Automating Reasoning in an Implementation of Constructive Type Theory

Douglas J. Howe
Ph.D. Thesis

TR 88-925
June 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

Automating Reasoning in an Implementation of Constructive Type Theory

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Douglas James Howe
January 1988

Abstract

The starting point for this thesis is the Nuprl proof development system. Nuprl is an environment for the development of formal computational mathematics and has a rich constructive type theory as a logical basis. It provides sophisticated editors and an integrated tactic mechanism that allows the programming of guaranteed-sound extensions to the inference system.

The work presented in this thesis concerns the automation of reasoning in Nuprl, and consists of three parts. The first is a collection of general-purpose tactics. These tactics are simple enough that their function can be readily understood, yet powerful enough to support development of substantial formal mathematics.

The second part is the use of Nuprl to solve an open problem in the theory of programming languages. The set of basic tactics together with various tools provided by Nuprl play a crucial role in the solution, and it seems that this problem is not tractable without computer assistance.

The third part is an implementation within Nuprl of mechanisms that support the use of Nuprl's type theory as a language for constructing theorem-proving procedures. The main component of the implementation is a large library of definitions, theorems and proofs. This library may be regarded as the beginning of a book of formal mathematics; it contains a complete formal development and explanation of a useful subset of Nuprl's metatheory, and of a mechanism for translating results established about this embedded metatheory to the object level. The type theory, besides permitting the internal development of this partial reflection mechanism, allows us to make abstractions that drastically reduce the burden of establishing the correctness of new theorem-proving procedures. Our library includes a formally verified term-rewriting system.

To Maureen.

Acknowledgements

I consider myself very lucky to have had Bob Constable as an advisor. He has been an excellent intellectual role model, and has been largely responsible for the stimulating research environment that I have enjoyed for the last few years. I am especially grateful for all the support and encouragement he has given me.

I have greatly enjoyed my association with the "Prl group" at Cornell. I learned a good deal from the other members of the group, and enjoyed our many discussions. I especially benefitted from my association with Stuart Allen; I learned more about type theory from him than from any other source.

Contents

Chapter 1

Introduction

The starting point for this thesis is the Nuprl proof development system [23]. Nuprl is an environment for the development of formal computational mathematics and has a rich constructive type theory as a logical basis. The system has a sophisticated proof editor and several decision procedures for integer arithmetic. The programming language ML is incorporated in such a way as to allow the user to write, and safely use, programs called *tactics* that can assist in the construction of proofs. The integration of the tactic mechanism and the proof editor permits the development of proofs that can serve as high-level explanations of formal arguments.

This thesis concerns the automation of reasoning in this novel setting. Our first step in raising the level of inference in Nuprl was to develop a substantial collection of general-purpose tactics. These tactics are simple enough that their function can be readily understood, yet powerful enough to support the development of substantial formal mathematics. Using them, it is usually a straightforward matter to translate an informal proof plan into a formal Nuprl proof, and the resulting proofs are usually understandable pieces of formal mathematics. The tactics have been used in a variety of applications, including some programming examples and a fundamental theorem of constructive number theory. They also make possible the work contained in the rest of the thesis.

One of the main applications we have made of this tactic collection is in the use of Nuprl to solve an open problem in the theory of programming languages. The problem arose in connection with a fundamental question concerning typed λ-calculi. The only known approach to the question was to characterize the computational behaviour of a term obtained from formalizing a non-trivial argument in a minimal calculus. Because of the syntactic complexity involved, others were not able to proceed with this approach. Using Nuprl, we quickly obtained the term of interest and then used the system to analyze its computational behaviour. It seems likely that this problem is not tractable without computer assistance.

Most theorem-proving systems that allow the user to soundly extend the inference mechanism with new theorem-proving procedures use one of two approaches. The first approach is to require that new procedures be proven correct in a formalized metatheory [26, 13, 61, 41]. The second is to provide a tactic mechanism [30, 23], which permits arbitrary new procedures to be written but which requires each application of such a procedure to generate a proof in terms of the primitive inference rules. It appears that only the second approach has been used in significant applications.

We have "boot-strapped" Nuprl to encompass both approaches. We show how Nuprl's powerful type theory can be used to develop a completely internal account of a portion of its metatheory, and how this allows new theorem-proving procedures to be programmed in the type theory. This partial-reflection mechanism and some applications have been implemented as a large Nuprl library. This library may be regarded as a computerized book of formal mathematics. It contains a complete formal development and explanation of what amounts to a substantial theorem-proving system.

The basic idea behind the reflection mechanism is simple. Nuprl's type theory is used to construct natural representations for a certain subclass of the terms of the type theory (roughly, the quantifier-free terms), and for contexts (that is, definitions and hypotheses). Theorems are then proven in Nuprl to connect the representations to the objects represented. This allows the results of computations involving representations to be used in making judgments about the represented objects.

The main application developed in the library is a term-rewriting system. To use the partial reflection mechanism for term-rewriting, one proves that a function f, when applied to a representative of a term satisfying certain properties, results in a second term representative such that the two represented terms are equal. Subsequently, f can be used to rewrite a term t by applying f to a representative u of t, obtaining a representative u' of a term t', and concluding that $t = t'$.

There are two major benefits related to the higher order nature of Nuprl's type theory and to the fact that the reflection mechanism is developed completely within the theory. First, the components of the mechanism are all present within the reflection library. This allows those who build on the library to use the mechanism to justify new classes of theorem-proving procedures. Second, essential abstractions can be made that drastically reduce the burden of proving the correctness of new theorem-proving procedures.

In the next section we give a very brief introduction to Nuprl. We then give a more detailed summary of the thesis. The chapter ends with a discussion of related work and some notes about reading the thesis.

1.1 Nuprl in Brief

Nuprl [23] is a system that has been developed at Cornell by a team of researchers, and is founded on the work of Bates and Constable [5, 6, 7]. It is intended to provide an environment for the solution of formal problems, especially those where computational aspects are important. One of the main problem-solving paradigms supported by the system is that of *proofs as programs*, where a program specification takes the form of a mathematical proposition implicitly asserting the existence of the desired programs. Such a proposition can be formally proved, with computer assistance, in a manner resembling the style of arguments in mathematics. The system can extract from the proof the implicit computational content which is a program that is guaranteed to meet its specification. With this paradigm, the construction of correct programs can be viewed as an applied branch of constructive mathematics.

Apart from the component of Nuprl that is related to the extraction of computational content, much of the system is independent of the constructive character of Nuprl's type theory. Since this type theory is sufficiently expressive to allow a direct treatment, as in Automath [28] and LF [33], of a wide variety of logics (*e.g.*, classical set theories such as ZFC), Nuprl can also be regarded as a system that supports formal reasoning in general. It provides a sophisticated collection of tools for the development, explanation, and manipulation of formal arguments that is intended to support large-scale development of formal mathematics.

The logical basis of Nuprl is a constructive type theory that is a descendent of the type theories of Martin-Löf [45, 46]. The rules of Nuprl deal with *sequents* of the form

$$x_1 : H_1,\ x_2 : H_2,\ \ldots,\ x_n : H_n \ \texttt{>>}\ A.$$

In the context of a proof, sequents are also called *goals.* The terms H_i are *hypotheses* or *assumptions*, and A is the *conclusion* of the sequent. A sequent is said to be true when, roughly, there is a procedure that, under the assumption that H_i is a type for $1 \leq i \leq n$, takes members x_i of the types H_i, and produces a member of the type A. An important point about the Nuprl rules is that they allow top-down construction of the procedure. A goal can be *refined* into *subgoal* sequents such that a procedure for the goal can be computed from procedures for the subgoals.

Nuprl has a large set of type constructors. In addition to such familiar type constructors as cartesian product and disjoint union, there are others whose purpose is to represent basic notions of constructive logic via the *propositions-as-types* correspondence. The basic idea behind this correspondence is that we can associate a type with each logical formula such that the formula is constructively true if the type associated with it has a member. Such a member can be called a *proof* of the proposition, since it contains the computational information that establishes constructive truth. For example, consider the type $x{:}A\texttt{->}B$, where the variable x may occur in B. The members of this type are computable functions that take a member a of A to a member of the type $B[a/x]$. If B represents a proposition P, then $x{:}A\texttt{->}B$ represents $\forall x{:}A\,.\,P$, since the latter is constructively true when there is a procedure which takes a member a of A to a proof of $P[a/x]$.

A statement P of constructive mathematics can be proved in Nuprl by first translating it into the type theory, obtaining a type T, and then attempting to prove >> T by applying refinement rules until no more unproved subgoals exist.

The system can then compute, or *extract*, a term t which is a member of the type T and which embodies the computational content of the theorem P. For example, if we prove in this way the formula

$$\forall x : int \;\; \exists y : int \;\; where \;\; x + y = 0,$$

the term extracted from the proof would be a function which takes an integer x and produces an integer y such that $x + y = 0$. The existence of such a function is the meaning of constructive truth for the formula.

Proofs in Nuprl are tree-structured objects where each node has associated with it a sequent and a refinement rule. The children of a node are the subgoals which result from the application of the refinement rule. For example, an application of the "&-introduction" rule might be displayed by the system as

```
>> A&B  BY intro
  >> A
  >> B.
```

A refinement rule is either a primitive inference rule, such as in the above example, or a program written in the ML language [30]. Such a program is called a refinement *tactic* (being similar to an LCF tactic [30]), and when given a sequent as input it applies primitive inference rules and other tactics to build a proof tree with the sequent as the root. This resulting proof tree is hidden except for its unproved leaves; these become the children of the input sequent, and the name of the tactic becomes its associated rule. Tactics can thus act as derived inference rules. These derived rules are guaranteed to be correct because of the way the type structure of ML is used.

