, x.b)$, reduce $b[t/x, \lambda(w.rec - ind(w; h, x.b))/h]$.

3.1 typing rules

These terms are the basis of a simple type theory. A type will be defined when we give it a canonical name and say which canonical terms are elements of it. One type is determined by the term *void*. It is a type with no members. We generally speak of the term which is the canonical name of a type as a type, so we say *void* is a type.

Another type is determined by the name *Type*. (This is the simple name we are using for the first universe, $U1$.) To define its canonical members we proceed inductively, and we introduce the auxiliary concept of a family of types over a type A . If B is a term and A is a type, and if for every element a of A the term $B[a/x]$ is a type, then B is a family of types indexed by A (using index variable x).

If A and C are types and if B is a type indexed by A (with index variable x) then $prod(A; x.B)$, $fun(A; x.B)$ and $union(A; C)$ are types. If x does not occur in B , then we omit the x . binding operator. If when T is a type, then F is, and if T occurs only positively in F then $rec(T.F)$ is a type. The positivity condition is given in detail in [14], but here we will use for F only terms built from $prod$ and $union$ which are all positive. (The essential idea is that T occurs only in the range of a function type, not in its domain.)

To specify what types these are, we must define their canonical elements. The canonical elements of the types $union(A; C)$ are those terms $inl(a)$ and $inr(c)$ where a is a term of type A and c is one of type C ; neither a nor c need be canonical, so we must also to define what it means generally for a term to be of some type T . We do this below.

The canonical elements of $prod(A, x.B)$ are the terms $pair(a; b)$ where a is of type A and b is of type $B[a/x]$. We know that these $B[a/x]$ are types since $prod(A; x.B)$ is.

The canonical element of $fun(A; x.B)$ are the terms $\lambda(x.b)$ such that for each a of type A , $b[a/x]$ is of type A .

The canonical elements of $rec(T.F)$ are those of the type $F[rec(T.F)/T]$.

A term with no free variable (i.e. a *closed term*) is of type A if it reduces to a canonical term of type A . Thus if $t[a/x, b/y]$ reduces to a canonical term type T , then $spread(pair(a; b); x, y.t)$ is of type T .

A term with free variables can be contingently of type A depending on assumption about the types of the free variables. To define this, let x_1, \dots, x_n be the free variables of t assume A_1 , is a type, and say that A_2 is a type, possibly depending on A_1 , if for each a_1 of type A_1 , $A_2[a/x]$ is a type. Likewise for A_n depending on A_1, \dots, A_{n-1} , and T depending on x_1, \dots, x_n . Now t is of type T if for each a_1 of type A_1 , A_2 of type $A_2[a_1/x]$, \dots , a_n of type $A_n[a_1/x_1, \dots, a_{n-1}/x_{n-1}]$, $t[a_1/x_1, \dots, a_n/x_n]$ is of type $T[a_1/x_1, \dots, a_n/x_n]$. For example, $inr(x)$ is of type $union(A; B)$ if x is assumed to be of type A .

3.2 semantics

Just as in the lectures we adopted a semantic approach to type theory, so at this point we do not give an enumeration of the rules which embody the ideas just described concerning reduction of terms (which become *computation rules*), the formation of types (which become *formation rules*) and the conditions for membership in types (which for the canonical terms become *introduction rules* and in the case of the noncanonical terms become *elimination rules*). A complete set of rules can be found in [14].

In the lectures at Marktoberdorf the semantic methods of Stuart Allen [1] were used as a rigorous basis for this semantic approach. But due to space limitations here we merely cite Allen's work and proceed informally since there is no need to use the rigorous approach for the results stated in this article.

3.3 encoding constructive logic

The *propositions-as-types principle* establishes a correspondence between types and propositions based on the idea that a proposition is true if and only if the corresponding type is inhabited. For decidable atomic propositions it is easy to begin the correspondence, namely a proposition P is taken to be an empty type if it is false and a type with at least one element if it is true; so *false* corresponds to the type *void*. One can use the proposition itself as a type. The correspondence is extended to compound propositions by the following definitions.

$$\begin{aligned}
 A \& B \text{ corresponds to } prod(A; B) \\
 A \vee B \text{ corresponds to } union(A; B) \\
 A \Rightarrow B \text{ corresponds to } fun(A; B) \\
 \forall x : A. B \text{ corresponds to } fun(A; x.B) \\
 \exists x : A. B \text{ corresponds to } prod(A; x.B)
 \end{aligned}$$

Among the atomic propositions are these types $eq(A; s; t)$ provided A is a type and s, t are of that type. These are types which are inhabited by *axiom* precisely when s and t are equal elements of type A ; otherwise the types are empty. The conditions under which equality holds are given next.

If $pair(a1; b1)$ and $pair(a2; b2)$ are in $prod(A; x.B)$, then they are equal if and only if $a1$ and $a2$ are equal in A , and $b1$ and $b2$ are equal in $B[a1/x]$. If $\lambda(x.b1)$ and $\lambda(y.b2)$ are in $fun(A; x.B)$, then they are equal if and only if $b1[a/x]$ equals $b2[a/x]$ in $B[a/x]$ for all a in A . If $inl(a1)$ and $inl(a2)$ are in $union(A; B)$, then they are equal if and only if $a1$ equals $a2$ in A . Similarly for the right injections $inr(b1)$ and $inr(b2)$.

The equality types can be used to express type membership as well. When we know that the type $eq(A; a; a)$ is inhabited, always by *axiom* and nothing more, then we know that a in fact belongs to the type A . We adopt the Nuprl convention that the judgement that A is a type is expressed as $eq(Type; A; A)$.

This correspondence between propositions and types is justified on semantic grounds and can be seen as a *realizability* semantics for constructive logic as is by now well-known.

3.4 abbreviations

The uniform prefix operator notation for terms is not the most convenient for readability, especially when writing the logical operators. In Nuprl there is an extensive definition facility that allows users to write new operator names and provide *display forms* for them. We will not explain that facility but instead provide several abbreviations for this article which can easily be written with the definition facility.

First, for the operators *fun* and *prod* we allow the more conventional mathematical symbols: $fun(A; B)$ is abbreviated $A \rightarrow B$ and the dependent form, $fun(A; x.B)$ is written $x : A \rightarrow B$. The product $prod(A; B)$ is abbreviated $A \times B$ and sometimes because of historical ties to Cambridge and Edinburgh ML this was written $A \sharp B$. The dependent form, $prod(A; x.B)$ is written $x : A \times B$ or $x : A \sharp B$. The disjoint union, $union(A; B)$, is abbreviated as $A + B$ and in some Nuprl texts as $A | B$. The equality type $eq(A; a; b)$ is written also as $a = b \in A$ or as $a = b \text{ in } A$ in Nuprl. In the special case of $eq(A; a; a)$ used to express membership, we also write $a \in A$. When we want to stress the logical meaning of types, we will use the notation from the correspondence listed above.

3.5 defining numbers and lists

natural numbers We can define the natural numbers using the recursive type constructor. First define $\mathbf{1}$ to be $fun(void; void)$ and let $*$ be its element $\lambda(x.x)$. Now let Nat be $rec(N.\mathbf{1} + N)$. Some typical elements are $inl(*)$, $inr(inl(*))$, $inr(inr(inl(*)))$, \dots . Of course the first of these represents zero. Define the successor function, *succ*, from Nat to Nat as $\lambda(n.inr(n))$. If we can define functions by primitive recursion

and prove propositions by Peano induction, then we can carry out elementary number theory in the normal way.

A *primitive recursive* function definition can be put in the form

$$\begin{aligned} f(0) &= b \\ f(n+1) &= \text{exp}(n, f(n)). \end{aligned}$$

Suppose that e is an expression in which u is a variable for the argument n and v is one for the inductive argument $f(n)$, then a linear form of this definition is given by the *induction combinator* $\text{ind}(n; b; u, v.e)$ whose evaluation rule is that $\text{ind}(0; b; u, v.e)$ evaluates to the value of b , and $\text{ind}(n+1; b; u, v.e)$ evaluates to the value of $e[n/u, \text{ind}(n; b; u, v.e)/v]$. For example the curried addition function is defined as

$$\begin{aligned} \text{add}(0) &= \lambda(y.y) \\ \text{add}(n+1) &= \lambda(y.\text{succ}(\text{add}(n)(y))). \end{aligned}$$

The linear form of this definition is just $\text{ind}(n; \lambda(y.y); u, v.\lambda(y.\text{succ}(v(y))))$. This operator is defined as $\text{recind}(n; h, x.\text{decide}(x; u.b; v.e[h(u)/v]))$.

lists Lists over a type A are defined as $\text{rec}(L.(1) + (A \times L))$ where $\text{inl}(\ast)$ now serves as nil , the empty list. We abbreviate these types as $A\text{list}$. The elements include nil , $\text{inr}(\text{pair}(a; \text{nil}))$, $\text{inr}(\text{pair}(a; \text{inr}(\text{pair}(b; \text{nil}))))$ where a and b are in A . The cons function which takes an element of A and a list and produces a new list with the element conjoined to the given list is $\lambda(x.\lambda(L.\text{pair}(x; L)))$. As in the case of natural numbers, we can develop the theory of lists if we have a primitive recursive function which defines new functions and builds proofs of propositions. To obtain this *list induction combinator* we build it from recind . The pattern of primitive recursion on lists which we capture is

$$\begin{aligned} f(\text{nil}) &= b \\ f(\text{cons}(a)(L)) &= \text{exp}(a, L, f(L)). \end{aligned}$$

For example the function which computes the length of a list is defined as

$$\begin{aligned} \text{len}(\text{nil}) &= 0 \\ \text{len}(\text{cons}(a)(L)) &= 1 + \text{len}(L). \end{aligned}$$

The linear form of this recursion combinator is $\text{listind}(l; b; h, t, v.e)$ which reduces according to the rules that $\text{listind}(\text{nil}; b; h, t, v.e)$ reduces to b , and $\text{listind}(\text{cons}(a)(L); b; h, t, v.e)$ reduces to $e[a/h, L/t, \text{listind}(L; b; h, t, v.e)/v]$. This can be defined from recind as $\text{recind}(L; f, x.\text{decide}(L; v.b; u.\text{spread}(u; h, t.e[f(t)/v])))$.

4 Basis for Metalevel Programming

In this section we discuss a particular way to provide metalevel programming facilities in the type theory we are using. The method is widely applicable as is shown in [2]. The basic idea is that we can define the term and proof structure of this type theory in itself if the right primitives are present. Then we can manipulate expressions which denote programs and proofs; this is the essence of metaprogramming. First let us remind ourselves of one of the most useful applications of metaprogramming (indeed the only one we examine in detail in this article), namely theorem proving tactics.

Formal proofs of interesting theorems tend to be large objects, so in practice some mechanism is used to shorten and abbreviate them. One of the most effective is the use of *tactics* (LCF [27], Nuprl [14], λ Prolog [39], HOL [26] and Oyster [11]). Proofs then are *tactic-trees*. These are proofs represented as trees in which certain steps are justified by programs (tactics) that compute subtrees. Tactics are metaprograms. We need to see how they relate to the ordinary kind.

In all heretofore existing systems, the tactics are written in a separate language, called the *metalanguage*, hence the origin of the acronym ML for the programming language ML. So the tactic trees use two languages, one the *object language*, the other a *metalanguage*. In LCF the object language is the polymorphic predicate calculus (PP λ), in Nuprl and Oyster it is constructive type theory and in HOL it is a simple type theory. The metalanguage in these systems is a programming language: ML in LCF, Nuprl and HOL, but Prolog in Oyster. We are now considering the situation in which the object language itself has a notion of computation adequate for expressing tactics. This raises the possibility of using a single language as both object language and metalanguage if we can manage to reflect proofs from the metalanguage into the object language. (This enterprise is especially interesting in the case of systems like Nuprl in which programs are extracted from proofs, for then the language of proofs is itself a metalanguage for proofs.)