There is one substantial decision procedure which is not a tactic. It is built into the Nuprl system and is invoked as the primitive inference rule `arith`. This procedure incorporates a congruence closure algorithm [42], an integer-expression normalizer and a procedure developed for use in the PL/CV system [20]. It only deals with expressions over the integers. There are two other special rules dealing with the integers; these simply encapsulate some commonly used facts about some of the built-in operations and relations over the integers. With the exception of these special rules, all of the automation of reasoning in Nuprl that has been done to date has been accomplished using the mechanisms provided by the system.

Interactions with Nuprl are centred on the *library*. A library contains an ordered collection of definitions, theorems and other objects. New objects are constructed using special-purpose window-oriented editors. The text editor and the definition facility support very readable notations for objects of Nuprl's type theory. The proof editor is used to construct proofs in a top-down manner, and to view previously constructed proofs. Proofs are retained by the system, and can be later referred to either by the user or by ML programs.

1.2 Automating Reasoning in Nuprl

Nuprl as just described cannot be immediately applied to significant developments of formal mathematics. For non-trivial theorems, formal proofs consisting of user-directed applications of primitive inference rules would be too lengthy and tedious to be feasible. The main problem addressed by this thesis is that of raising the level of inference in Nuprl.

Our work is guided by a paradigm of formal reasoning which is centred about the user. In this paradigm, the user does *not* play the role of an oracle to computer-led search for a proof, supplying guiding information when requested to by a program pursuing a predetermined strategy for proof-discovery. Rather, he starts with an informal proof plan involving natural kinds of reasoning, and then directs the system in an attempt to construct a formal proof whose structure resembles that of the informal plan. Ideally the system would provide for concise expression of "high-level" proof-steps, and would automatically prove "trivial" facts. The meanings of "high-level" and "trivial" cannot be fixed in advance; they vary not only with the problem domain, but within domains as well. During the development of mathematics not only are proven facts accumulated, but also proof techniques. Thus the automation of reasoning requires not only a collection of useful procedures, but also support for the construction of new procedures.

A First Step: Basic Tactics

We have constructed a large collection of general-purpose tactics and other ML objects. This collection can be roughly divided into two categories. Those in the first category amount to derived inference rules encapsulating common patterns of reasoning. Those in the second category are related to a general purpose tactic called the *autotactic*, which is described later. The tactics in the first group are for the most part fairly unambitious, in that they do not include any powerful theorem-proving procedures, and do not make much use of heuristics. They have well-defined functions, are readily understood, but support a reasonably high level of general reasoning. More specifically:

- *They are predictable.* Their actions are sufficiently predictable to make it straightforward to select the appropriate tactics during the course of a proof. Usually little experimentation is required in order to find the tactics needed to implement a proof plan.

- *They can be easily combined.* It is often possible, using the basic tactics and a small set of combinators, to quickly construct special purpose tactics for particular problems.

- *The resulting proofs are readable.* Since their functions can be readily understood, formal proofs that contain them are intelligible. The level of inference that they support often permits formal proofs to bear a strong resemblance to the informal proofs on which they are based.

The autotactic is so-called because the system will apply it automatically after every invocation of a primitive inference rule. Also, users usually explicitly call it as the last part of a proof step that uses a refinement tactic. It is the purpose of this tactic to prove most of the "trivial" subgoals that other tactics produce. Most of these subgoals involve showing that some term is a type or is a member of a type; proving these kinds of subgoals is a problem in Nuprl because, for example, it is recursively undecidable whether a term denotes a type. Since most of the tactics of the first category produce many trivial subgoals that are not easily conceptually related to the main function of the tactics, the structure of a formal proof (as well as the time taken to complete it) would be seriously degraded if these subgoals were not usually proven automatically. Fortunately, the autotactic has in practice been able to prove the vast majority of such subgoals. As a simple example, consider the introduction step

```
>> A->B   BY (I ...)
  A >> B.
```

In this example, `I` is a tactic that simply choses the appropriate introduction rule to apply. The three dots indicate the use of a Nuprl definition that has the effect of applying the autotactic after `I`. Here we see that the single subgoal has just the result we would expect. However, the introduction rule that the tactic `I` invoked also generates a subgoal to show that A is a type; this subgoal was proved by the autotactic.

It is difficult to describe the level of inference that our tactic collection supports. Most of the tactics perform functions which are rather low-level, especially when compared to theorem-proving systems that incorporate some powerful mechanism such as resolution or heuristically guided rewriting. Although the tactics can be applied in such a way as to provide a reasonably high level structure to a proof, such proofs will in general exhibit a high degree of user-direction. On the other hand, the collection has been sufficiently powerful to permit the rapid formalization of significant pieces of mathematics.

Some of the applications made of the tactic collection will be outlined in the next two subsections; more detailed accounts will appear in Chapters 3 and 5. The remainder of this subsection contains a brief description of some of the highlights of the collection. See Chapter 3 for details.

Among the tactics not related to the autotactic are tactics for the following.

- *Computation.* These tactics allow the user to perform certain kinds of symbolic computation on Nuprl goals. For example, there are tactics to normalize terms in a sequent, and to expand definition instances.

- *Rule application.* Some of these attempt to compute the parameters needed by certain inference rules; others perform repetitive applications of certain rules.

- *Simple derived rules.* This group comprises many unrelated tactics each of which combines a small number of rules in some relatively fixed pattern. For example, there is a tactic which makes available the properties of a specified term that follow from membership in a subtype.

- *Chaining.* There are tactics for both forward and backward chaining. In backward chaining, known facts (lemmas or hypotheses) are treated as inference rules and applied backwards. As a simple example, if we know $A \Rightarrow B$, then to prove >> B it suffices to prove >> A. In forward chaining, the applications are in a forward direction, so that new hypotheses are inferred from old. Both forms use sophisticated (but essentially first order) matching techniques to determine when a known fact is applicable and, if so, what terms are required to instantiate any universal quantifiers.

- *Simple rewriting.* These tactics allow certain kinds of lemmas that assert equalities to be used as (conditional) rewriting rules.

- *Application of "pattern" theorems.* One can partially prove theorems that contain what are tantamount to syntactic variables, and then use these proofs as patterns in constructing other proofs.

The most important single tactic is the autotactic. One of its most important functions is to prove *membership* goals of the form

$$>> t \texttt{ in } T.$$

There is no algorithm which decides whether or not statements are true (or provable). Fortunately, in practice the tactic called `Member` has been effective in dealing with such goals. `Member` is an interactive typechecker that can be extended, via ML library objects, to incorporate special methods for those membership tasks that are not already adequately handled. It can usually automatically prove goals to which it is applied, and when it cannot it usually presents the user with subgoals that are simpler than the goal, and that are provable if the goal is (that is, it does not go too far in its search for a proof). It can deal with partial functions (whose domain is a subtype), and with implicitly polymorphic functions (functions that can be applied to objects from a range of types, but that do not take the relevant type as an argument). In addition to `Member`, the autotactic includes components for trivial propositional reasoning and for proving subgoals involving integer arithmetic. The latter component uses the powerful built-in decision procedure `arith`.

Applications

The tactic collection (or earlier versions of it) has been used fairly extensively by the author and others. The applications of it made by the author include the following.

- *Quicksort.* A theorem was proven whose extraction is the "quicksort" sorting algorithm.

- *Regular sets.* A small piece of the theory of regular sets was developed. The main theorem proved was that for any regular sets r and s, $(r+s)^* = (r^*s^*)^*$. The development was semantic rather than axiomatic: regular sets were taken to be Nuprl sets (types), and the operations of regular set theory were defined instead of assumed.

- *The pigeon-hole principle.* We proved that every injection from a finite set to a smaller set must map two distinct members of its domain to the same member of its range.

- *The fundamental theorem of arithmetic.* We proved that every positive integer has a unique prime factorization. The proof yields via extraction a program which computes the factorization.

- *Saddleback search.* We solved a problem from Gries' book *The Science of Programming* [31]. The problem involves efficiently searching a sorted matrix.

- *Girard's paradox.*

- *Partial reflection, term-rewriting, etc.*

Below are brief discussions of the saddleback search problem and Girard's paradox. The last application above is the topic of the next section, and the last two applications have separate chapters devoted to them. The basic tactics have also been used by others; for example, in [17] Cleaveland used them to develop the theory of synchronization trees and an interpretation of Hennessy-Milner logic.

Saddleback Search

As part of a "verification assessment study" lead by Kemmerer [39], various program verification systems were applied to a "randomly selected" problem from Gries' *The Science of Programming.* The problem, whose solution has been called "saddleback search", is to efficiently find the location of a given integer in a matrix of integers whose rows and columns are sorted in non-decreasing order. The complete solution, including planning, took one afternoon. A detailed description of the solution is given in Chapter 3. A brief description is given here in order to illustrate the use of the basic tactic collection, and to provide a concrete example of proof construction in Nuprl.

The library that was constructed contains nine objects in addition to a set of basic definitions that is shared by most Nuprl libraries. The first of the nine is a simple definition for matrix application:

$$B(i,j) \equiv B(i)(j)$$

This is not precisely as the definition would appear if it were viewed on a Nuprl screen; the parameters here are the italicized identifiers, whereas Nuprl uses angle brackets (*e.g.*, `<B>` instead of B). The second two objects (a theorem and a definition) together give a definition of the proposition that an integer x occurs in an m by n matrix B in a column between i and p and in a row between j and q:

```
x∈B{m,n}(i:p, j:q)  ≡
∃r:{0..(m-1)}. ∃s:{0..(n-1)}. where B(r,s)=x & i≤r≤p &  j≤s≤q.
```

The next two objects in the library define the property of an m by n matrix B being sorted:

```
sorted(B){m,n}  ≡
∀j:{0..(n-1)}. ∀i,p:{0..(m-1)}. i<p  =>  B(i,j)≤B(p,j) &
∀i:{0..(m-1)}. ∀j,q:{0..(n-1)}. j<q  =>  B(i,j)≤B(i,q).
```

Following this are two ML objects which contain the definitions of two simple tactics called `RowSorted` and `ColSorted` which are used to apply the fact that a matrix is sorted.

The final two objects in the library are the main theorem and a lemma. The statement of the main theorem is

```
∀m,n:N+. ∀B:({0..(m-1)}->{0..(n-1)}->Int) where sorted(B{m,n}).
  ∀x:Int. x∈B{m,n}(0:(m-1),0:(n-1)) ∨ ¬(x∈B{m,n}(0:(m-1),0:(n-1)))
```

The program extracted from a proof of this theorem is a function that when given inputs of the appropriate type produces an indication of whether or not x occurs in the matrix B together with, in the former case, the pair of integers which give the location of x.

The algorithm implemented by our proof proceeds as follows. Start at the lower left corner of the matrix (*i.e.*, at position $\langle m-1, 0\rangle$). If x is equal to the matrix element b at this position, then we are done. If $x < b$ then since the matrix is sorted x must be smaller than everything in the last row, so we can throw away the last row and repeat the procedure with the smaller matrix. Similarly, if $x > b$, then we can discard the first column and repeat. This algorithm has a simple recursive structure, where a solution for an upper-right block of the matrix can be found in terms of a solution for a smaller block obtained by deleting the bottom row or the leftmost column. This is the motivation for the lemma:

```
∀m,n:N+. ∀B:({0..(m-1)}->{0..(n-1)}->Int) where sorted(B{m,n}).
   ∀x:Int.∀k:N. ∀i:{0..(m-1)}. ∀j:{0..(n-1)} where i+(n-j) = k.
      x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))
```

This can be interpreted as asserting that for every k, we can find whether or not x occurs in any upper-right block of B that has a total of $k-1$ rows and columns.

```
* top
>> ∀m,n:N+. ∀B:({0..(m-1)}->{0..(n-1)}->Int) where sorted(B{m,n}).
     ∀x:Int. ∀k:N. ∀i:{0..(m-1)}. ∀j:{0..(n-1)} where i+(n-j) = k.
       x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

BY (OnVar ‘k‘ (NonNegInd ‘l‘) ...)

1* 1. m: N+
   2. n: N+
   3. B: {0..(m-1)}->{0..(n-1)}->Int
   4. x: Int
   [5]. sorted(B{m,n})
   6. i: {0..(m-1)}
   7. j: {0..(n-1)}
   [8]. i+(n-j)=0
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

2* 1. m: N+
   2. n: N+
   3. B: {0..(m-1)}->{0..(n-1)}->Int
   4. x: Int
   5. l: int
   6. 0<l
   7. ∀i:{0..(m-1)}. ∀j:{0..(n-1)} where i+(n-j)=l-1.
      x ϵ B{m,n}(0:i, j:(n-1)) ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))
   [8]. sorted(B{m,n})
   9. i: {0..(m-1)}
   10. j: {0..(n-1)}
   [11]. i+(n-j)=l in Int
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))
```

Figure 1.1: Proof by induction.

```
* top 2
...
>> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

BY (Cases ['B(i,j)<x'; 'B(i,j)=x'; 'x<B(i,j)'] ...)

1* 12. B(i,j)<x
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

2* 12. B(i,j)=x
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

3* 12. x<B(i,j)
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))
```

Figure 1.2: The main case split.

The proof of this lemma is by induction on k. The first step is shown in Figure 1.1. What is shown is similar to what would be displayed by Nuprl if the step were to be viewed using the proof editor. The asterisk in the upper left of the figure indicates that the subproof below is complete, and the text to the right of it gives an address within the proof tree for the displayed node (`top` indicates that the node is the root of the tree). Below this are the sequent which is the *main goal* of the node, the tactic applied at the node (the text following `BY`), and the *subgoals* generated by the tactic. The tactic used here performs induction on k (using l as a new variable for the induction step). As mentioned earlier, the notation $(T\ldots)$ indicates that the autotactic is applied after the tactic T.

The first subgoal in Figure 1.1 is proved in two steps. The beginning of the proof of the induction step is shown in Figure 1.2. The hypotheses numbered 1 to 11 in the first figure have been elided in Figure 1.2. In this step, we are considering the block of the matrix that extends from the top to row i and from column j to the right. As in the informal description of the algorithm, we do a case analysis on the relation between x and $B(i,j)$.

The complete proof of this lemma has 21 steps. The proof of the main theorem is just an application of the lemma and has 2 steps. For more on this proof, see Chapter 3.

Girard's Paradox

For the purpose of studying the effect of the addition of a type of all types to programming languages with dependent types Meyer and Reinhold in their paper *"Type" Is Not a Type* defined λ^{Π}, a polymorphically typed λ-calculus with dependent types, and $\lambda^{\tau\tau}$, the calculus obtained from λ^{Π} by adding the axiom $\tau \in \tau$, where τ

represents the type of all types. They claimed that by following the proof of Girard's paradox [29] they could construct in $\lambda^{\tau\tau}$ a polymorphic fixed-point combinator

$$Y \in \Pi\, t{:}\tau\,.\,(t \to t) \to t$$

such that for any type t and any $f \in t \to t$

$$Ytf = f(Ytf).$$

They then showed that the existence of such a Y had some important implications for $\lambda^{\tau\tau}$:

- A non-normalizing term is derivable.
- Equality of terms is undecidable.
- $\lambda^{\tau\tau}$ does not conservatively extend base-type theories, so that classical reasoning about base-type objects is not valid in the programming language formed by $\lambda^{\tau\tau}$ together with base-types.

However, they were not able to establish the existence of Y. It appears that the problem may involve so much formal detail that a solution without mechanical assistance is infeasible. A non-trivial proof must be carried out in a formal system in which even the basic notions of logic need to be encoded, and the behaviour under reduction of a very large term must be characterized.

In [24], Coquand exhibited several theories in which Girard's paradox could be formalized. He verified these assertions by using a mechanical proof checker, but did not analyze the computational properties of the proof.

In [57], Reinhold considered an extension of $\lambda^{\tau\tau}$ that included the natural numbers (together with a recursion combinator) and a dependent sum (or sigma) type constructor. He outlined a proof that by following Martin-Löf's simplification of the proof of the paradox [44], it was possible to obtain in the extension a term Y, of the proper type, such that there exist terms Y_n, $n \geq 0$, such that $Y_0 \equiv Y$ and for all n

$$Y_n tf = f(Y_{n+1} tf).$$

He called a term satisfying this property a *looping combinator*, and conjectured that an analogous construction in $\lambda^{\tau\tau}$ would give a fixed-point combinator. If the fixed-point property cannot be obtained, then the argument in [49] establishing the failure of conservative extension for $\lambda^{\tau\tau}$ is no longer valid. However, the looping property is enough to prove both the undecidability of equality and the existence of a non-normalizing term [56].

In Chapter 4 is a proof that at least one particular formalization of Girard's paradox yields a term which is a looping combinator and *not* a fixed-point combinator. The computer plays an essential role in the argument. Nuprl and the basic tactic collection were used to deal with the vast amounts of syntactic detail that an analysis of the computational behaviour of the paradox seems to entail. It was straightforward (taking less than two days) to formalize a proof of the paradox in a subset of the Nuprl type-theory that is similar to $\lambda^{\tau\tau}$. Because of extraction, it was possible to construct the desired term *implicitly*; the formal proof, although providing the construction of a massive term, bears a strong resemblance to the conventional proof on which it is based. The proof that the resulting term is a looping but not a fixed-point combinator can be reduced to the verification of a small number of machine-checkable assertions about reduction. These verifications were carried out using Nuprl's term-manipulation facilities. The argument that is given in Chapter 4, then, is not a complete proof, in that it depends on the correctness of Nuprl's implementation. More specifically, if the formal argument carried out in the system constitutes a valid proof in the Nuprl type theory, and if the implementation of ML is correct insofar as it was used to perform certain reduction sequences and to verify certain simple properties of some Nuprl proofs, then the resulting term has the properties we claim.

A New Class of Tactics

The basic tactics described earlier have been sufficient for several substantial applications. However, the tactic mechanism has a severe limitation. As a simple example, consider equations involving an associative commutative operator "$\cdot$". One way to check that an equation

$$a_1 \cdot a_2 \cdot \ldots \cdot a_m = b_1 \cdot b_2 \cdot \ldots \cdot b_n$$

holds is to sort the sequences $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ with respect to some ordering on terms, and check that the results are identical. A tactic that emulated this informal procedure would have to chain together appropriate instances of lemmas (for example, for the associativity and commutativity of $\cdot$) and rules (for example, the substitution rule). This indicates two major problems. First, the tactic writer must be continually concerned with generating Nuprl proofs, and this can increase the intellectual effort involved in constructing theorem-proving procedures. Secondly, there is a major efficiency problem. In the example, even though it is known in advance that the sorting algorithm is sufficient to establish equality, every time the tactic is called it must re-justify the algorithm.

The mechanisms we have implemented allow procedures such as the one just described to be used directly. The main part of our implementation consists of a large collection of Nuprl definitions, theorems and proofs. This *library* may be regarded as the beginning of a book of formal mathematics; it contains the formal development and explanation of a useful subset of Nuprl's meta-theory, and of a mechanism for translating results established about this embedded meta-theory to the object level. The most important application of this is to the automation of reasoning: one can write Nuprl programs that directly encode such meta-level procedures as the sorting-based algorithm given above, prove that the program is correct, and then apply it in the construction of Nuprl proofs. Applications of such programs can be done in a manner consistent with Nuprl's proof development paradigms.