For Nuprl we know that there is such a class of proofs. This was shown in [2] and is related to an extensive study of the problem in [32, 30, 32]. The work of Allen, Constable, Howe and Aitken which is summarized here uses the reflection of the proof concept in a formal theory to define the notion of proof in that theory. An interesting closure principle is revealed in these results. It is based on a fixed point construction over a class of proof-like trees which interleave objects from the object language and the metalanguage. In this context metaprograms are just ordinary programs of the object language that compute with the special internal data types of terms and proofs that represent terms of the object language.

The resulting concept of reflected proof is especially noteworthy because it is encapsulated in a single recursive type, in contrast to other approaches which result in an infinite tower of types or languages [32, 50] or would result in an infinite tower if extended to provide the same closure [51].

The notion of a reflected proof turns out to be more useful than the unreflected kind. In the LICS paper [2] there are seven applications of the concept mentioned. They

span the areas of theorem proving, type definition and functional program evaluation. First, it is mentioned how to shorten proofs and speed-up the generation and checking of proofs by proving that the tactics are correct. Second, a kind of polymorphism can be provided for theorems involving universe types. Third, it is mentioned that one can build evaluation mechanisms provably equivalent to the built-in evaluator yet more efficient on certain terms. Fourth, as a special case of the third application, it is claimed that expressions can be detected that can be destructively updated, thereby allowing a procedural evaluation for them. Fifth, it is said that reflection gives one approach to expressing computational complexity. Sixth, it is suggested that certain systems extensions can be explored in a mathematically controlled way by running code from incomplete proofs of metatheorems describing them. Finally, proof transformations are mentioned. One can see that these could be used to build new logics on top of the built-in logic of a theory.

4.1 representation

The context of this work is that we are standing outside of a formal language L (traditionally called the *object language*). In this article it is the type theory we introduced. The syntax of L is given by an inductively defined class of terms, as we have seen in the previous section defining the type theory. This is a type in the observer's language or *metalanguage*, ML . We are especially interested in those languages L with a computable evaluation relation on terms [25, 14, 27, 29, 45], again of the kind given by the reduction relation we defined above on the type theory terms.

We are concerned with languages L in which types can be defined. From outside of L we can see a type T of L as the class of terms which have that type, and we require that for any term t , t and its value t' have the same types, *i.e.*, if $t \in T$ then $t' \in T$; furthermore, we require that if $t \in T$ then t and t' are equal as members of T , *i.e.*, $t = t'$ in T . To be specific, the results of [2] will apply to the *type systems* defined in [1] which includes the one defined here.

A basic concept in defining reflection is the notion that a type of the language L represents a mathematical class S .

Definition 3 *Given a relation t reps s between terms t and members s of a class of objects S , we say that a type T of the language L represents S via reps if*

1. *for any term t and any $s, s' \in S$, if t reps s and t reps s' then s is s' .*
2. *for any $s \in S$ there is a term t such that t reps s .*
3. *for any terms t and t' , $t = t'$ in T if and only if there is an $s \in S$ such that t reps s and t' reps s .*

The first two conditions are general requirements for calling something a representation relation. The third is specific to the conventional use of a type as a system of notations

for objects of some class. If t is a member of T then $\downarrow(t)$ denotes the member of S that t represents. Note that a given object may have many representatives. In the representation schemes used below we pick out for each s a *standard* representation denoted $[s]$; the function taking s to $[s]$ will be computable.

In the case of the class of terms we have defined so far, there is a natural representation relation that can be designed into the language. Recall that the set of terms is the least set containing variables and some set of operator names, and such that if op is an operator name, t_1, \dots, t_n are terms, $n \geq 0$, and $\bar{v}_1, \dots, \bar{v}_n$ are lists of variables, then $op(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$ is a term. We say that t_i are the *subterms*; the \bar{v}_i are *binding variables* with *scope* t_i and they bind the variables occurring in t_i . The operators defined so far have at most three of these subterms, in the *decide* for example. Also we have used the convention that if there are no binding variables, then we do not write the dot, and if there are no subterms, as say in *Type*, then we do not write the parentheses either. We need now to include as well operator names among the terms in anticipation of the need for a term to represent each operator name. Suppose that we capitalize the operator for that purpose; so we add *VOID*, *FUN*, *PROD*, etc.

We also need now types *Var* and *Op* in L that represent variables and operator names, respectively. Indeed, we add them to the list of canonical terms of this type theory. Then a natural representation of a term $op(\bar{v}_1.t_1; \dots; \bar{v}_n.t_n)$ is a pair

$$\langle OP, (\langle \bar{V}_1, T_1 \rangle, \dots, \langle \bar{V}_n, T_n \rangle) \rangle$$

where OP represents op , \bar{V}_i represents \bar{v}_i and T_i represents t_i . Using this representation scheme we can show that the type *Term* defined as

$$rec(T. Op + Var + Op \times ((Var\ list \times T)\ list))$$

represents the set of terms of L .

The first disjunct of the disjoint union in the body of *Term*, *Op*, is used to represent the names of the operator names. So for example, *VOID* is now a new operator with zero arguments. We need a term to represent it. Instead of following the infinite regress of making up new operator names for this and then new names for the names, etc., we agree that these names of operator names can represent themselves. Thus the term representing the operator *VOID* is $inl(VOID)$, and the term representing the term *VOID* is $inr(inr(pair(inl(VOID); nil)))$. By contrast the term representing *void* is $inr(inr(pair(VOID); nil))$.

4.2 representing proofs

We are now going to define the representation of proofs. But since we have not included a metalevel definition of them and have not give specific primitive rules, we proceed differently than we did for terms. First we rely on the internal definition to inform us about the external one. Second, we describe proofs schematically, leaving out the particular rule names. We will not need those names at all in this article and they

only serve to clutter the presentation. But in fact there are examples of Nuprl proofs in the subsequent sections which serve to make the concept sufficiently concrete.

The proofs we deal with involve *sequents*, which are objects of the form

$$x_1:A_1, \dots, x_n:A_n \gg B$$

where each $x_i:A_i$ is a *hypothesis* consisting of a variable x_i and a term A_i , and where B is a term called the *conclusion* of the sequent. There is a natural representation of this class based on the representation of terms, namely,

$$\text{Sequent} \equiv (\text{Var} \times \text{Term}) \text{ list} \times \text{Term}.$$

Reflection of terms and evaluation is studied in more depth in [13].

The design of this class of proofs is intended to exploit their representability in two ways. Tactics are internally represented and a reflection rule is included.

First consider these design constraints. We take from the Nuprl design the general form of proof as a finite tree whose nodes are labeled with a sequent and some discrete justification “text” which is intended to indicate how the inference of the sequent from the sequents labeling the offspring is justified. The sequent at the root is called the *goal* of the tree. The leaves of a proof tree that are unjustified are the *premises* of the proof. Next, we require that the sequent and justification text completely (and effectively) determine the list of immediate *subgoals*, that is, the goals of the immediate subtrees. Finally, we require that proofs be effectively recognizable. However, we do not require that proofhood be effectively decidable; this is a necessary result of allowing arbitrary computable tactics, since one cannot then decide which (untyped) programs will produce proofs. To simplify this presentation, we shall develop the basic ideas without incorporating extraction of programs from proofs.

The rules that exploit the internal proof representation are the tactic rule and the reflection rule. An application of a tactic rule can be written

$$\begin{array}{l} s \quad \text{By tactic } t \\ s_1 \\ \vdots \\ s_n \end{array}$$

where s is the goal and the s_i are the subgoals. The justification text here contains a term t which represents a proof with goal s and premises s_1, \dots, s_n . An application of the reflection rule can be written

$$\begin{array}{l} s \quad \text{By refl } i, t \\ \gg \text{ provable } ([s], t, [i]) \\ s_1 \\ \vdots \\ s_n \end{array}$$

where t represents the list s_1, \dots, s_n , $[s]$ is the standard representative of sequent s , and $[i]$ is the standard representative of the number i . The number i is called the *level* of the reflection rule. $provable([s], t, [i])$ is a type representing the proposition that there is a proof with goal (root) s , premises s_1, \dots, s_n , and in which every use of reflection is of level less than i .

Another kind of justification is a primitive rule. We do not need to specify the form of justification text for applications of primitive inference rules. If g, s_1, \dots, s_n , $n \geq 0$, are sequents and r is the justification text for an instance of a primitive rule then define $instance(r, g, s_1, \dots, s_n)$ to be true if g is the conclusion and s_1, \dots, s_n are the premises of the instance of the rule named by r . Below we will refer to r simply as a primitive rule. More generally, we will not refer to “justification text” and instead only speak of justifications and rules.

Definition 4 *A justification is either a primitive rule, a tactic rule consisting of a single term, a reflection rule consisting of a nonnegative integer and a term, or the empty justification.*

The first problem considered in [2] was to devise a (non-circular) definition of a self-referential class of proofs and to demonstrate their validity. It is easier to deal with representation before defining proofs, so following the LICS presentation exactly we introduce *preproofs*, which are trees having the same components as proof trees but without restrictions on how the components fit together.

Definition 5 *A preproof is a tuple*

$$(s, r, \Delta_1, \dots, \Delta_n)$$

where s is a sequent, r is a justification, $n \geq 0$ and for each i , $1 \leq i \leq n$, Δ_i is a preproof.

The *root* and *leaves* of a preproof are defined in terms of the obvious interpretation of preproofs as trees. The *premise list* of a preproof is the list of the sequents, in left-to-right order, occurring at leaves that have the empty justification. We will refer to the premise list simply as the *premises* of the preproof. The root of a preproof is also called the *goal* of the preproof.

Above we defined types for the terms that represent terms and sequents. We can similarly define types representing justifications and preproofs (we assume that the justification text for primitive rules is representable—this is easy to arrange). Details are given in the final subsection in exactly the form worked out by Stuart Allen.

The definition of reflective proofs involves a syntactic fixed-point construction, so we need to introduce a parameter on which to take the fixed point. (For those interested only in being able to treat tactics internally, this detail is not necessary.) So for any term q we will define q -proofs; these will be just like proofs except for the reflection

rule. The purpose of the reflection rule is to allow us to make an inference step by proving that there is a proof justifying the step; the sequent which expresses the existence of such a proof must involve a type representing proofs. In a q -proof we use the term q in place of this as-yet unknown type of proofs. We will also need a particular fixed term T in the definition of q -proofs. We will not define T at this point; later we will specify the property required of T for validity of proofs, and make it plausible that T can be constructed (details are given at the end). The important point at this stage is that T is simply a particular term; exactly what term we pick is relevant *only* to the validity argument, since it enters in the definition of proof only as essentially a piece of text in a proof tree.

Definition 6 *Let q be any term. We inductively define the set $pf(q)$ of q -proofs as a subset of the set of preproofs. A q -proof is a preproof*

$$(s, r, \Delta_1, \dots, \Delta_n)$$

where s is a sequent, r is a justification and

1. $\Delta_1, \dots, \Delta_n$ are q -proofs,
2. if r is the empty justification then $n = 0$,
3. if r is a primitive rule then

$$\text{instance } (r, s, g_1, \dots, g_n)$$

where g_1, \dots, g_n are the goals of the preproofs $\Delta_1, \dots, \Delta_n$,

4. if r is a tactic rule consisting of a term t then t represents a preproof Δ such that Δ is a q -proof, the goal of Δ is s and the premises of Δ are the goals of $\Delta_1, \dots, \Delta_n$, and
5. if r is a reflection rule consisting of a number i and term t then $n \geq 1$, t represents the sequent list consisting of the goals of $\Delta_2, \dots, \Delta_n$, and the goal of Δ_1 is the sequent

$$\gg T(q)([s])(t)([i]).$$

The *immediate subproofs* of a q -proof

$$(s, r, \Delta_1, \dots, \Delta_n)$$

are $\Delta_1, \dots, \Delta_n$ and also, in the case that r is a tactic rule consisting of a term t , the preproof represented by t . A q -proof has *level* k if any number appearing in a reflection rule is less than k . More precisely,

$$\Delta = (s, r, \Delta_1, \dots, \Delta_n)$$

has level k if each immediate subproof of Δ has level k and in the case that r is a reflection rule r 's number is less than k .