The basic idea of the reflection mechanism is simple. Using Nuprl's recursive-type constructor, we define a type `Term0` that represents a certain subset of the terms of Nuprl's theory. Definitions and hypotheses are represented by *environments*, which are lists associating atoms with objects that can be thought of as the values of the atoms. We then define what it means for a member of `Term0` to be well-formed with respect to an environment. Let `Term`(α) be the type of all members of `Term0` that are well-formed with respect to α. We can demonstrate the existence of Nuprl functions `type` and `val` such that for any α,

$$\texttt{type}(\alpha) \quad \texttt{in} \quad \texttt{Term}(\alpha)\texttt{->SET}$$

and

$$\texttt{val}(\alpha) \quad \texttt{in} \quad \texttt{t:Term}(\alpha)\ \texttt{->}\ \texttt{type}(\alpha)\texttt{(t)},$$

where a member of `SET` is a type together with an equality relation on that type.

To show how the above can be put to use we consider a simplified example. Suppose that we have constructed a function f of type

```
Term0 -> Term0
```

and that it preserves equality in some environment α. More precisely, suppose that for any term t in `Term`(α),

$$\texttt{type}(\alpha)(t)\ \texttt{=}\ \texttt{type}(\alpha)(f(t))\ \texttt{in SET}$$

and

$$\texttt{val}(\alpha)(t)\ \texttt{=}\ \texttt{val}(\alpha)(f(t))\ \texttt{in}\ \texttt{type}(\alpha)(t)$$

(the second equality means that the equivalence relation from `type`(α)(t) is satisfied). For example, f might be based on the sorting procedure mentioned earlier. Suppose that we are in the course of proving a Nuprl theorem, that the current goal is of the form `>>` T (`>>` is Nuprl's turnstyle), and that we want to "apply" f to T. The first step is to *lift* the goal. To do this, we apply a special tactic which takes as an argument the environment α. This tactic first computes a member t of `Term`(α) such that the application `val`(α)(t) computes to a term identical to T, and then generates the subgoal

$$\texttt{>>}\ \texttt{val}(\alpha)(t).$$

We now apply a tactic which in effect computes $f(t)$ as far as possible, obtaining a value $t' \in$ `Term`(α), and produces a subgoal

$$\texttt{>>}\ \texttt{val}(\alpha)(t').$$

At this point, we can proceed by applying other rewriting functions, or by simply computing the conclusion and obtaining an "unlifted" goal >> T' to which we can apply other tactics.

An important aspect to this work is that the partial reflection mechanism involves no extensions to the logic; the connection between the embedded meta-theory and the object theory is established completely within Nuprl. This means, first, that the soundness of the mechanism has been formally verified. Secondly, there is considerable flexibility in the use of the mechanism. All of its components are present in the Nuprl library, so one can use them for new kinds of applications without having to do any metatheoretic justification. Rewriting functions are just an example of what can be done; many other procedures that have proven useful in theorem proving, congruence closure for example, can be soundly added to Nuprl. Thirdly, the mechanism provides a basis for stating new kinds of theorems. For example, one can formalize in a straightforward way the statement of a familiar theorem of analysis: *if $f(x)$ is built only from x, $+$, $\cdot$, ..., then f is continuous.*

Finally, and perhaps most importantly for the practicability of this approach, it is possible to make abstractions that drastically reduce the burden of verifying the correctness of new procedures. Since contexts (environments) are represented, one can prove the correctness of a procedure once for a whole class of applications. For example, a simplification procedure for rings could be proven correct for any environment in which the values of certain atoms satisfy the ring axioms, and then be immediately applied to any particular context involving a ring. A general procedure such as congruence closure could be proved once and for all to be correct in any environment whatsoever. Another kind of abstraction is indicated by the rewriting example to be given later. Functions such as `Repeat` are proved to map rewriting functions to rewriting functions; these kinds of combinators can often reduce the proof of correctness for a new rewriting function to a simple typechecking task that can be dealt with automatically.

One of the original motivations of this work was to provide tools that would be of use in formalizing Bishop-style constructive real analysis [8]. The design of the reflection mechanism incorporates some of the notions of Bishop's set theory. However, our work is more generally applicable, since Nuprl's type theory is capable of directly expressing problems from many different areas. It should be of use in other areas of computational mathematics (*e.g.*, in correct-program development). Using the techniques developed in AUTOMATH [27] and LF [33], it can be applied to reasoning in a wide variety of other logics, constructive in character or not. For example, any first-order logic can be embedded within Nuprl; in such a case, the reflection mechanism will encompass the quantifier-free portion of the logic.

We will now look at an example actually developed in the reflection library, where a simple rewriting function is applied to an equation involving addition and negation over the rational numbers. The environment used is α_`Q`, which contains "bindings" for rational arithmetic. For example, it associates to the Nuprl atom `"Q"` the set `Q` of rational numbers, and to the atom `"Q_plus"` the addition function. The rewriting functions for this environment are collected into the type `Rewrite(α_Q)`. The requirement for being a rewriting function over `α_Q` is actually stronger than indicated earlier; the criteria given there must hold for any α which extends `α_Q`.

The member of `Rewrite(α_Q)` we will apply is

```
Repeat( TopDown( rewrite[x;y]( -(x+y) -> -x+-y ) ) ),
```

(this function is named `norm_wrt_Q_neg`). This does what one might expect: given a term, it repeatedly, in top-down passes, rewrites subterms of the form `-(`x`+`y`)` to `-`x`+-`y. The functions `Repeat` and `TopDown` both have type

```
Rewrite(α) -> Rewrite(α)
```

for any α. The argument to `TopDown` in our example rewriting function is by definition the application of a function of three arguments: a list of atoms that indicate what constants are to be interpreted as variables for the purpose of matching; and two members of `Term0` which form the left and right sides of a rewrite-rule. Definitions are used to make members of `Term0` appear as close as possible to the terms they represent. The term `x+y` that appears in the definition of `norm_wrt_Q_neg` is, when all definitions are expanded,

```
inl( < "Q_plus", inl(<"x",nil>) . inl(<"y",nil>) . nil > ).
```

The sequent we apply this function to is

```
>> -(w+x+y*z+z) = -w+-x+-(y*z)+-z in Q
```

(the hypotheses declaring the variables to be in `Q` have been omitted). The first step is to apply the tactic

```
(LiftUsing ['α_Q'] ...),
```

which generates the single subgoal

```
α:Env, (...)
>> ↓( val( α, -(w+x+y*z+z) = -w+-x+-y*z+-z in Q ) ).
```

The notation `(...)` indicates that the display of a hypothesis has been suppressed by the system, and the *squash* operator (↓) will be explained later. This elided hypothesis contains (among other information) the fact that α is equal to an environment formed by extending `α_Q` with entries for representatives of the variables `w`, `x`, `y` and `z`. The next step is to apply the tactic

```
(RewriteConcl 'norm_wrt_Q_neg' ...),
```

which generates a single subgoal with conclusion

```
↓( val( α,-w+-x+-(y*z)+-z = -w+-x+-(y*z)+-z in Q ) ).
```

This subgoal can be proved either by a procedure that does simple equality reasoning on lifted sequents, or by normalizing the conclusion and applying the usual autotactic.

The Nuprl library containing the reflection mechanism and some applications contains more than 1300 objects, and is by far the largest development undertaken so far using Nuprl. The theorems were proved using the basic tactics outlined earlier. A description of the library is contained in Chapter 5. An ideal presentation of the library would involve the use of the Nuprl system itself, since the system is designed not only for the construction of formal arguments, but also for their presentation. The best that can be done in this thesis is to explain the basic definitions and theorems, and to give some idea of the character of the formal proofs.

About two thirds of the library is devoted to necessary preliminaries (such as list theory) and to basic facts about the partial reflection mechanism. Although this part of the library involves a large number of theorems, the effective content is small as far as a user is concerned. Most of the theorems are internal to the development of the main mechanism. Most of the remainder fit into a small number of categories. For example, there are a large number of "monotonicity" lemmas, which concern the relation of certain objects to environment extension. One such lemma states that if α_2 extends α_1, then any term which is well-formed in α_1 is well-formed in α_2.

The remaining third of the library contains some applications of the partial-reflection mechanism. The main application is to term-rewriting. The type of rewriting functions is defined as above, and a powerful set of operations for constructing rewriting functions is established. The basic operation is for turning an equational fact into a rewrite based on first-order matching. The other operations are combinators for combining rewrites. These operations incorporate a notion of *failure*, as in ML; this is the main control mechanism for rewriting strategies. The result is a language for rewrite-programming whose implementation has been formally verified.

There are also two smaller implemented applications. First, as an example of a procedure that works on an entire lifted sequent, there is an equality procedure that takes equalities from the lifted hypotheses, and decides whether the equality in the conclusion follows via symmetry, transitivity and reflexivity. Secondly, there is an implementation of a version of the sorting algorithm described earlier. This algorithm normalizes expressions over a commutative monoid (a set together with an identity element and a commutative associative binary operation).

In order to give an idea of the range of possible applications we present a few details about representation. The type `Term0` of *meta-terms* is defined recursively. A meta-term is either

1. an integer, or

2. a "function application" consisting of an atom (representing a function) together with a list of meta-terms (representing a list of arguments), or

3. an "equality" consisting of a pair of meta-terms and an atom (where the atom represents the set from which the equality is taken), or

4. an "injection", consisting of an atom and a pair of meta-terms.