We now need to argue that most of the above can be internalized in type theory. We have given a definition of a type $Term$ which represents the class of terms. Given a term t , we can easily determine the member $[t]$ of $Term$ which represents t . It is straightforward to write a function in type theory which reflects this. In particular, we can construct a term Rep of type $Term \rightarrow Term$ such that for every member t of $Term$, the term $Rep(t)$ represents $[u]$ where u is the term that t represents.

We also need a type that represents the class of q -proofs. This is straightforward to construct since the forms of inductive definition we have used are directly supported in this type theory. In the next subsection we define a term Pf such that if Q is a term representing a term q , then $Pf(Q)$ represents $pf(q)$.

We can now define the fixed-point exactly as it occurs in the LICS paper. Let d be the term

$$\lambda x. Pf(Ap(x, Rep(x)))$$

where Ap is a term which is a type theoretic function such that $Ap([a], [b])$ evaluates to $[a(b)]$. Finally, let p be the term $d([d])$.

Definition 7 *The set pf of proofs is $pf(p)$.*

Note that $Pf([p])$ represents the class of all proofs. If we reduce the term $p \equiv d(d)$ we get $Pf([p])$, so in Nuprl's type theory p is a type equal to $Pf([p])$. Hence p is also a type representing the class of all proofs.

Before we can prove the validity of proofs, we need to assume a certain property of the term T that was used in the definition of q -proofs. Suppose that S represents a sequent s , L represents a list l of sequents, and I represents a number i . We assume that the term $T(p)(S)(L)(I)$ is a type representing the class of all members Δ of pf such that Δ has level i , the goal of Δ is s , and the premise list of Δ is l .

T is actually quite straightforward to construct. We want $T(p)(S)(L)(I)$ to be a type consisting of all members of p with level I , goal S and premises L . This requires functions in type theory which compute the premises and goal of a proof, and a predicate for the level of a proof. Although it may appear that we need to know what proofs are in order to define T , functions in the type theory of Section 3 are untyped (as they are in full Nuprl) and can be defined to work on (representations of) preproofs. For the level predicate, the preproof represented by a tactic rule must be computed, and this can be done by a simple recursive algorithm. The level predicate will be partial on preproofs but total on proofs.

A proof is *valid* if the goal is true when the premises are. The proof of validity is mostly abstract with respect to the definition of truth for sequents. We assume, of course, that the primitive rules are valid. The only property of truth we will need is that if the sequent $\gg A$ is true, then A is a type that has a member.

Theorem 1 *Proofs are valid.*

Proof. We show by induction on n that proofs of level n are valid. For each n we argue by induction on the immediate-subproof relation. Suppose, then, that

$$\Delta = (s, r, \Delta_1, \dots, \Delta_n)$$

is a proof whose premises are true. Let $\overline{\Delta}$ denote the list $\Delta_1, \dots, \Delta_n$. By induction the goal of each Δ_i is true. We now consider the cases for r .

If r is the empty justification then s is a premise hence true.

If r is a primitive rule then s is true by our assumption of the validity of primitive rules and the fact that *instance* (r, s, g_1, \dots, g_n) where g_i is the goal of Δ_i .

If r is a tactic rule with term t then t represents a subproof Δ' of Δ , so by induction Δ' is valid. The premises of Δ' are the goals of $\overline{\Delta}$, so they are true, hence the goal of Δ' is true, and this is s .

If r is a reflection rule with integer i and term t , then the goal of Δ_1

$$\gg T(p, [s], t, [i])$$

is true, so the term in this sequent is a type that has a member, call it t . From our assumption about T it follows that t represents a proof Δ' of level less than i . By the induction hypothesis (of the induction on level) Δ' is valid. The premises of Δ' are the goals of $\Delta_2, \dots, \Delta_n$ (by the assumption about T) and so are true. The goal of Δ' must then be true, and this is s . \square

Tactic rules and reflection rules are eliminable from complete proofs. A *primitive* proof of s is a proof with goal s where every node is either a premise or is justified by a primitive rule. A *complete* proof is one with no premises. By an inductive proof of the same structure as the one just given for validity of proofs, we can show the following

Theorem 2 *If s has a complete proof then it has a complete primitive proof.*

In fact, given the extraction property to be discussed in the next subsection, there is a simple procedure for computing the primitive proof.

4.3 extraction

For this type theory, we are not satisfied with the validity of proofs. We want to be able to *extract* programs from proofs. For example, if we have proven a sequent of the form

$$\gg \forall x. \exists y. R(x, y)$$

then we expect to be able to derive from the proof a program which when given an x will produce a y such that $R(x, y)$ is true.

We need to refine our notion of truth of sequents somewhat. For the discussion here, say that t *satisfies* a sequent

$$x_1:A_1, \dots, x_n:A_n \gg B$$

if the free variables of t are among x_1, \dots, x_n and if for all members x_1, \dots, x_n of A_1, \dots, A_n respectively, t is a member of B .*

We require an automatic procedure for producing a satisfying term from a complete proof; we call this an *extraction* procedure. In analogy with our definition of validity for proofs, we say a partial term-valued function F of proofs and term lists is valid if for any term list b_1, \dots, b_m and any proof Δ with $n \leq m$ premises, if b_1, \dots, b_n satisfy the premises of Δ then $F(\Delta, b_1, \dots, b_m)$ satisfies the goal of Δ . We allow the list b_1, \dots, b_m to be longer than need be in order to make the definition of extraction given below easier.

As an implicit parameter for constructing our extraction procedure we assume a procedure $prextr(r, s, b_1, \dots, b_m)$ which is used to extract from proofs justified by a primitive rule.

We can now recursively define extraction as a partial function $extr$. To define

$$b = extr((s, r, \Delta_1, \dots, \Delta_n), b_1, \dots, b_m),$$

let $g(i)$ be one plus the sum of the lengths of the premise lists of $\Delta_1, \dots, \Delta_i$ and let h_i be $extr(\Delta_i, b_{g(i-1)}, \dots, b_m)$. If r is the empty justification then $b = b_1$. If r is a primitive rule then $b = prextr(r, s, h_1, \dots, h_n)$. If r is a reflection rule then

$$b = extr(\downarrow(fst(extr(h_1))), h_2, \dots, h_n)$$

where $fst(t)$ is the first component of the pair term that t evaluates to. Finally, if r is a tactic rule consisting of a term t then

$$b = extr(\downarrow(t), h_1, \dots, h_n).$$

4.4 details of the proof term

We will now present exactly Stuart Allen's construction of a proof term. In the following formalization of proofs, we use the notation $\overline{x_{i\dots j}}$ to denote a list containing the elements x_i, x_{i+1}, \dots, x_j . The *case* expression is used as destructor for the *Justification* type. It has the form *case j of c* where c is a sequence of clauses of the form $t_i x_i \rightarrow r_i$. If j evaluates to an n th injection of v , then the value of the expression is the value of r_n with x_n bound to v . Note that t_n is merely a mnemonic tag. The version of the *rec* type constructor used in the definition of *IsAPf*, $rec(z, x. t; a)$, is used to define an instance of a parameterized family of recursive types. t is a term that defines the type. In it z stands for the family of types being defined, and x for the parameter. a

*This is only an approximation of Nuprl's satisfaction relation. For more details, see [1].

is the actual parameter. As a simple example of the form's use, the proposition that a function f over the natural numbers eventually takes on the value 0 can be represented by the type $rec(P, x. f(x) = 0 \vee P(x + 1); 0)$.

The only function we will define on preproofs is *level*; *premises* and *root* are similar inductions over PPf (the type representing preproofs). The term T used in Section 4.2 is $\lambda qsl. Pobl(q; i; s; l)$. The *recind* form used below is a primitive form for constructing recursive functions over members of a recursive type.

$$Term \equiv rec(t. Op \mid Var \mid Op \times ((Var \text{ list} \times t) \text{ list}))$$

$$Sequent \equiv Term \text{ list} \times Term$$

$$Justification \equiv \{1\} \mid Prim \mid int \times Term \mid Term$$

$$PPf \equiv rec(s. Sequent \times s \text{ list} \times Justification)$$

$$\begin{aligned} level(p; l) &\equiv recind(p; \langle s, j, \overline{p_{1\dots n}} \rangle, h. \bigwedge_{i=1}^n h(p_i) \wedge \\ &\quad \text{case } j \text{ of} \\ &\quad \quad PREMISE } x \rightarrow true \\ &\quad \quad PRIM } x \rightarrow true \\ &\quad \quad REFL } \langle n, t \rangle \rightarrow n < l \\ &\quad \quad TACTIC } t \rightarrow \exists p: PPf. t \text{ Reps}PPf p \wedge h(p)) \end{aligned}$$

$$Pobl(Q; l; g; s) \equiv \exists p: Q.$$

$$level(p; l) \wedge$$

$$root(p) = g \in Sequent \wedge$$

$$premises(p) = s \in Sequent \text{ list}$$

$$roots(x) \equiv map \text{ root } x$$

$$\begin{aligned} IsAPf(p; Q) &\equiv rec(z, \langle s, j, \overline{p_{1\dots n}} \rangle. \bigwedge_{i=1}^n z(p_i) \wedge \\ &\quad \text{case } j \text{ of} \end{aligned}$$

$$PREMISE } x \rightarrow n = 0 \in int$$

$$PRIM } r \rightarrow instance(r; s; roots(\overline{p_{1\dots n}}))$$

$$REFL } \langle l, t \rangle \rightarrow$$

$$n > 0 \wedge$$

$$t \text{ Reps}SequentList roots(\overline{p_{2\dots n}}) \wedge$$

$$p_1 = \langle nil, T(Q)(Repsequent(s))(t)(repint(l)) \rangle$$

$$\in Sequent$$

$$TACTIC } t \rightarrow \exists x: PPf.$$

$$root(x) = s \in Sequent \wedge$$

$$premises(x) = roots(\overline{p_{1\dots n}}) \in Sequent \text{ list}$$

$$t \text{ Reps}PPf x \wedge z(x);$$

$$p)$$

$$Pf(Q) \equiv \{ p: PPf \mid IsAPf(p; Q) \}$$

The final definition of $Pf(())\mathbb{Q}$ is given using the *set type* of Nuprl. In the type theory of this article that type is not used, but it can be replaced by a dependent product with no loss of expressive power. So here we take $\{x x: A \mid B\}$ as $prod(A; x.B)$. This is in fact the way Martin-Löf treated sets in his type theories.

5 Metaprogramming

We now want to use the results of the previous section in support of metalevel programming. There is a sufficient basis for us to be able to use the type theory terms as a programming language operating on themselves and on proofs. Infact there is much more, there is a basis for proving that these programs are correct *and* using these correctness proofs to justify tactics as new rules of inference. For this later capability we need the reflection rule. However in the Marktoberdorf lectures and in this paper we will examine only the first of these capabilities, namely the use of the type theory terms as a metaprogramming notation. First we discuss briefly the matching problem and the value of formalized metareasoning.

A simple but important algorithm used to support automated reasoning is called *matching*: given two terms it produces a substitution, if one exists, that maps the first term to the second. In this section the matching algorithm is used to illustrate an approach to automating reasoning in which tactics for finding proofs are extracted from constructive proofs of metatheorems. Later in a subsection the algorithm is derived and verified following exactly an informal presentation of it in Section 5.2. The treatment of this example suggests how these theories can be soundly extended by the addition of constructive metatheorems about themselves to their libraries of results.

In this section we will describe the basic idea that automated theorem proving can be seen as the implementation of a constructive *metamathematics*. In metamathematics one proves theorems about doing mathematics. Some of these theorems have interesting computational content, such as a result claiming that a theory is decidable or a result saying that any proof of a theorem of one form can be converted to one of another form (say from prenex form into nonprenex form). We will show how some theorem proving algorithms can be understood as implementations of theorems about proofs and formulas.

One of the critical reasons for adopting this point of view is that much of the knowledge we discover about theorem proving is potentially useful to the automated system itself. In order to be used by machines, the knowledge must be formalized. Conversely, the more we can say about the algorithms we use, the more effectively we can use them. This suggests that we want to develop them in the context of a formal theory. Related reasons for doing this are discussed by Davis and Schwartz [20], Boyer and Moore [8] and Shankar [46].

One of the major advantages of formalizing knowledge about theorem proving is that we can use it to extend the stock of derived rules of inference. This in some cases

allows us to avoid running theorem-proving algorithms. To the extent that results from other parts of mathematics are useful in theorem proving, as in the use of graph algorithms in congruence closure, we want to be able to prove that they are correct and preserve the correctness of the logic when they are used in derived inference rules.

We begin with an informal derivation of the match algorithm and then present a formalization of its development that Douglas Howe carried out in the Nuprl proof development system. This formalization is based directly on the informal account.

5.1 syntax of terms for matching problem

The matching algorithm applies to a subset of the type of terms defined above. We do not consider terms with binding operators. A simple definition of the terms we consider would be to say that a variable is a term, and if t_1, \dots, t_n are terms and f is a function symbol, then $f(t_1, \dots, t_n)$ is a term. The important point for manipulating terms is not the particular way of displaying them but rather the component parts and a means of accessing them. So for a composite term such as $f(t_1, \dots, t_n)$, the components are f and the list (t_1, \dots, t_n) . The important feature of a variable is that it is the base case of the definition. So the official definition of a term specialized to this case is that it is a variable or it is a pair $\langle f, l \rangle$ where f is a function symbol and l is a list of terms. This is the mathematical definition, but we can agree to display a composite term as $f(l)$. If we let Var be the set of variables and Fun the set of function symbols, then the type $Term$ is defined as

$$rec(Term. Var + (Fun \# Term list)).$$

The induction principle for terms is given by the principle for this recursive type. We use it to prove the following theorem.

Theorem 1 *For all propositional functions P on terms, if $\forall x: Var. P(x)$, and if for all $Term$ lists l and function symbols f , whenever $P(t)$ holds for all terms t on l then $P(f(l))$, then $\forall t: Term. P(t)$.*

We will carry out constructions based on the structure of a term. To facilitate them we introduce a case discriminator written as

$$case\ t : y \rightarrow a; f.l \rightarrow b.$$

This means that if the term t is a variable then the value is a , where in the expression a , y stands for the variable; if the term is composite then the value is b where f and l are names in b of the function symbol and subterm list respectively. This case analysis is used together with induction on term structure to define concepts on terms. For example, here is a definition of the notion that a variable x occurs in a term t .

Definition 8 *A variable x occurs in a term t , abbreviated xet , if in the case that t is a variable y then $y = x$, and in the case that t is $f(l)$, then x occurs in some term u of l .*

The form of inductive arguments on term structure is clarified by seeing the computational meaning of induction. The induction rule defined above is not only a way to prove properties of terms, but it is also a method of computing based on the structure of a term, just as mathematical induction is a way of computing based on the structure of natural numbers. Recall that the computational form associated with induction on terms is denoted $rec_ind(t; h, x. b)$. To review computing by “term induction” on a term t , let us carry it through for the notion of x occurs in t . The expression b is a case discriminator on x , precisely,

$$rec_ind(t; occurs, z. case\ z: y \rightarrow x=y; \\ f, l \rightarrow \text{for some } u \text{ on } l. occurs(u))$$

Recall that the general rule for evaluating the recursion induction form $rec_ind(t; h, z. b)$ is that it reduces to

$$b[t/z, (\lambda w. rec_ind(w; h, z. b))/h].$$

So in the case of a term such as $f(x, v)$ the computation of the recursive definition x occurs in $f(x, v)$ is the following:

$$rec_ind(f(x, v); occurs, z. case\ z: y \rightarrow x=y \\ f, l \rightarrow \text{for some } u \text{ in } l. occurs(u))$$

reduces to (letting $occurs$ denote $\lambda w. rec_ind(w; \dots)$)

$$case\ f(x, v): y \rightarrow x=y \\ f, l \rightarrow \text{for some } u \text{ in } l. occurs(u)$$

which reduces to

$$\text{for some } u \text{ in } (x, v). occurs(u).$$

Taking u to be x results in

$$case\ x: y \rightarrow x=y \\ f, l \rightarrow \dots$$

In this case the term is a variable and the form reduces to

$$x = x$$

which is true. So x does occur in $f(x, v)$.

5.1.1 substitutions

For the matching algorithm we need to study the substitution of terms for variables in terms, as in substituting 2 for x in $3 * x = z$ to obtain $3 * 2 = z$.

Definition 9 *A substitution is defined as a list of pairs of a variable with a term, such as*

$$(\langle x, 2 \rangle, \langle y, z \rangle, \langle z, x + y \rangle).$$

So a substitution is an element of the type $\text{Var} \times \text{Term}$ list. Call this type Sub .

Given a substitution s and a variable x we write $s(x)$ to designate the term paired with x (the first if there is more than one). This notion can be defined precisely by a list induction form, namely

$$s(x) = \text{list_ind}(s; x; h, tl, v. \text{ if } h.1 = x \text{ then } h.2 \text{ else } v).$$

(In this example we let $h.1$ and $h.2$ select the first and second members of a pair respectively.)

The application of a substitution s to a term t is written as $s(t)$; it is defined by induction on the structure of t .

Definition 10 *If t is a variable y , then $s(t)$ is $s(y)$. If t is f, l , then $s(t)$ is obtained by applying s to each element of the list l . If we let $\text{map}(s, l)$ denote the function that applies s to each term on the list l , then $s(t)$ is*

$$\text{rec_ind}(t; h, z. \text{ case } z : y \rightarrow s(y) ; f, l \rightarrow f(\text{map}(s, l))).$$

Abbreviate $\text{map}(s, l)$ by $s(l)$.

The *domain* of a substitution are those variables that appear as the first element of a pair. We write $x \in \text{dom}(s)$ to indicate that x is in the domain of substitution s .

Definition 11 *We say that s_1 is a sub-substitution of s_2 , written $s_1 \subset s_2$, if for every variable x such that $x \in \text{dom}(s_1)$, $x \in \text{dom}(s_2)$ and $s_1(x) = s_2(x)$. We say that a substitution is minimal, $\text{min}(s)$, if no variable occurs twice on the left side of a pair.*

5.2 matching

We are now ready to discuss the problem of (first-order) matching. This is a basic matter in automated reasoning; for instance see [10, 23, 7, 52]. We say that term t_1 matches t_2 if there is a substitution s such that $s(t_1) = t_2$. For example, $f(x, g(y, z))$ matches $f(g(y, z), g(y, z))$ using the substitution $(\langle x, g(y, z) \rangle)$. But $f(x, y)$ does not match $g(x, y)$ because the outer operators are different. We will prove that for all terms t_1 and t_2 either t_1 matches t_2 , in which case we can find a minimal substitution

s such that $s(t_1) = s(t_2)$, or else no substitution will yield a match. More precisely, define $match?(t_1)$ if for every $t_2 \in Term$

$$\begin{aligned} & \exists s: Sub. s(t_1) = s(t_2) \ \& \ min(s) \ \& \ \forall x: Var. x \in dom(s) \Leftrightarrow x \in t_1 \\ & \vee \forall s: Sub. \neg(s(t_1) = s(t_2)). \end{aligned}$$

The main theorem can now be stated.

Theorem 2 (*match_thm*): $\forall t: Term. match?(t)$.

The idea of the proof is to check the outer structure of t_1 and t_2 . If t_1 is a variable, then the pair $\langle t_1, t_2 \rangle$ is the substitution. Otherwise t_1 is compound, say $\langle f_1, l_1 \rangle$. If t_2 is a variable then there is no match. So assume it has the form $\langle f_2, l_2 \rangle$. If $f_1 \neq f_2$, then there is no substitution. Otherwise the outcome depends on the result of trying to match l_1 and l_2 .

Suppose l_1 is *nil*. If l_2 is also *nil* then the empty substitution matches t_1 and t_2 ; otherwise there is no match. Suppose l_1 is $a_1.u_1$; if l_2 is *nil*, then there is no match. So suppose l_2 is $a_2.u_2$. First match a_1 and a_2 . If they do not match, then neither do t_1 and t_2 . Suppose they do, and let s be the substitution such that $s(a_1) = a_2$. Now try recursively to match u_1 and u_2 .

If these do not match, then neither do t_1 and t_2 . If s' matches them, then we check whether s and s' are compatible substitutions, *i.e.* whether they agree on common variables. If they do, then the final substitution is the union of s and s' . If they are incompatible, then we argue as follows that no match exists.

5.3 formal metamathematics

Most of the rest of this section is devoted to describing a formal proof that Douglas Howe carried out in Nuprl of the above theorem about matching. The formal proof took him about a day to construct using Nuprl. A complete listing of the proof can be obtained from Professor Howe at Cornell. The complete version is available with the distribution of the Nuprl system, and it can easily be read even by people not familiar with the details of type theory. The presentation here is based on the Marktoberdorf lecture notes which in turn came from [15].

Before we can proceed with the example, we need to give a brief description of those parts of Nuprl not covered in the type theory being used here. After the example, we discuss how programs such as matching can be incorporated in Nuprl's own inference mechanisms.

5.4 Nuprl

Most of the relevant type theory underlying Nuprl has been described earlier in this article. The structure of proofs was presented as an internal data type, but in order

to read output from the Nuprl *system*, some notation needs to be given and related to the internal proof type. First, we know that the inference rules of Nuprl deal with *sequents*, which are displayed as

$$H_1, H_2, \dots, H_n \gg P$$

where P is a formula and where each H_i is either a formula or a variable declaration of the form $x:T$ for x a variable and T a type. The H_i are referred to as *hypotheses*, and P is called the *conclusion* of the sequent. Sequents, in the context of a proof, are also called *goals*.

We have also seen in the definition of the proof type that a proof in Nuprl is a tree-structured object where each node has associated with it a sequent and a rule. We call rules in Nuprl *refinement* rules because we think of them as being applied backwards: given a goal, we *refine* it by using an inference rule whose conclusion matches the goal, obtaining *subgoals* which are the premises of the rule. The children of a node in a proof tree are the subgoals which result from the application of the refinement rule of the node. Following Bates [5] we call such logics *refinement logics*, and we consider them further in this article in our discussion of logic variables.

A key feature of Nuprl's proof structure as defined in the previous section is that a refinement rule need not be just a primitive inference rule, but can also be a *tactic* written in Nuprl itself or in the programming language ML [27]. As with a primitive refinement rules, the application of a tactic to a goal produces subgoals. Because of ML's and Nuprl's strong typing property, it is guaranteed that the subgoals imply the goal. When a tactic is used to refine a goal, the rule at the new node of the proof tree will show the text of the tactic. This gives a means for constructing higher level proofs that can serve as explanations of formal arguments.

Suppose we have proven a theorem

$$\forall i, j, m, n: \text{Int}. i \leq j \ \& \ m \leq n \Rightarrow m+i \leq n+j$$

and named it `mono`. An example of a refinement step using a tactic which applies this theorem is the following.[†]

$$\begin{array}{l} x:\text{Int}, 0 \leq x \gg x+2 \leq 2*x+2 \quad \text{BY (Lemma 'mono' ...)} \\ x:\text{Int}, 0 \leq x \gg x \leq 2*x \end{array}$$

The tactic `Lemma` computes the necessary terms and applies the rules necessary to instantiate the theorem. The three dots after the tactic indicate that a general purpose tactic called the *autotactic* was applied after `Lemma`. In this example, the autotactic proved the most trivial subgoals produced by `Lemma` (that x , $2*x$ and 2 are integers and that $2 \leq 2$). The result of the refinement is the single subgoal shown.

[†]We will use typewriter typeface when presenting objects constructed (or hypothetically constructed) with the Nuprl system.