This last kind of meta-term is motivated by Bishop's approach to constructive mathematics, and is intended to allow the representation of terms which contain a certain kind of partial function. In his constructive set theory, the subset property has computational significance, and partial functions generally require as an argument, in addition to a member of a subset, evidence that the member is in the subset. For example, division over the reals of x by y requires an integer $n > 0$ where $1/n < |x|$.

Environments contain the information required to give meaning to meta-terms. An environment consists of a type environment and a function environment. The former associates atoms with sets. The latter associates an atom with a *meta-type* that is formed from atoms which have meaning in the type environment, and with a value from the meaning of the meta-type. The environments also contain other information which will be discussed later. A meta-type has a simple structure: it is an atom (representing the result type of an application) together with a list of atoms (representing argument types). A meta-term is well-formed with respect to an environment α if all of the atoms in it are appropriately bound in α, and if the meta-types match. What is meant by "match" is defined precisely in Chapter 5; only a simple example will be given here. Suppose the meta-term t is a function application with atom f and with a single argument meta-term a. An atom A, representing a result type for a, can be computed from a and α, and an atom A' representing the domain type of f can be obtained from the entry for f in α. The term t is well-formed if a is well-formed, and if either A and A' are identical, or the value of A is a subset (in a sense to be made precise later) of the value of A', or if the value of A' is a subtype of the value of A and the value of a is a member of A'. Well-formedness is in general undecidable; however, it can usually be proved without user assistance.

There are some questions concerning the practicability of the reflection approach that I am not yet able to fully answer. Some small examples have been developed, but conclusive answers will not be possible until the approach is tested in a major application. In my opinion, it will be a useful tool for the implementation of a significant portion of Bishop's book on constructive analysis [8], but such an undertaking is beyond the scope of this thesis.

One of the questions is whether Nuprl encodings of reasoning procedures are adequately efficient. The data types these procedures operate on are not significantly different from those used to implement the Nuprl theory, so this question reduces to the open question of whether Nuprl programs in general can be made efficient. In the example given earlier (involving rational arithmetic), the actual rewriting (the normalization of the term that is the application of the rewriting function to its argument) took about ten seconds. The normalization was done using Nuprl's evaluator, which employs a naive call-by-need algorithm and uses extracted programs directly (*i.e.*, no optimizations are done).

Another question is whether lifting a sequent and maintaining a lifted sequent are sufficiently easy. Currently, these operations are somewhat slow. For instance, in the example, the lifting step and the rewriting step each took a total of about two to three minutes. The main reason for this problem has to do with certain gross inefficiencies associated with Nuprl's term structure and definition mechanism. Solutions are known and awaiting implementation. The details in Chapter 5 show that the operations are not inherently slow.

The most difficult and important question is whether it is feasible to formally verify in Nuprl significant procedures for automating reasoning. The term rewriting system gives some positive evidence, as do some of the other examples outlined above that point out the utility of the means for abstraction available in Nuprl.

1.3 Comparison with Other Work

The work presented in this thesis builds on the approach to automating reasoning that is embodied in the design of Nuprl [21, 23]. For a discussion of how this approach fits into the spectrum of methods for automating reasoning, see [21]. In this section, we will contrast our work with other research involving tactics, and with other approaches to using formalized meta-theory for automating reasoning. Some comparisons specific to the analysis of the computational behaviour of Girard's paradox can be found in Chapter 4.

LCF

The design of Nuprl was strongly influenced by the Edinburgh LCF system [30]. The ML language and the first tactic mechanism were designed as part of LCF. The main difference between tactics in Nuprl and LCF is that in Nuprl they are incorporated into the structure of proofs. In LCF, theorems are proved in a dialogue with ML. A list of unproved subgoals is maintained, and the user proceeds by supplying a tactic to be applied to a specified member of the list. The tactic possibly generates further subgoals, adding to the list. The user continues until the list is empty. At this point, the proof is complete, and the set of tactic calls made implicitly provides the construction of a proof object. However, these proof objects (which cannot actually be constructed) would be formed solely from applications of primitive inference rules; any abstraction provided by the tactics that were used would be lost. The level of detail in such a proof would in general completely obscure its structure.

A record of an LCF proof can be formed from the tactics used to construct the proof. Using *tacticals*, these tactics can be combined into a single tactic which expresses the complete proof strategy. In such a proof summary, no intermediate results appear; this is similar to what would be obtained if all the sequents were erased from a Nuprl proof, and is inadequate as an explanation of a formal proof.

Some of the differences between the LCF and Nuprl systems have a significant bearing on the problem of designing tactics. First, PPLAMBDA, the logic of LCF, is very different from Nuprl's constructive type theory. It is based on a domain-theoretic view of computation, and incorporates a classical predicate calculus. Secondly, Nuprl makes it possible for proofs that are constructed by tactics to serve as explanations of formal arguments. This gives rise to a new consideration in the design of tactics. Finally, there are often important requirements for the structure of a formal proof. For a theorem that is the specification of a program, a proof will yield via extraction a program meeting the specification. Often one wishes to implement a specific algorithm, and so not just any proof will do. Mostly because of these differences, the tactics that we have written have quite a different character than those written for use in LCF [18, 50, 54, 53, 55, 58]). Details are given in Chapter 3.

The tactic mechanism of LCF suffers the same serious drawback as that of Nuprl: tactics must always in effect construct proofs that use only the primitive inference rules. For example, in Paulson's implementation of rewriting in LCF [53], rewriting proceeds with chains of lemma applications and instances of the substitution rule. In our implementation (part of which is inspired by Paulson's work) rewriting is just direct term manipulation.

Metatheoretic Extensibility

A theorem-proving system is *metatheoretically extensible* if it can be extended by new inference rules or procedures that are justified in a formalized metatheory. In a sense, Nuprl, together with the partial reflection mechanism we have implemented, does not fall in this class. Inference procedures that are developed and proved correct using the mechanism do not extend the type theory since the entire mechanism is developed *within* the theory. However, the same basic idea underlies our work and some previous systems that incorporate metatheoretic extensibility.

In [26], Davis and Schwartz described the design of a system called VT that allows the sound addition of new inference rules. Their system is based on a formal system FS that resembles set theory and that has a decidable subtheory LFS. It is possible to represent within the FS theory the proof system of FS. To add a new inference rule to the system, one writes an LFS predicate Φ which recognizes instances of the rule, and then proves in FS that whenever Φ is true of the representations of formulas $A_1, \ldots, A_n$ and B, there is a representation of an FS proof of B from assumptions $A_1, \ldots, A_n$. The decidability of LFS allows the system to incorporate Φ into its proof checking procedure.

In [61], Weyrauch described FOL, a system that is based on first order logic and that has a separate meta-theory. Mathematical theorems are proved in a theory whose language and proof system are represented as constants in another theory. A reflection principle allows information to be passed between the two theories, so that a theorem in the first can be proved by applying a meta-theorem to its representation in the second. These meta-theorems state that a proof exists whenever some predicate is true of (the representation of) a formula. The predicate applications can usually be simplified to the constant *true* or *false.* A *semantic attachment* mechanism is provided so that the user can speed up the simplification procedure by attaching lisp programs to logical constants. There is no requirement that the correctness of the programs be proven, and so FOL's extensibility mechanism is unsound.

Both FOL and VT involve the formalization of a *proof system.* In our approach, constructing an inference procedure and proving its correctness involves reasoning about meta-terms and their meanings; in FOL and VT one additionally must be concerned with the construction of objects representing proofs. FOL has the further complication of having a separate metatheory. A more important distinction between our work and both FOL and VT has to do with programming. For a system to have a practical metatheoretic-extensibility mechanism, it must provide strong support for the construction of correct programs. In VT, one codes procedures as formulas in a sublanguage of set theory (although Davis and Schwartz do show how in theory one could incorporate an arbitrary programming language into VT). The formal metatheory of FOL does not contain a programming language, although the system's simplifier can turn certain equational specifications into rewriting programs. On the other hand, Nuprl was specifically designed for solving problems of computational significance. Nuprl's rich type theory supports the proof-as-programs paradigm and higher-order programming. The latter feature plays a crucial role in the term-rewriting system we have constructed, as well as in the construction of the partial reflection mechanism itself.

The work not involving Nuprl which is closest to ours is Boyer and Moore's implementation of *metafunctions* [13] in their theorem-proving system [14]. They axiomatize a one place predicate `FORMP` and a two place function `MEANING` which roughly correspond to our well-formedness predicate and evaluation function, respectively. The axiomatizations are parameterized by the set of definitions that exist in the current state of the prover. `FORMP` is axiomatized to be true of all Lisp s-expressions that represent a formula that is well-formed in the current state of the prover. `MEANING`(t,A), where t represents a term and A associates variable representatives with other objects, is axiomatized to be equal to the term represented by t with variables replaced according to A. To construct a new rewriting function f, one proves that if `FORMP`(t), then `FORMP`$(f(t))$ and

$$\texttt{(EQUAL (MEANING } t\texttt{) (MEANING } f(t)\texttt{))}$$

The system, if presented with this theorem, will be able to subsequently use f whenever the set of definitions existing in the prover contains the current set. Things are so arranged that the rewriting functions, when incorporated, operate directly on the implementation's representations of terms, and this permits very efficient execution.