5.5 a formal account of matching

Howe’s Nuprl library containing the formalization of matching can be divided into two parts. The first part, consisting of about 150 objects, is general. It contains definitions and simple theorems about the representation of logic within Nuprl. It also contains a minimal development of the theory of lists. The second part is particular to the formalization of matching; it contains about 40 definitions and theorems, pertaining to substitution and the syntax of terms.

The following description has three parts. First is a brief discussion of the tactics that were used in building the proofs in this library. This account is a summary of a much more extensive explanation given in Howe’s thesis [30]. (The entire tactic library has been documented and explained by Paul Jackson. This documentation is available with the Nuprl system.) Next is a presentation of the definitions and the simple theorems leading up to the main results of the library. Finally there is a detailed description of the proofs of the main theorem. In particular, we will examine one of the proofs of a lemma in its entirety in order to illustrate the character of reasoning about formal metatheory in Nuprl. All definitions and theorems not in the general portion of the library will be at least stated in this account.

5.5.1 preliminaries

Most of the work in proving the theorems in the library, in terms of number inference steps taken, was done by two general tactics. The first of these is the *autotactic*. It is usually able to completely prove subgoals involving typechecking (showing that a term is in a certain type or that a formula is well-formed—these properties are recursively undecidable even in the subset of Nuprl used in this article because of the generality of the type theory) and simple kinds of simple integer arithmetic and propositional reasoning. Associated with definitions in the library are theorems which give a type for the defined term. This style was adopted by Howe in his thesis in order to incorporate type checking into the autotactic; these types are used by the autotactic in proving the numerous typechecking subgoals that arise. The autotactic is usually effective in hiding the details of the type theory from the user; for example, there are no typechecking subgoals in the proofs of the main theorems described below. The autotactic is usually invoked via a Nuprl definition: when $(T \dots)$ appears in a proof, it means that the autotactic was applied after the tactic T .

The other general tactic is called **Simp** and was used by Howe to simplify terms. This tactic can be updated from the library (through a special kind of library object that can contain an ML form) with new simplification clauses in two basic ways. First, one can specify that a theorem be used for simplification. The theorem should be a universally quantified formula of the form

$$A_1 \Rightarrow A_2 \Rightarrow \dots \Rightarrow A_n \Rightarrow R(a, b)$$

where R is an equivalence relation (such as if-and-only-if or Nuprl’s built-in equality). The simplifier will then always try to rewrite terms matching a to an instance of

b. Second, one can directly specify a simplification that is computationally justified (where one term can be obtained from another by a sequence of forward or backward computation steps). For example, one can specify that the length $|h.l|$ of a list with head h always be simplified to $|l|+1$.

The “general” portion of the match library contains numerous updates to the simplifier, mostly for propositions and some for list theory. The simplifier, like the autotactic, is invoked by a definition: the notation $(T \dots +s)$ indicates that the simplifier and autotactic were both applied after T . One of the variants of this has `sc` in place of `+s`; which means that the simplifier was applied only to the conclusions of the subgoals produced by T .

Several other tactics use information from the library. `Induction` is used to perform induction on a variable appearing in a goal. If there is an induction principle in the library for the type of the variable, that is used, otherwise the type should be a primitive one (not a definition instance) for which there is an induction rule. This can be seen as a simple heuristic of the kind used by Boyer and Moore and in the Clam system. `Unroll` is similar to induction except that no induction hypothesis is generated. Finally, `Decide` takes a formula P and performs a case analysis on whether or not P is true. Decidability $(P \vee \neg P)$ is non-trivial in Nuprl, and `Decide` attempts to use theorems of the appropriate form to justify the case analysis.

There are quite a few tactics for purely logical reasoning; most of these are simple. For example, `ILeft` reduces proving a disjunction to proving the left disjunct, and `ITerm` reduces proving an existentially quantified term $\exists x: T. P(x)$ to proving $P(t)$ for some supplied term t . (“I” is for introduction). For analysing hypotheses there are various “elimination” tactics: `E` applies to most kinds of formula and performs one step of analysis; `SomeE` applies to “some” (or “exists”) formulas; and `EOn` (which requires a term argument) applies to universally quantified formulas. `HypCases` uses a universally quantified disjunction in the hypothesis list to perform a case analysis; it takes a list of terms as arguments to instantiate the quantified variables with. The most powerful tactic for logical reasoning is `Backchain` which uses (universally quantified) implications in the hypothesis list in a search for a complete proof. For example, if the conclusion of a sequent is Q and $P \Rightarrow Q$ is a hypothesis, then it reduces proving Q to proving P and then attempts to recursively prove Q ; if this fails it tries another hypothesis of a similar form.

There are several tactics for lemma application. One of these is `Lemma`, an example application of which was given above. `FLemma` uses a forward-reasoning analogue of what `Lemma` does. For example, if the theorem referred to is $P \Rightarrow Q$ and P is a (specified) hypothesis, then Q is added as a new hypothesis. Finally, `CaseLemma` uses a named lemma to determine a case analysis.

Finally, there are a few miscellaneous tactics that should be mentioned. The tactic `Thin` and its variants are used to remove unwanted hypotheses from a goal. `AndThin` takes a tactic that applies to hypotheses, applies it to a hypothesis and then discards the hypothesis. The so-called *tactical THEN* is used to combine tactics; if T_1 and T_2 are tactics then the tactic T_1 `THEN` T_2 first applies T_1 then applies T_2 to the resulting

subgoals. The last tactic we discuss is `Expand`. This tactic takes a list of names of definitions and expands all instances of those definitions in the goal.

5.5.2 definitions and simple theorems

Since the definitions given in Section 5.1 are given in type-theoretic terms, their formalizations are almost identical. In particular, the explanations given there still apply, so we will do little more here than to simply present the formal versions.

The simple theorems given below were easy to prove, and we will only present a proof of one of them. These theorems required on average about five steps each. What a “step” is is not well-defined since we can always collapse an entire proof into a single step by composing all the tactics used in the proof. We will rely on Howe’s description of some representative proofs to convey what a typical step is in this setting.

The definitions for syntax are straightforward. We first define the types of representatives of functions and variables in terms of Nuprl’s type of atoms (character strings).

```
Var == Atom
Fun == Atom
```

The type of term representatives is a recursive type of syntax trees:

```
Term == rec(T. Var | Fun # T list).
```

The equality relations for terms and lists of terms are frequently used, so definitions are made for them.

```
t1=t2 == t1=t2 in Term
l1=l2 == l1=l2 in Term list
```

Note that ambiguities in notations are acceptable in Nuprl (since definition instances do not have to be parsed). The “injections” or “term constructors” are defined by

```
x == inl(x)
f(l) == inr(<f,l>)
```

We prove an induction principle and a useful special case of it:

```
>> ∀P:Term->U1. (∀x:Var. P(x))
    => (∀l:Term list. ∀f:Fun. (∀t:l. P(t)) => P(f(l)))
    => ∀t:Term. P(t)
>> ∀P:Term->U1. ∀x:Var. P(x)
    => ∀l:Term list. ∀f:Fun. P(f(l))
    => ∀t:Term. P(t)
```

Here the proposition $\forall t:l. P(t)$ asserts that $P(t)$ is true for every member t of the list l . The first of these theorems can be read as “The property P of terms is true for all terms if (1) it is true for all variables and (2) if f is a member of Fun and l is a list of terms satisfying P then P is true of $f(l)$ ”. For our purposes, the type U1 can be thought of as the type of all propositions.

The form for case analysis of terms is the following.

```
case t: x→a; f,l→b == decide(t; x.a; ap. let f,l=ap in b)
```

This uses the type theory’s operators for analyzing a member of a disjoint union and for decomposing a pair. As in Section 5.1, we can now define when a variable *occurs* in a term.

```
x∈t == rec_ind(t; P,z. case z: y → y=x; f,l→ ∃u:l. P(u))
```

We also define a version of this predicate for lists

```
x∈l == ∃t:l. x∈t
```

so that a variable occurs in a list if it occurs in some member of the list. For this definition, as with other defined recursive functions and predicates, we add appropriate clauses to the simplifier. For example, we want any term of the form $x \in f(l)$ to simplify to $x \in l$.

The type of substitutions and the application of a substitution to a variable and a term are defined as before. We also define application to a list of terms.

```
Sub == Var#Term list
s(x) == list_ind(s; x; h,l1,v. if h.1=x then h.2 else v)
s(t) == rec_ind(t; h,z. case z: x→s(x); f,l→f(map(h,l)))
s(l) == map(λt.s(t), l)
```

As in Section 5.1, the notations $.1$ and $.2$ denote projections from a pair, and $\text{map}(h,l)$ applies the function h to every element of the list l .

We will need several properties of substitutions. In particular, we define when a variable is in the domain of a substitution, when one substitution is contained in another, when a substitution is minimal (that is, no two pairs in it have the same variable component) and when two substitutions are inconsistent (that is, disagree on some variable in both their domains).

```
x∈dom(s) == ∃p:s. p.1=x
s1⊂s2 == ∀x:Var. x∈dom(s1) => x∈dom(s2) & s1(x)=s2(x)
min(s) == list_ind(s; True; h,l,v. ¬(h.1∈dom(l)) & P)
ncst(s1,s2) == ∃x:Var. x∈dom(s1) & x∈dom(s2) & ¬(s1(x)=s2(x))
```

Finally, it is handy to have the following definition for the main theorem.

$$\begin{aligned} \text{match?}(t1) == & \forall t2:\text{Term}. \exists s:\text{Sub}. s(t1)=t2 \ \& \ \text{min}(s) \\ & \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \iff x \in t1 \\ & \vee \forall s:\text{Sub}. \neg(s(t1)=t2) \end{aligned}$$

Thus $\text{match?}(t1)$ asserts that for every $t2$ either there is a minimal matching substitution or there is no matching substitution. The minimality and the condition that a variable be in the domain of the substitution exactly if it occurs in t are both required for the inductive proof to go through. There is also a version of the above definition for lists.

$$\begin{aligned} \text{match?}(l1) == & \forall l2:\text{Term list}. \exists s:\text{Sub}. s(l1)=l2 \ \& \ \text{min}(s) \\ & \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \iff x \in l1 \\ & \vee \forall s:\text{Sub}. \neg(s(l1)=l2) \end{aligned}$$

This list version is defined simply for convenience, as will be made clear when we present the proof of the main theorem.

The following seven simple lemmas are incorporated into the simplifier and never have to be explicitly referenced. The fifth of these may look trivial; this is because Howe used the same display form for term equality as for equality in Var .

```
>>  $\forall x,y:\text{Var}. \forall s:\text{Sub}. \forall t:\text{Term}. x=y \implies (\langle x,t \rangle.s)(y) = t$ 
>>  $\forall x,y:\text{Var}. \forall s:\text{Sub}. \forall t:\text{Term}. \neg(x=y) \implies (\langle x,t \rangle.s)(y) = s(y)$ 
>>  $\forall x,y:\text{Var}. \forall s:\text{Sub}. \forall t:\text{Term}. x=y \implies x \in \text{dom}(\langle y,t \rangle.s) \iff \text{True}$ 
>>  $\forall x,y:\text{Var}. \forall s:\text{Sub}. \forall t:\text{Term}. \neg(x=y) \implies x \in \text{dom}(\langle y,t \rangle.s) \iff x \in \text{dom}(s)$ 
>>  $\forall x,y:\text{Var}. x=y \iff x=y$ 
>>  $\forall f1,f2:\text{Fun}. \forall l1,l2:\text{Term list}. f1(l1)=f2(l2) \iff f1=f2 \ \& \ l1=l2$ 
>>  $\forall x:\text{Var}. \forall f:\text{Fun}. \forall l:\text{Term list}. x=f(l) \iff \text{False}$ 
```

The following is the first theorem proved that has an interesting computational interpretation. Constructively it means that we can decide whether or not two terms are equal. From the proof of this theorem Nuprl can extract a decision procedure.

$$\text{>> } \forall t1,t2:\text{Term}. t1=t2 \vee \neg(t1=t2)$$

The next two theorems establish that the value of a substitution on a term (list) is characterized by its values on the variables occurring in the term (list).

$$\begin{aligned} \text{>> } & \forall s1,s2:\text{Sub}. \forall t:\text{Term}. \\ & \quad s1(t)=s2(t) \iff (\forall x:\text{Var}. x \in t \implies s1(x)=s2(x)) \\ \text{>> } & \forall l:\text{Term list}. \forall s1,s2:\text{Sub}. \\ & \quad s1(l)=s2(l) \iff (\forall x:\text{Var}. x \in l \implies s1(x)=s2(x)) \end{aligned}$$

```

* top
>>  $\forall x:\text{Var}. \forall s:\text{Sub}. \neg(x \in \text{dom}(s)) \vee \exists t:\text{Term}. x \in \text{dom}(s) \ \& \ s(x)=t$ 

BY (On 's' Induction ...+s)

1* 1. x: Var
   2. s: Sub
   3. p: Var#Term
   4.  $\neg(x \in \text{dom}(s)) \vee \exists t:\text{Term}. x \in \text{dom}(s) \ \& \ s(x)=t$ 
   >>  $\neg(p.1=x \vee x \in \text{dom}(s))$ 
       $\vee \exists t:\text{Term}. (p.1=x \vee x \in \text{dom}(s)) \ \& \ (p.s)(x)=t$ 

```

Figure 1: Proof by induction on s .