The most important differences between our work and that of Boyer and Moore are related to the fact that in the former the entire mechanism is developed within the logic. The representation of contexts (environments) provides a crucial means for generalization. In Boyer and Moore's system, one cannot, for example, prove the correctness of metafunctions that implement congruence closure, or a simplification procedure for all rings, since proofs must always take place in the context of a fixed set of associations between identifiers and defined functions. Another advantage to our approach is that the set of methods for applying a "meta-fact" is not fixed. In the metafunction approach, the mechanism for applying meta-theoretic results at the object level is built into the system. Metafunctions must be total functions (*i.e.*, applicable to any s-expression) that preserve meaning whenever given a well-formed argument. With our approach there is no such restriction. For example, one can construct procedures that have an arbitrary predicate as a precondition to their application. As another example, one could define *conditional* rewrites whose applications generate further proof obligations. These obligations could be in turn be dealt with by further rewriting, or could be collected and presented to the user as subgoals.

There are two other distinctions worth pointing out. The types $\texttt{Term}(\alpha)$ can represent a richer class of objects than those that are definable in Boyer and Moore's system. For example, partial functions (those whose domains are subtypes) can be represented, as can undecidable predicates such as the equality of real numbers. Secondly, the metafunction approach shares with VT and FOL the lack of higher-order features.

Formalized Metareasoning in Type Theory

Concurrent with our work has been an independent line of research conducted by Constable and Knoblock [41, 40]. Their goal has been to unify the object and the meta-languages of Nuprl by replacing ML with Nuprl's type theory. Their paper [41] presents two different approaches to this problem.

The first is to develop the needed mechanisms entirely within Nuprl. This involves defining a type to represent Nuprl proofs, proving the existence of a function that maps a complete (as a proof) member p of this type to the proposition which is represented by the main goal of p, and proving that such a proposition is always true. Unfortunately, they have been unable to carry out a proof of the main claim stated in their paper. The main source of the difficulties is that the method they are using requires formal analysis of Nuprl's highly complicated proof structure. This approach appears to have been abandoned.

The second is to have, as in FOL, a separate meta-theory. This approach is the subject of Knoblock's thesis [40]. He defines a hierarchy of formal systems PRL_i $(0 \leq i)$. PRL_0 is just the usual Nuprl type theory. Each PRL_{i+1} is the base logic augmented by types which represent the syntax and proof theory of PRL_i. Much of the work of Knoblock's thesis has been to show how this hierarchy can fill the role currently played by ML. In particular, he develops the tactic mechanism in this setting. As an example, consider the case of *complete* tactics (those which either fail, or completely prove a goal). Let $proof_0$ denote the type in PRL_1 that represents the proofs of PRL_0. To write a complete tactic for constructing PRL_0 proofs, one writes a Nuprl function that operates on members of $proof_0$, and proves in PRL_1 that it is correct whenever a specified precondition (the "admissibility predicate") is satisfied. To prove a goal in a PRL_0 proof, one can invoke this function as a refinement tactic; the system will simply produce a single subgoal to prove that the admissibility predicate holds of the representative in $proof_0$ of the goal. The important point here is that the Nuprl function does not need to be executed (unless extraction demands it). If the admissibility predicate has an appropriate form, it can simply be evaluated to *true* or *false*.

In contrast to our approach, Knoblock's hierarchical system rests on a formalization of the proof system. New inference procedures for PRL_0 that are implemented as functions in PRL_1 must build representations of PRL_0 proofs to justify the new inference, although in many applications the functions will not have to be executed. It may be possible, however, to design abstraction mechanisms that lessen the burden on the tactic designer of having to be concerned with proof construction. A more serious difficulty is that tactics cannot be "lifted" in the hierarchy. For example, a tactic which operates on members of $proof_0$ cannot in general be applied to members of $proof_1$. One way to fix this is to incorporate the desired property into the definition of the type of tactics, much as we have done in building environment extension into our definition of the type of rewriting functions.

Our partial reflection mechanism avoids the complications related to having a hierarchy of metatheories, although at the price of a loss of generality. A more important distinction between our work and Knoblock's, though, is that ours provides structure that supports direct construction of useful procedures. Consider, for example, the term-rewriting system described earlier. Its design makes crucial use of the structure of lifted sequents. If a sequent has been lifted with respect to some environment α, and if t is one of its constituent meta-terms, then t is well-formed with respect to α. From the definition of well-formedness, it follows that any subterm u of t can be replaced by any term u' that has an equal value. This is because the value given by α of any meta-function occurring in t must be functional with respect to the equalities associated with its argument types. Hence the design of the rewriting system depends on certain properties of meta-terms (*i.e.*, those properties that follow from well-formedness) remaining true. The $proof_0$ representation of PRL_0 sequents to not entail such properties. It seems likely that in order to build within the hierarchy a rewriting system comparable to ours, one would have to construct analogues of our all of our basic mechanisms.

1.4 Notes to the Reader

Chapter 2 gives some details about the Nuprl system and should be read before the remaining chapters. Chapters 3, 4 and 5 deal with the basic tactic collection, Girard's paradox and the partial-reflection library respectively. Several discussions in Chapters 4 and 5 about Nuprl proofs assume an acquaintance with the material of Chapter 3. The libraries described in Chapters 4 and 5 involve a style of definition that is discussed in the first section of Chapter 3. Except for these dependencies, Chapters 3, 4 and 5 may be read independently.

The second half of the thesis contains appendices. By far the largest of these is Appendix C, which contains a complete listing of the library described in Chapter 5. Appendices A and B supplement Chapters 4 and 2 respectively.

Throughout the thesis we use typewriter typeface exclusively for ML expressions and for terms of the Nuprl type theory (the latter may contain definition instances). When we give a term of the type theory, what is shown is similar to what might be seen on the screen during a Nuprl session, with the following exceptions.

- We have taken some liberties with parenthesization and whitespace.
- We have manually corrected for a quirk of the system where the display form associated with a definition is occasionally lost during proof development.
- We will often use metavariables in terms of the type theory. These will have italic typeface. For example, `x+y` is a term containing Nuprl variables `x` and `y`, whereas we might say "if x and y are terms then so is x+y".

Chapter 2

Nuprl

This chapter gives some details about the Nuprl system and type theory. For a more complete account, see the Nuprl book [23]. The version of Nuprl used for most of the work presented in this thesis is slightly different than the one the book describes. The more important differences will be described in the last section of this chapter.

2.1 The System

Typically, a user interacts with Nuprl by creating, deleting, modifying and viewing objects. Objects are named, and are stored in the *library.* The objects in the library are linearly ordered, and an object can depend only on objects that precede it. There is a special window for viewing the current contents of the library. Libraries can be dumped to or loaded from a file.

Nuprl has two subsystems for evaluating programs. The programs can be written in ML and run by the ML subsystem, or they can be written in the programming language that is contained in Nuprl's type theory. The latter programs are handled by a special purpose call-by-need evaluator. ML will not be discussed here. The ML used in Nuprl is very close to what is described in the LCF manual [30].

A library object is either a definition, an "eval" object, an ML object, or a theorem. All but the last contain text, and can be edited by a special-purpose window-oriented text-editor. Objects are processed by the system when the user commands that they be *checked.* In the case of eval or ML objects, the contents are evaluated as Nuprl or ML programs, respectively. Such evaluations usually result in new associations between names and values.

A definition object, when checked, creates a new notation. The body of a definition has the form

$$\textit{text} == \textit{text'}$$

where *text* and *text'* are pieces of text that give the left and right hand sides of a definition of a template. Identifiers enclosed in angle brackets (< and >) are taken to be the formal parameters of the template. (Actually, the formal parameters on the left may contain an addition documentation string that is used when the definition is displayed in the library window). A definition may be invoked by name in any text-editor window. What the user sees is just the left hand side, and he can use the text editor to substitute text for the formal parameters. When text containing definitions is processed, such as when it is parsed as a term, the right hand sides, with the appropriate replacements for formal parameters, are substituted. The structure of a term incorporates the text from which it was derived; this allows the notations given by definitions to be retained (but not always) when terms are displayed.

The description of theorems and proofs that was given in Chapter 1 is adequate for the needs of this thesis.

2.2 The Type Theory

This section contains an informal semantics of the Nuprl type theory. For a precise account of the part of the theory that does not involve recursive types, see Allen's thesis [3]. For an extension of the semantics to include recursive types, see Mendler's thesis [48]. Our account is based on one that appears in the Nuprl book [23], and proceeds in four stages. First is a definition of the set of terms of the type theory. Second, an evaluation relation on the terms is defined. Next, we discuss what it means for a term to be a member of a type, and for two terms to be equal members of a type. Finally, we discuss the meaning of sequents and make a few points about the rules. It is assumed that the reader is familiar with common syntactic notions concerning variable binding and substitution. Denote by $t[a/x]$ the result of substituting a for the free occurrences of x in t.

Terms

Every closed term in the Nuprl theory is either *canonical* or *noncanonical.* The canonical terms are the terms which can be the results of evaluation. Following is an inductive definition of the set of terms of the Nuprl type theory. With each term constructor introduced, we will indicate whether its closed instances are canonical or noncanonical. Variables, denoted by identifiers that are strings of letters, are terms. Certain identifiers are reserved for operator names and cannot be used as variables. Each integer is a canonical term. A double-quote ("""), followed by string of characters not containing double-quotes, followed by a double-quote, together form a canonical term called an *atom.* Below we present the rest of the terms. In this presentation, let x, y, u, and v range over variables, and a, b, s, t, A, and B range over terms. In any subterm of the form $x_1, \ldots, x_n . b$, the variables x_i become bound in b. In the noncanonical terms below, certain subterms will be called *principal arguments*; these subterms will be indicated by placing them in square brackets after the term.