The computational content of the next simple theorem is a procedure which “looks up” a binding for a variable x in a substitution s , producing an indication of whether or not the variable is in the domain of s and in the former case giving the term bound to x .

```

>>  $\forall x:\text{Var}. \forall s:\text{Sub}. \neg(x \in \text{dom}(s)) \vee \exists t:\text{Term}. x \in \text{dom}(s) \ \& \ s(x)=t$ 

```

For future reference, the name of this theorem is `sub_lookup`. The proof is short and is given in its entirety as Figures 1 to 4. The figures show the proof steps as they would appear to a Nuprl user except that to save space we have removed from some steps some of the hypotheses that appear in earlier steps.

The first step in the proof is in Figure 1 and is by induction on the list s . This figure shows the goal sequent, then the rule applied, in this case a tactic, and then the subgoal sequent with its hypothesis list numbered and displayed vertically. The asterisk at the top left is Nuprl’s indication that the proof is complete, and `top` is the tree address of the proof node within the proof tree for the theorem. Note that in this step the simplifier and autotactic together automatically proved the base case and left the remaining subgoal in a simplified form. The step applied to this subgoal is shown in Figure 2. Here we decompose (“eliminate”) the pair p (and thin, or discard, its hypothesis) and then perform some reduction steps to simplify the subgoal. The next step is shown in Figure 3. In this goal we know that the property we are proving is inductively true of s , and we have to prove it for $\langle x1, t1 \rangle.s$. We want to do a case analysis on whether $x1=x$, so we use the tactic `Decide`. The negative case was proved automatically, and we are left with a simple subgoal in the other case. The last step (Figure 4) is trivial.

The structure of the program extracted from this proof is determined by the steps we have shown. The first step (using induction) introduces list recursion. In the next step, the induction hypothesis corresponds to a recursive call of the function we are defining. To lookup the value of x in $p.s$ we first decompose the pair p into its components $x1$

```

* top 1
3. p: Var#Term
4.  $\neg(x \in \text{dom}(s)) \vee \exists t:\text{Term}. x \in \text{dom}(s) \ \& \ s(x)=t$ 
>>  $\neg(p.1=x \vee x \in \text{dom}(s))$ 
     $\vee \exists t:\text{Term}. p.1=x \vee x \in \text{dom}(s) \ \& \ p.s(x)=t$ 

BY (AndThin E 3 THEN Reduce ...)

1* 4. x1: Var
    5. t1: Term
    >>  $\neg(x1=x \vee x \in \text{dom}(s))$ 
         $\vee \exists t:\text{Term}. (x1=x \vee x \in \text{dom}(s)) \ \& \ (\langle x1, t1 \rangle.s)(x)=t$ 

```

Figure 2: Let $x1$ and $t1$ be the components of p .

```

* top 1 1
3.  $\neg(x \in \text{dom}(s)) \vee \exists t:\text{Term}. x \in \text{dom}(s) \ \& \ s(x)=t$ 
4. x1: Var
5. t1: Term
>>  $\neg(x1=x \vee x \in \text{dom}(s))$ 
     $\vee \exists t:\text{Term}. (x1=x \vee x \in \text{dom}(s)) \ \& \ (\langle x1, t1 \rangle.s)(x)=t$ 

BY (Decide 'x1=x' ...+s)

1* 5. x1=x
    6.  $\neg(x \in \text{dom}(s)) \vee \exists t:\text{Term}. x \in \text{dom}(s) \ \& \ s(x)=t$ 
    >>  $\exists t:\text{Term}. t1=t$ 

```

Figure 3: Case analysis on whether or not $x1=x$.

```

* top 1 1 1
5. x1=x
6.  $\neg(x \in \text{dom}(s)) \vee \exists t:\text{Term}. x \in \text{dom}(s) \ \& \ s(x)=t$ 
>>  $\exists t:\text{Term}. t1=t$ 

BY (ITerm 't1' ...)

```

Figure 4: Take t to be $t1$.

and $t1$. The use of `Decide` in the next step corresponds to the introduction of an “if-then-else” expression whose condition is $x1=x$. In the case where $x1$ is not equal to x , a recursive call is made (that is, the induction hypothesis is used, automatically). In the other case the value $t1$ is returned.

The last theorems in Howe’s library before the three main theorems (which are discussed below) are added to the simplifier and forgotten.

```
>> ∀s1,s2:Sub. ∀x:Var. ∀t:Term.
      s1⊆s2 => (<x,t>.s1 ⊆ <x,t>.s2 <=> True)
>> ∀s1,s2:Sub. ∀x:Var. ∀t:Term.
      s1⊆s2 => ¬(x∈dom(s1)) => (s1 ⊆ <x,t>.s2 <=> True)
>> ∀s1,s2:Sub. ∀x:Var. ∀t:Term.
      s1⊆s2 => x∈dom(s2) => s2(x)=t => (<x,t>.s1 ⊆ s2 <=> True)
```

5.5.3 matching

The match procedure is mostly the product of the last three theorems in the library.

```
>> ∀s1,s2:Sub. min(s1) & min(s2)
      => ncst(s1,s2)
      ∨ ∃s:Sub. min(s) & s1⊆s & s2⊆s
      & ∀x:Var. x∈dom(s) => x∈dom(s1) ∨ x∈dom(s2)
>> ∀l:Term list. (∀t:l. match?(t)) => match?(l)
>> ∀t:Term. match?(t)
```

The names of these theorems in the library are `sub_union`, `lmatch_thm` and `match_thm` respectively. The proof of the first theorem has 23 steps, the proof of the second has 25, and the last has 13. Space limitations prevent us from presenting all of these proofs, so we will instead give the complete proof of the last theorem and just give some of the highlights of the other two. The level of inference in the three proofs is roughly the same.

The proof of `match_thm` is given in Figures 5 to 17. This proof is rather self explanatory, so not much additional explanation will be given. The first step (Figure 5) is by induction on t . The proof of the base case (where t is a variable) is easy and is contained in Figures 6 and 7. The first step in the proof of the induction step (Figure 8) is to apply the second of the three “main” theorems listed above, and then (Figure 9) we expand the definition of `match?`. In the next step (Figure 10) we “unroll” $t2$, doing a case analysis on whether $t2$ is a variable or an application. The variable case is easy (Figure 11). In the other case (Figure 12) we do a case analysis on whether the function parts of the two terms are the same.

The case where they are not the same is proved in one step (Figure 13). When they are the same we need to know if one argument list matches the other (Figure 14).

```

* top
>>  $\forall t:\text{Term}. \text{match?}(t)$ 

BY (On 't' Induction ...)

1* 1. x: Var
   >>  $\text{match?}(x)$ 

2* 1. l: Term list
   2. f: Fun
   3.  $\forall t:l. \text{match?}(t)$ 
   >>  $\text{match?}(f(l))$ 

```

Figure 5: Proof by induction on t.

```

* top 1
1. x: Var
>>  $\text{match?}(x)$ 

BY (Expand 'matchp' ...+s)

1* 1. x: Var
   2. t2: Term
   >>  $\exists s:\text{Sub}. s(x)=t2 \ \& \ \text{min}(s) \ \& \ \forall x1:\text{Var}. x1 \in \text{dom}(s) \ \Leftrightarrow \ x=x1$ 
       $\vee \ \forall s:\text{Sub}. \neg(s(x)=t2)$ 

```

Figure 6: Use the definition of match?.

```

* top 1 1
1. x: Var
2. t2: Term
>>  $\exists s:\text{Sub}. s(x)=t2 \ \& \ \text{min}(s) \ \& \ \forall x1:\text{Var}. x1 \in \text{dom}(s) \ \Leftrightarrow \ x=x1$ 
    $\vee \ \forall s:\text{Sub}. \neg(s(x)=t2)$ 

BY (ILeft THENM ITerm '[<x,t2>]' ...+s)

```

Figure 7: The left disjunct is true: take s to be [<x,t2>].

```

* top 2
1. l: Term list
2. f: Fun
3.  $\forall t:l. \text{match?}(t)$ 
>> match?(f(l))

BY (FLemma 'lmatch_thm' [3] ...) THEN Thin 3

1* 3. match?(l)
   >> match?(f(l))

```

Figure 8: By lmatch_thm and 3 we have match?(l).

```

* top 2 1
1. l: Term list
2. f: Fun
3. match?(l)
>> match?(f(l))

BY (Expand ''matchp'' ...+s)

1* 3. t2: Term
   4. match?(l)
   >>  $\exists s:\text{Sub}. f(s(l))=t2 \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \Leftrightarrow x \in l$ 
       $\vee \forall s:\text{Sub}. \neg(f(s(l))=t2)$ 

```

Figure 9: Use the definition of matchp.

```

* top 2 1 1
3. t2: Term
4. match?(1)
>>  $\exists s:\text{Sub}. f(s(1))=t2 \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \ \Leftrightarrow \ x \in l$ 
     $\vee \ \forall s:\text{Sub}. \neg(f(s(1))=t2)$ 

BY (On 't2' Unroll ...)

1* 4. x: Var
>>  $\exists s:\text{Sub}. f(s(1))=x \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \ \Leftrightarrow \ x \in l$ 
     $\vee \ \forall s:\text{Sub}. \neg(f(s(1))=x)$ 

2* 4. l2: Term list
5. f2: Fun
>>  $\exists s:\text{Sub}. f(s(1))=f2(l2) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \ \Leftrightarrow \ x \in l$ 
     $\vee \ \forall s:\text{Sub}. \neg(f(s(1))=f2(l2))$ 

```

Figure 10: Case analysis on the term kind of t2.

```

* top 2 1 1 1
4. x: Var
>>  $\exists s:\text{Sub}. f(s(1))=x \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \ \Leftrightarrow \ x \in l$ 
     $\vee \ \forall s:\text{Sub}. \neg(f(s(1))=x)$ 

BY (IRight ...+s)

```

Figure 11: In this case there are no matching substitutions.

```

* top 2 1 1 2
4. l2: Term list
5. f2: Fun
>>  $\exists s:\text{Sub}. f(s(1))=f2(l2) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \ \Leftrightarrow \ x \in l$ 
     $\vee \ \forall s:\text{Sub}. \neg(f(s(1))=f2(l2))$ 

BY (Decide 'f=f2' ...)

1* 6. f=f2
   >>  $\exists s:\text{Sub}. f(s(1))=f2(l2) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \ \Leftrightarrow \ x \in l$ 
       $\vee \ \forall s:\text{Sub}. \neg(f(s(1))=f2(l2))$ 

2* 6.  $\neg(f=f2)$ 
   >>  $\exists s:\text{Sub}. f(s(1))=f2(l2) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \ \Leftrightarrow \ x \in l$ 
       $\vee \ \forall s:\text{Sub}. \neg(f(s(1))=f2(l2))$ 