Redex	**Contractum**
(\x.b)(a)	b[a/x]
spread(<a,b>;x,y.t)	t[a, b/x, y]
decide(inl(a);x.s;y.t)	s[a/x]
decide(inr(b);x.s;y.t)	t[b/y]
list_ind(nil;s;x,y,u.t)	s
list_ind(a.b;s;x,y,u.t)	t[a, b, list_ind(b;s;x,y,u.t)/x, y, u]
atom_eq(i;j;s;t)	s *if* $i = j$; t *otherwise*
int_eq(m;n;s;t)	s *if* $m = n$; t *otherwise*
less(m;n;s;t)	s *if* $m < n$; t *otherwise*
m+n	$m + n$
ind(m;x,y.s;b;u,v.t)	b *if* $m = 0$; t[m,ind($m - 1$;x,y.s;b;u,v.t)/u, v] *if* $m > 0$; s[m,ind($m + 1$;x,y.s;b;u,v.t)/x, y] *if* $m < 0$.
rec_ind(a;h,z.d)	d[a,\z.rec_ind(z;h,z.d)/h, z]

Figure 2.1: Redices and Contracta

Canonical terms: void; int; atom; axiom; nil; U*k* for all natural numbers *k*; inl(*a*); inr(*b*); *A* list; \x.b; a<b; <a,b>; a.b; x:A#B, where *x* becomes bound in *B*; x:A->B, where *x* becomes bound in *B*; A|B; {x:A|B}, where *x* becomes bound in *B*; rec(x.A), where *x* becomes bound in *A*; and a=b in *B*.

Noncanonical terms: -t [t]; any(a); t(a) [t]; a+b [a,b]; a-b [a,b]; a*b [a,b]; a/b [a,b]; a mod b [a,b]; spread(a;x,y.t) [a]; decide(a;x.s;y.t) [a]; list_ind(a;-s;x,y,u.t) [a]; ind(a;x,y.s;b;u,v.t) [a]; rec_ind(a;h,z.d); atom_eq(a;b;-s;t) [a,b]; int_eq(a;b;s;t) [a,b]; and less(a;b;s;t) [a,b].

If the variable *x* does not occur in *B*, then x:A->B can be written a A->B, and x:A#B as A#B. Nuprl also has a quotient-type constructor, but it is not used in any of the work described in this thesis. The term constructors rec_ind and rec are not part of the type theory given in the Nuprl book; see Appendix B for more on them.

Evaluation

Evaluation is inductively defined in terms of the redex-contractum relationship given in Figure 2.1. In the figure, in addition to the conventions above, *m* and *n* range over integers, and *i* and *j* range over atoms. Also, the redex-contractum pairs for the integer operations other than + are analogous to the pair for +, and have been omitted.

A term *s* *evaluates* to a term *t* if *t* is canonical, and either *s* is identical to *t*, or *s* is a noncanonical term such that (1) its principal arguments evaluate to some terms, (2) the result of replacing the principal arguments with their values is a redex, and

(3) the corresponding contractum evaluates to t.

Membership

Some terms denote *types*; with each type is associated a set of terms called the *members* of the type and an equality relation on the members. A term is a member of a type exactly when it evaluates to a member of the type, so to give the members of a type it suffices to give the canonical members. We will not deal with the issue of when a term denotes a type, or when two types are equal, except to note that equal types have the same members.

What follows is an informal description of types and their canonical members. The description is a slight modification of one that appears in the Nuprl book.

The integers are the canonical members of the type `int`. The atoms are the canonical members of the type `atom`. The type `void` is empty. The type $A|B$ is a disjoint union of types A and B. The term `inl(`a`)` (`inr(`b`)`) is a canonical member of $A|B$ whenever $a \in A$ ($b \in B$). The operator names `inl` and `inr` are mnemonic for "inject left" and "inject right". The canonical members of a type $x:A\#B$ ("dependent product") are the terms $<a,b>$ such that $a \in A$ and $b \in B[a/x]$. Note that the type from which the second component is selected may depend on the first component. The canonical members of the type A `list` represent lists of members of A. The empty list is represented by `nil`, while a nonempty list with head a and tail b is represented by $a.b$, where b is a member of the type A `list`.

A type `rec(`$T.A$`)` is a *recursive* type. Its members are the members of

$$A[\texttt{rec}(T.A)/T].$$

The recursive type constructor used in our work is a simplified version of what is described in the Nuprl book. It is also slightly different from Mendler's version [48].

A term of the form $t(a)$ is an *application* of t to a, and a is its *argument.* A canonical member of a type $x:A\text{->}B$ is a *function*, and is a lambda term $\backslash x.b$ whose application to any member a of A is a member of $B[a/x]$. It is required that applications of a function to equal members of A be equal in the appropriate type.

The significance of some constructors derives from the representation of propositions as types, where the proposition represented by a type is true if and only if the type is inhabited. If a and b are members of `int` then the term a<b is a type which is inhabited if and only if a is less than b. If a and b are members of A then the term a=b `in` A is a type which is inhabited if and only if $a = b \in A$. The term a=a `in` A is also written a `in` A; this term is a type and inhabited if and only if $a \in A$. If a=b `in` A or a<b is inhabited, then it has the unique canonical member `axiom`.

A type of the form $\{x:A|B\}$ is a *set type.* Its members are the members a of A such that $B[a/x]$ is inhabited. For example, `{x:int|0<x}` has just the positive integers as canonical members. An important use of the set type is as a *squash* operator. We will denote by $\downarrow(A)$ the type `{0=0 in int|`A`}`; this type is the

"squash" of A since it has a unique member `axiom` if A is inhabited, and is empty otherwise.

We now give the equalities for the types already discussed. Members of `int` are equal if and only if they have the same value. The same goes for `atom`. Canonical members of $A \mid B$, $x{:}A\#B$ and A `list` are equal if and only if their corresponding immediate subterms are equal (in the corresponding types). Members of $x{:}A\text{->}B$ are equal if and only if their applications to any member a of A are equal in $B[a/x]$. The types a<b and (a=b in A) have at most one canonical member, so equality is trivial. Equality in $\{x{:}A \mid B\}$ is just the restriction of equality in A.

The type Uk, for k positive, is called a *universe.* The members of Uk are types. The universes are cumulative; if j is less than k then membership and equality in Uj are just restrictions of membership and equality in Uk. Uk is closed under all the type–forming operations except formation of Ui for i greater than or equal to k. Equality in Uk is the restriction of type equality (which has not been discussed) to members of Uk.

To see how logic is defined in Nuprl using the propositions-as-types correspondence, see Appendix C.

Sequents and Rules

The rules of Nuprl deal with *sequents.* A sequent has the form

$$x_1{:}A_1\texttt{,}\ \ldots\texttt{,}\ x_n{:}A_n\ \texttt{>>}\ B\quad \texttt{[ext}\ b\texttt{]}\,.$$

Ignoring the issue of type equality, such a sequent is said to be true if B is a type and b is a member of B whenever $x_1, \ldots, x_n$ are terms such that for each i, A_i is a type and x_i is a member of A_i. The term b is called the *extraction* of the sequent, and it is not actually a component of the system's representation of sequents; in particular, the system does not display the "ext" field when is displays a sequent. Instead, because of the design of the rules, the system can take any complete (and sometimes an incomplete) proof and compute the extraction fields. Only the extraction from the top sequent of a proof-tree can be accessed. The pseudo-term `term_of(`i`)`, for i an identifier, stands for the term extracted from the theorem named i.

There are a large number of rules in the Nuprl proof system. Some of them are for general reasoning about equality. Most of the rules fall into one of three categories. For each type, there are elimination ("elim"), introduction ("intro"), and equality-introduction ("equality-intro") rules. An elim rule analyzes the type when it appears in a hypothesis of a sequent, usually producing subgoals that have new hypotheses. An intro rule analyzes the type when it appears as the conclusion of a sequent. An equality-intro rule applies to sequents whose conclusion is an equality term.

Two rules of special importance are the *direct computation* rules. These rules allow sequences of redex contractions to be applied to part of a sequent. These sequences are specified with *tagged terms.* For example, to specify some contractions to be done within the conclusion of a sequent, one annotates a copy of the conclusion with "tags" that indicate the number of computation steps to be done to the tagged subterm.

2.3 Changes to Nuprl

We have made some minor changes to Nuprl that are worth noting. First, we have extended the direct computation facility. The new direct computation rules are essentially the reverse of the old ones: a term t can be replaced by a term t' if t can be obtained from t' by a sequence of contractions. The new rules are justified the same way as the old. See [3] for details.

The "eval" rules are more efficient versions of the direct computation rules. Direct computation uses a very slow symbolic computation algorithm that contracts redices by performing substitutions. The eval rules use Nuprl's evaluator. Evaluation is controlled somewhat differently than direct computation. Direct computation allows fine control over the reductions that are done; with evaluation, all one can specify is which `term_of` terms are to be treated as constants (*i.e.*, not expanded into the terms they stand for) during evaluation.

See appendix B for more on the additions that have been made to the proof system since the Nuprl book [23] was written. There have also been quite a few minor changes to other parts of the system. These are all documented in [37].

Chapter 3

Basic Tactics

This chapter contains some details on the collection of basic tactics that was outlined in Chapter 1. Most of the tactics in the collection will be at least mentioned. The ones that are omitted are mostly either simple variants of objects that are mentioned, or have turned out not to be very useful in practice. The last part of the chapter contains descriptions of two applications of the collections.

Most of this chapter has the form of a user's manual, although many of the descriptions will omit details that are uninteresting but necessary for actual use. A typical description will be headed by a schematic application that may contain type information. For example, a description of the function `destruct_hyp` might be introduced by

`destruct_hyp (`*i*`:int) (`*p*`:proof): tok#term.`

This indicates that the function has type

`int -> proof -> tok#term.`

The parameters i and p may be referred to in the description.