```

Figure 12: Either $f=f2$ or not.

```

* top 2 1 1 2 2
3. match?(l)
4. l2: Term list
5. f2: Fun
6.  $\neg(f=f2)$ 
>>  $\exists s:\text{Sub}. f(s(1))=f2(l2) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var}. x \in \text{dom}(s) \ \Leftrightarrow \ x \in l$ 
     $\vee \ \forall s:\text{Sub}. \neg(f(s(1))=f2(l2))$ 

BY (IRight ...+s)

```

Figure 13: Clearly no match in this case.

```

* top 2 1 1 2 1
3. match?(1)
6. f=f2
>>  $\exists s:\text{Sub. } f(s(1))=f2(12) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var. } x \in \text{dom}(s) \ \Leftrightarrow \ x \in 1$ 
     $\vee \ \forall s:\text{Sub. } \neg(f(s(1))=f2(12))$ 

BY (Expand ‘‘lmatchp’’ THEN HypCases ['12'] 3 ...)

1* 6.  $\exists s:\text{Sub. } s(1)=12 \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var. } x \in \text{dom}(s) \ \Leftrightarrow \ x \in 1$ 
    >>  $\exists s:\text{Sub. } f(s(1))=f2(12) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var. } x \in \text{dom}(s) \ \Leftrightarrow \ x \in 1$ 
         $\vee \ \forall s:\text{Sub. } \neg(f(s(1))=f2(12))$ 

2* 6.  $\forall s:\text{Sub. } \neg(s(1)=12)$ 
    >>  $\exists s:\text{Sub. } f(s(1))=f2(12) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var. } x \in \text{dom}(s) \ \Leftrightarrow \ x \in 1$ 
         $\vee \ \forall s:\text{Sub. } \neg(f(s(1))=f2(12))$ 

```

Figure 14: By 3 either 1 matches 12 or not.

```

* top 2 1 1 2 1 1
5. f=f2
6.  $\exists s:\text{Sub. } s(1)=12 \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var. } x \in \text{dom}(s) \ \Leftrightarrow \ x \in 1$ 
>>  $\exists s:\text{Sub. } f(s(1))=f2(12) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var. } x \in \text{dom}(s) \ \Leftrightarrow \ x \in 1$ 
     $\vee \ \forall s:\text{Sub. } \neg(f(s(1))=f2(12))$ 

BY ((OnLast (SomeE ‘‘s’’) THEN ILeft THENM ITerm ‘s’ ...) ...sc)

```

Figure 15: Let s match 1 with 12. Then s matches f(1) and f2(12).

```

* top 2 1 1 2 1 2
5. f=f2
6.  $\forall s:\text{Sub. } \neg(s(1)=12)$ 
>>  $\exists s:\text{Sub. } f(s(1))=f2(12) \ \& \ \text{min}(s) \ \& \ \forall x:\text{Var. } x \in \text{dom}(s) \ \Leftrightarrow \ x \in 1$ 
     $\vee \ \forall s:\text{Sub. } \neg(f(s(1))=f2(12))$ 

BY (IRight ...+s)

1* 5. s: Sub
    6. s(1)=12
    7.  $\forall s1:\text{Sub. } \neg(s1(1)=12)$ 
    >> False

```

Figure 16: 1 does not match 12, so there is no matching substitution.

```

* top 2 1 1 2 1 2 1
5. s: Sub
6. s(1)=12
7.  $\forall s1:Sub. \neg(s1(1)=12)$ 
8. f=f2
>> False

BY (EOn 's' 7 ...) THEN (Contradiction ...)

```

Figure 17: A match for $f(1)$ and $f2(12)$ would match 1 and 12.

When it does, we get the required matching substitution (Figure 15). Otherwise, we obtain a contradiction (Figures 16 and 17).

A discussion of the other two main theorems of the library can be found in the paper with Howe [15].

5.6 applying formal metamathematics

How can these results be applied to Nuprl itself? One method is to define the type of terms to match exactly the terms of Nuprl. Then the theorems such as the match theorem apply to Nuprl itself. In this form they are *enlightening* but not *useful*. To make them useful it must be possible to apply them in proving theorems. One way to do this is outlined in Howe's thesis [30]. Another method of proceeding is to formalize the metatheory of Nuprl, making the types of terms and proof two of the basic types. This is the approach we have taken here. It is this requirement that necessitated the material in the preceding section.

6 Logic Programming in Type Theory

The use of constructive type theory in defining basic computing concepts is new, and the community is learning how to improve the exact definitions of the basic concepts. A similar process went on in set theory for decades before the elegant and polished accounts that we now read were developed. Some work along these lines is still being done.

In the case of type theory we can see steady development and improvement. For example the treatment of inductively defined types has improved over the past five years. The first accounts, [16, 37] led to an implementation in Nuprl. Experience with these types revealed that the full notion of parameterized mutually inductive types was awkward. Subsequent work by Coquand, Paulin, and Dybjer has led to a more elegant and polished set of rules for the general case [19] [44, 21], and extensive experience with the simple unparameterized case has shown just how vital the concept

is in defining basic concepts [30]. Likewise new type constructors have been studied such as intersection types [Backhouse] [1] and partial types [48, 17, 47]. In this section we look briefly at one other possible improvement to these type theories, one that is germane to metalevel programming, namely the use of logic variables. Although these are sometimes considered to be *metavariables*, we outline an approach that treats them in the object logic.

6.1 refinement logics

When proving a theorem we often work backwards from the goal to axioms or assumptions. This is sometimes called *top-down development*. For instance we argue that $((A \Rightarrow B) \& (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$ by saying we need a proof of $(A \Rightarrow C)$ from a proof of $(A \Rightarrow B) \& (B \Rightarrow C)$. To prove $A \Rightarrow C$, we need a proof of C from A and the other assumptions. We can prove C if we can prove B because we are assuming $B \Rightarrow C$. But we can prove B if we can prove A because we assume $A \Rightarrow B$. But we assume A , so there is a proof.

The *tableau* proof style [49] is top-down, and Nuprl's logic is top-down. This kind of logic was also called a *refinement logic* by Bates [5]. The convenient way to present the rules of a refinement logic is to state the goal first, then the rule name and any information needed to determine the subgoals, and finally the subgoals. So for example the rule for proving $A \& B$ from assumption H is written in Nuprl as

$$\begin{aligned} H >> A \& B \text{ By and intro} \\ & 1. H >> A \\ & 2. H >> B. \end{aligned}$$

A rule which requires information beyond the rule name to generate the subgoals is

$$\begin{aligned} H, \forall x: A.B >> G \text{ by all elimination on } a \\ H >> a \in A \\ H, \forall x: A.B, B[a/x] >> G \end{aligned}$$

We need the name of the element of A to generate the subgoals.

In Nuprl the existential introduction rule also requires extra information. It is

$$\begin{aligned} H >> \exists x: A.B \text{ By intro } a \\ H >> a \in A \\ H >> B[a/x]. \end{aligned}$$

This rule is inconvenient because it interrupts the top-down development, forcing the user to choose a early in the proof.

One of the powerful features of logic programming languages, which are also top-down, is that they employ the concept of a *logic variable* to delay existential choices. If Nuprl had logic variables, for now let's write them as capitals, X, Y, Z, \dots , then the existential introduction rule could be written as

$$\begin{aligned}
H &>> \exists x : A.B \text{ By intro } X \\
&1. H >> X \in A \\
&2. H >> B[X/x].
\end{aligned}$$

The key assumption underlying the use of logic variables is that they are instantiated simultaneously at all occurrences. So if in the subproof of $B[X/x]$ we learn that X must be a particular value a , and we set X to a , then the first subgoal becomes $H >> a \in A$.

We will discuss below one way to treat logic variables in refinement logics, and we apply it in a logic programming setting. But first we consider the Prolog computation mechanism as it can be explained in a refinement logic like Nuprl without logic variable.

6.2 Prolog proof procedure

A “program” in Prolog is a set of inductive definitions of relations. For instance here is a Prolog form of the primitive recursive function definition

$$\begin{aligned}
f(0, y) &= y \\
f(s(n), y) &= h(n, f(n, y)).
\end{aligned}$$

It is a pair of relations

$$\begin{aligned}
&\forall y. R(0, y, y) \\
&\forall u, z, y. (R(z, y, v) \Rightarrow R(s(z), y, h(z, v))).
\end{aligned}$$

In general a program can consist of several relations or program *clauses*, say C_1, \dots, C_n . Each of them defines a relation in the form $\forall \bar{x}. (A_1 \& \dots \& A_n \Rightarrow B)$ where the A_i, B are atomic formulas, n may be 0. We say B is the “head” of its clause.

A goal clause has the form $\forall \bar{x}. (A \Rightarrow \exists \bar{y}. B)$. A constructive proof of this will find functions which compute a witness for this existential quantifier. The form of the function is simply an expression for the vector \bar{y} of values in terms of the primitives. The expression is presented as a substitution for \bar{y} . The proof procedure can find many expressions, so it is finding more than one function.

The Prolog proof procedure can be expressed without logic variables. It is just a specific search which uses unification and backchaining. The unification can be done “under the existential quantifiers.” Let Prog be the primitive recursive program above. Consider this Prolog goal

$$Prog >> \exists u. R(s(s(0)), b, u)$$

The first step is to unify $R(s(s(0)), b, u)$ against the head’s of the program clauses. There are only two choices, one succeeds.

$$\begin{aligned}
R(s(s(0)), b, u) &\text{ unifies with} \\
R(s(z), y, h(z, v)).
\end{aligned}$$

The resulting substitution is $z := s(0), y := b, u := h(s(0), v)$. We need to know whether there is a value for v . This generates a subgoal of the form

$$\exists v_2. R(s(0), b, v_2)$$

by backchaining on the second program clause. We can express the proof obligation by sequencing in this subgoal. The proof starts as

$$Prog \gg \exists v. R(s(s(0)), b, v) \text{ By seq } \exists v_2. R(s(0), b, v_2)$$

$$1. Prog, \exists v_2. R(s(0), b, v_2) \gg \exists u. R(s(s(0)), b, u)$$

$$2. Prog \gg \exists v_2. R(s(0), b, v_2).$$

Now the interesting point is that the entire first subgoal can be proved simply by elimination on $\exists v_2$ and backchaining. The first step results in the subgoal

$$Prog, v_2 : N, R(s(0), b, v_2) \gg \exists u. R(s(s(0)), b, u).$$

Then we substitute properly in the second program clause (the rule is \forall -elimination) to obtain

$$Prog, v_2 : N, R(s(0), b, v_2), R(s(0), b, v_2) \Rightarrow R(s(s(0)), b, h(s(0), v_2)) \\ \gg \exists u. R(s(s(0)), b, u).$$

Now we know that if we take u to be $h(s(0), v_2)$ the proof is finished by simple backchaining.

Notice that the definition of u as $h(s(0), v_2)$ is not complete until we know v_2 , but finding it has been relegated to the second subgoal, and substituting it into $h(s(0), v_2)$ is part of the definition of the sequent rule. So the entire problem has h reduced to proving the second subgoal, and it *is of the exact same form* as the original goal. This means we can apply the same method. We call this method the *Prolog search procedure*.

In Nuprl this method can be coded as a *proof tactic*, so we can see Prolog evaluation as the result of trying a fixed tactic on goals of the form $Prog \gg \exists v. B$. The tactic is guaranteed to generate only subgoals of this form; the other subgoals are automatically proved.

6.3 proof expressions in refinement logics

Before we can discuss the role of logic variable in refinement logics we need to consider how the justification for the goal of a sequent can be written as an algebraic expression. First consider the simple case of a step taken by *and introduction*.

$$H \gg A \& B \text{ By intro}$$

$$1. H \gg A$$

$$2. H \gg B.$$

Let us write after each proved sequent an expression that codes why it is true, that is how it was proved; call these *proof expressions*. If we are looking at a complete proof of $H \gg A \& B$, then $H \gg A$ is proved and so is $H \gg B$. Suppose the proof expression for the first is a and for the second b . Then we can propagate up the proof tree an expression for $H \gg A \& B$; we synthesize it out of expression for the subproofs. In this case we can take $pair(a; b)$ as the expression required; and we show these expressions as in this example

$$\begin{array}{l}
H \gg A \& B : pair(a; b) \\
1. H \gg A : a \\
2. H \gg B : b.
\end{array}$$

We read the synthesis of proof expressions *bottom-up* and the subgoals top-down. Here are the rules for the Intuitionistic Predicate Calculus written with proof expressions indicated.

$$H, x : A \gg A : hyp(x)$$

$$\begin{array}{l}
H \gg A \& B : pair(a; b) \\
H \gg A : a \\
H \gg B : b
\end{array}$$

$$\begin{array}{l}
H, s : A \& B \gg G : spread(z; x, y, g) \\
H, x : A, y : B \gg G = g
\end{array}$$

$$\begin{array}{l}
H \gg A \vee B : inl(a) \\
H \gg A : a
\end{array}$$

$$\begin{array}{l}
H, z : A \vee B \gg G : decide(z; x.g_1; y.g_2) \\
H, x : A \gg G : g_1
\end{array}$$

$$\begin{array}{l}
\text{or } H \gg A \vee B : inr(b) \\
H \gg B : b
\end{array}$$

$$H, y : B \gg G : g_2$$

$$\begin{array}{l}
H \gg A \Rightarrow B : \lambda(x.b) \\
H, x : A \gg B : B
\end{array}$$

$$\begin{array}{l}
H, f : A \Rightarrow B \gg G : ap(f; a; y.g) \\
H, f : A \Rightarrow B \gg A : a \\
H, y.B \gg G : g
\end{array}$$

$$H, x : false \gg G : any(x)$$

$$\begin{array}{l}
H \gg G : seq(a; x.g) \\
H \gg A : a \\
H, x : A \gg g
\end{array}$$

$$\begin{array}{l}
H \gg \exists x : A.B : pair(a; p) \\
H \gg A : a \\
H \gg B[a/x] : p
\end{array}$$

$$\begin{array}{l}
H, z : \exists x : A.B \gg G : spread(z; x, y.g) \\
H, xA, y : P \gg G : g
\end{array}$$

$$\begin{array}{l}
H \gg \forall x : A.B : \lambda(x.b) \\
H, x : A \gg B
\end{array}$$

$$\begin{array}{l}
H, f : \forall x : A.B \gg G : ap(f; a; y.g) \\
H, f.\forall x : A.B \gg A : a \\
H, y.B \gg G : g
\end{array}$$

6.4 logic variables

One approach to introducing logic variables into a refinement logic is suggested by looking at the existential introduction rule. If we want to introduce a variable as a witness for the quantifier, then it needs to be typed and “declared.” There does not seem to be a natural way to do it in the pure logic, but in the presence of proof expression we can try this idea. We allow variables to range over these proof expressions. So a possible existential introduction rule is

$$\begin{array}{l} H \gg \exists x : A.B \text{ by intro } X \\ \quad 1. H \gg A : X \\ \quad 2. H \gg P[X/x] \end{array}$$

The advantage of this approach is that X is available in the entire proof tree, just as we want, without having to add it to the hypothesis lists. Also we need just two rules for using these variables. First we need a definition rule saying that X satisfies these conditions, and it is of type A and satisfies P . We also need a rule to refine it or specify its value further.

In order for this notion to be flexible enough, we need to be able to indicate the dependence of X on variables declared in H . Then we could see X as a name for the function from H into A . So the complete account of logic variables must treat them as *second-order* variables. We write $X[z_1, \dots, z_b]$ where z_i are declared in H . Now $X[\bar{z}]$ can be used in those contexts that provide the values needed for \bar{z} .

Acknowledgements I would like to thank my colleagues Stuart Allen, Douglas Howe and Bill Aitken of Cornell who were coauthors on the two papers from which much of this lecture material came. I especially want to thank Doug Howe whose implementation of the matching algorithm was central to one of the sections. Indeed I have used snapshots of the proof that he prepared to explain that experience, and in this article I have borrowed from the organization of that material for our joint paper cited often in this text.

I would also like to thank Randy Pollack for many hours of stimulating discussions of type theory and especially for his help with formulating the concept of a logic variable. We may produce joint results on this topic in the near future showing how the concept defined here can be used to describe higher-order unification in a refinement logic.

References

- [1] Stuart F. Allen. *A non-type-theoretic semantics for type-theoretic language*. PhD thesis, Cornell University, 1987.
- [2] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William Aitken. The Semantics of Reflected Proof. In *Proc. of Fifth Symp. on Logic in Comp. Sci.*, pages 95–197. IEEE, June 1990.
- [3] P. Audebaud. Partial Objects in the Calculus of Constructions. In *Proc. of Sixth Symp. on Logic in Comp. Sci.*, pages 86–95, IEEE, Vrije University, Amsterdam, The Netherlands, 1991.
- [4] D. Basin and Robert L. Constable. Metalogical frameworks. In *Proc. of the Second Annual Workshop on Logical Frameworks*, Edinburgh, UK, June 1991.
- [5] J. L. Bates. *A Logic for Correct Program Development*. PhD thesis, Cornell University, 1979.
- [6] E. Bishop and D. Bridges. *Constructive Analysis*. Springer-Verlag, New York, 1985.
- [7] W. W. Bledsoe and D. W. Loveland. *Automated Theorem Proving: After 25 Years*. American Math Soc., 1984.
- [8] R. S. Boyer and J. S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In *The Correctness Problem in Computer Science*, pages 103–84. Academic Press, New York, 1981.
- [9] W. Buchholtz et al. Iterated Inductive Definitions and Subsystems of Analysis. *Recent Proof-Theoretical Studies*, Lecture Notes in Mathematics, Vol. 897, pages 16–77, 1981.
- [10] A. Bundy. *The Computer Modeling of Mathematical Reasoning*. Academic Press, New York, 1983.
- [11] A. Bundy. A Broader Interpretation of Logic in Logic Programming. In *Proc. 5th Symp. on Logic Programming*, page ??, 1988.
- [12] J. Chirimar and Douglas J. Howe. Implementing constructive real analysis: a preliminary report. In *Symposium on Constructivity in Computer Science*. Springer-Verlag, 1991. To appear.
- [13] Robert L. Constable, Stuart F. Allen, and Douglas J. Howe. Reflecting the open-ended computation system of constructive type theory. In H. Schwichtenberg, editor, *Logic, Algebra and Computation*, NATO ASI Series, Vol. F79, pages 265–280. Springer-Verlag, 1990.

- [14] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
- [15] Robert L. Constable and Douglas J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R.B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book*, pages 45–76. Elsevier Science Publishers (North-Holland), 1990.
- [16] Robert L. Constable and N.P. Mendler. Recursive Definitions in Type Theory. In *Proc. of Logics of Prog. Conf.*, pages 61–78, January 1985. (Cornell TR 85-659).
- [17] Robert L. Constable and Scott F. Smith. Computational foundations of basic recursive function theory. In *Third Symp. on Logic in Comp. Sci.*, pages 360–371, IEEE, Edinburgh, UK, 1988. (Cornell TR 88-904).
- [18] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.
- [19] T. Coquand and C. Paulin-Mohring. Inductively defined types, preliminary version. In *COLOG '88, International Conference on Computer Logic*, Lecture Notes in Computer Science, Vol. 417, pages 50–66. Springer-Verlag, 1990.
- [20] M. Davis and J. T. Schwartz. Metamathematical extensibility for theorem verifiers and proof checkers. *Comp. Math. with Applications*, 5:217–230, 1979.
- [21] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Proc. of the First Annual Workshop on Logical Frameworks*, pages 280–306, Sophia-Antipolis, France, June 1990.
- [22] S. Feferman. Formal theories for transfinite iterations of generalized inductive definitions and some subsystems of analysis. In *Proc. Conf. Intuitionism and Proof Theory*, pages 303–326, North-Holland, Buffalo, NY, 1970.
- [23] J.H. Gallier. *Logic for Computer Science, Foundations of Automatic Theorem Proving*. Harper and Row, NY, 1986.
- [24] J-Y. Girard. Une extension de l'interpretation de Gödel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symp.*, pages 63–69. Springer-Verlag, NY, 1971.
- [25] J. Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge Tracts in Computer Science, Vol. 7. Cambridge University Press, 1989.
- [26] M. Gordon. HOL: A machine oriented formalization of higher order logic. Technical Report 68, Cambridge University, 1985.
- [27] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, Lecture Notes in Computer Science, Vol. 78. Springer-Verlag, NY, 1979.

- [28] T. Griffin. A formulas-as-types notion of control. In *Proc. of the Seventeenth Annual Symp. on Principles of Programming Languages*, pages 47–58, 1990.
- [29] S. Hayashi and H. Nakano. *PX: A Computational Logic*. MIT Press, Cambridge, MA, 1988.
- [30] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.
- [31] Douglas J. Howe. Equality in lazy computation systems. In *Proc. of Fourth Symp. on Logic in Comp. Sci.*, pages 198–203. IEEE Computer Society, June 1989.
- [32] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proc. of the First Symp. on Logic in Comp. Sci.*, pages 237–248. IEEE, 1986.
- [33] J. Lipton. Logic programming in the Nuprl type theory environment, preprint. Cornell University, Ithaca, NY, 1991.
- [34] Z. Luo. ECC, an extended calculus of construction. In *Proc. of Fourth Symp. on Logic in Comp. Sci.*, pages 385–395, Pacific Grove, CA, June 1989.
- [35] Per Martin-Löf. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, Amsterdam, 1973.
- [36] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–75. North-Holland, Amsterdam, 1982.
- [37] P.F. Mendler. Recursive Types and Type Constraints in Second-Order Lambda Calculus. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 30–36. IEEE, June 1987.
- [38] P.F. Mendler. *Inductive Definition in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1988.
- [39] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symp. on Logic Programming*. ACM, 1987.
- [40] Chetan Murthy. *Extracting Constructive Content for Classical Proofs*. PhD thesis, Cornell University, Dept. of Computer Science, 1990. (TR 89-1151).
- [41] Chetan Murthy. An evaluation semantics for classical proofs. In *Proc. of Sixth Symp. on Logic in Comp. Sci.*, pages 96–109. IEEE, Amsterdam, The Netherlands, 1991.
- [42] Chetan Murthy and J. Russell. A constructive proof of Higman’s Lemma. Technical Report 89-1049, Cornell University, Ithaca, NY, January 1989.

- [43] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*. Oxford Sciences Publication, Oxford, 1990.
- [44] C. Paulin-Mohring. *Extraction in the calculus of constructions*. PhD thesis, University of Paris VII, 1989.
- [45] F. Pfenning. Elf: a language for logic definition and verified metaprogramming. In *Proc. of Fourth Symp. on Logic in Comp. Sci.*, pages 313–321, 1989.
- [46] N Shankar. Towards mechanical metamathematics. *J. Automated Reasoning*, 1(4):407–434, 1985.
- [47] S.F. Smith. *Partial Objects in Type Theory*. PhD thesis, Cornell University, Ithaca, NY, 1989.
- [48] S.F. Smith and R. L. Constable. Partial objects in constructive type theory. In *Proc. of Second Symp. on Logic in Comp. Sci.*, pages 183–93. IEEE, Washington, D.C., 1987.
- [49] R. M. Smullyan. *First-Order Logic*. Springer-Verlag, New York, 1968.
- [50] A. Troelstra. *Metamathematical Investigation of Intuitionistic Mathematics*, Lecture Notes in Mathematics, Vol. 344. Springer-Verlag, 1973.
- [51] R. Weyrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.
- [52] L. Wos, R. Overbeek, L. Ewing, and J. Boyle. *Automated Reasoning*. Prentice-Hall, Englewood Cliffs, NJ, 1984.