Many of the tactics described perform some kind of rewriting on a component of a sequent. There are usually several ways to apply a particular kind of rewriting; for example, one can apply it just to the conclusion, or to some hypotheses, or to all hypotheses and the conclusion. Usually, only one of these will be mentioned (usually the one which applies rewriting everywhere). For example, only `Eval` will be described, although there are also tactics `EvalHyp` and `EvalConcl`.

3.1 Definitions

To take full advantage of the tactic collection, the user must follow a simple convention in making definitions. As pointed out in the previous chapter, although Nuprl's definition facility allows considerable flexibility in making notations for terms, it cannot reasonably serve as an abstraction mechanism. A solution to this deficiency is to define objects via extraction. Using this scheme, a complete definition consists of two library objects: a theorem object whose extraction is the defined term, and a definition object which gives a display form for the `term_of` term which denotes the extraction. Sometimes the defining theorem also serves to give a type to the defined term; other times, a separate theorem is used for this. The library objects associated with a definition are related by a naming convention (involving trailing underscores).

There are two styles of definition in this scheme. We will describe these using two examples that involve slightly modified (for simplicity) versions of objects appearing in the library of Appendix C. The first style of definition is exemplified by the definition of a function which takes the maximum of two natural numbers. The definition object for this function is called `N_max`, and its body is

```
max(<m>,<n>) == term_of(N_max_)(<m>)(<n>).
```

The theorem `N_max_` has statement

```
>> N -> N -> N,
```

and is proved by explicitly introducing the desired object, using the tactic invocation

```
(ExplicitI 'λm n. less(m;n;n;m)' ...).
```

The second style of definition is exemplified by the definition of the length function (for lists). The definition object `length` is

```
|<l>| == term_of(length_)(<l>).
```

The theorem `length_` has statement

```
>> Object
```

and extraction

```
λl. list_ind(l; 0; h,t,v.v+1).
```

The type of the theorem extracted from is the trivial type `Object` because it is convenient to have the polymorphism of `length` implicit. (The type `Object` never appears except in these kinds of definitions. See Appendix B for more information on `Object`). In other words, instead of taking as arguments a type and a list over that type, the length function as defined above just takes a list as an argument. The typing lemma for this definition is called `length__` and is

```
∀A:U1. ∀l:A list. |l| in N
```

In summary, an instance of a definition is actually an application of a `term_of` term to some arguments. Definition objects in the library should be regarded as incidental additions that make displayed terms easier to read.

3.2 Terms

There is a large set of functions for dealing with terms. Most of these functions are for analyzing, constructing, or recognizing terms. Only three of them will be described here.

`get_type` (p:`proof`) (t:`term`). The result is a type T for t, such that the autotactic can (almost always, in practice) prove that t is a member of T under the hypotheses of the sequent p. The algorithm that computes T uses a function g that computes a type given a term t and a typing environment e (an association between Nuprl variables and types). This function uses a simple recursive algorithm. If t is a variable, the result is obtained by looking up t in e. If t is a canonical term, the result is computed from types for the immediate subterms. This is not alway possible (in which case an exception is raised), as when t is a lambda-abstraction. See the discussion of the membership tactic below for an explanation of why this kind of failure tends not to be a problem in practice. When t is a non-canonical term, the first step is to compute a type T for its principle argument. Head normalization and stripping off of set types is then repeatedly applied to T until a type T' is obtained that is canonical and not a set type. If t is an application $f(a)$, then T' must be a function type $x:A$`->`B (else failure), and the result is $B[a/x]$. If t is of the form `decide(`a`;`u`.`b`,`v`.`c`)`, then T' must be a disjoint union $A|B$, and the result is g applied to b and the environment e updated with an association between u and A; or if this fails, an analogous application to c. The `spread` case is analogous. If t is an induction form, then an attempt is made to compute the result just from the base case. Finally, if t is of the form `term_of(`a`)`, then the result is the main goal of the theorem named a.

A special case is when t is an instance of a definition of the second kind described in Section 3.1. In such a case, there is a theorem that is a universally quantified statement ending with a term of the form a `in` A. Matching of a against t is used to compute terms for instantiating the quantified variables appearing in A, and the resulting instance of A is returned as the type of t.

`match_part_in_context` f A t p l. Find a subformula A' of A, and a list of terms that yield t when substituted in A' for the variables that are bound by the universal quantifiers in A whose scope contains A'. The result is the list of terms. The subformula A' is one which can be obtained by descending through conjunctions, through universal quantifiers, through implications via the consequent, and finally by applying the function f, which should be a term destructor. First-order matching is used to compute substitutions for the free variable occurrences of A' that are bound in A. If matching does not provide instantiating terms for all the universal quantifiers encountered on the descent to A', then type information is used. In particular, for each universally quantified variable x which has been instantiated with a term a, a type is computed for a using `get_type` p a, and this type (or certain derived types, if necessary) is matched against the type obtained from the appropriate universal quantifier. After all possible type information has been used, the term list l is used for any further required instantiating terms.

The use of type information is important for definitions that use implicit polymorphism. As a simple example, consider the function `tl` that computes the tail of a list. Suppose that it has the typing lemma

```
∀A:U1. ∀l:A list. tl(l) in  A list.
```

In computing the type of a particular application `tl(`s`)`, the matching of `tl(`s`)` against `tl(l)` gives an instantiation only for `l`. If a type for s can be computed, and this type is (or can be reduced to) a list type, then matching against the declared type `A list` for `l` gives an instantiation for `A`. Instantiating the lemma with the computed terms yields a type for `tl(`s`)`.

`tag_redices (`t`:term)`. This tags redex-contractum pairs in t so that direct computation will result in all the contractions being performed. This function can be updated from ML objects in the library (by adding to a global list). This allows the user to make some of the tactics that use reduction take into account definitional extensions to the redex-contractum relation.

3.3 Computation

There are many tactics which are based on the evaluation and direct computation rules discussed in Chapter 2. These include the tactics for definition expansion.

`Eval`. Perform evaluation on the conclusion and all hypotheses, expanding `term_of` terms whenever required for computation to proceed.

`EvalOnly` *names*. Like `Eval`, except that terms of the form `term_of(`x`)` are only expanded if x is a member of the list *names*.

`EvalExcept` *names*. Like `Eval`, except that terms of the form `term_of(`x`)` are treated as constants when x is a member of the list *names*.

`EvalSubtermOfConcl` b *names* P. Evaluate a subterm of the conclusion that satisfies the predicate P. The arguments b and *names* have the same purpose as in the evaluation rule (see Appendix B).

`CC`. Compute the conclusion to head normal form.

`ComputeEquands`. If the conclusion is of the form a`=`b `in` A then head-normalize the "equands" a and b, else fail.

`Reduce`. Repeatedly perform reductions of redex-contractum pairs wherever the legal-tagging restrictions permit it.

`HypModComp(`i`:int)`. It can require excessive computation time to demonstrate that a hypothesis is equal (as a type) to the conclusion of a sequent by completely normalizing both terms and then checking that the results are identical. `HypModComp` provides an approximation to complete normalization that is usually much faster. It proceeds by applying reductions to a subterm in the hypothesis and a corresponding subterm in the conclusion. Initially, the subterms are the entire respective terms. At each stage, it applies the smallest number of reductions such that the two resulting subterms are either instances of the same definition, or are in head normal form with the same head. If this is impossible, the tactic fails, otherwise the procedure is repeated on the immediate subterms of the resulting subterms. An important application of this tactic is in the inclusion tactic described below.

There are several tactics that are used for *unfolding* (expanding) and *folding* (the inverse of unfolding) definitions. Because definitions are applications of `term_of` terms to some arguments, unfolding a definition occurrence in a sequent requires using direct computation to replace the appropriate `term_of` term by its denotation and to do reduction steps until all the arguments to the definition instance are substituted into the body of the definition.

`Unfold` *names.* Unfold all definitions whose name is in the list *names.*

`Fold` *names.* Add as many instances of the named definitions as possible. This is not alway easy, since arguments to a definition instance can disappear when the definition is expanded. This is often the case for arguments that are types. For example, the length function given in Section 3.1 could have been defined to be

```
λA. λl. list_ind(l; 0; h,t,v.v+1),
```

which has type

```
A:U1 -> l:(A list) -> N.
```

To fold a term

list_ind(l; 0; h,t,v.v+1)

into an application of the length function, the type of l has to be computed.

`UnrollDefs` *names.* For each *name* in *names* and each instance of the definition named *name*, do the following. Unfold the definition, then fail if the result is not a redex that is of the form `list_ind(...)` or `rec_ind(...)`, otherwise contract the redex and attempt to fold it to an instance of the definition *name.* For example, unrolling `|h.t|` gives `1+|t|`.

3.4 Tacticals

A *tactical* in LCF is a function which is used for combining tactics. The basic LCF tacticals have analogues in Nuprl.

T `THEN` T'. Apply the tactic T. If it fails, then fail, otherwise apply T' to the subgoals generated by T. This tactical has several variants which incorporate useful restrictions on the kind of subgoal T' is applied to. T `THENS` T', when applied to a proof p, only applies T' to subgoals that have the same conclusion as p. `THENO` applies its second argument only to goals that have a *different* conclusion. `THENM` and `THENW` apply their second argument only to goals that are not membership goals or well-formedness goals respectively. (A membership goal is one whose conclusion is of the form `>>` t `in` T, and a well-formedness goal is one whose conclusion is of the form `>>` T `in` Ui.) Because many tactics have actions that are well-defined except for the generation of "junk" subgoals, the variants are used extensively for combining these tactics (both at the top level in proof construction, and in the construction of other tactics).

`Repeat` T. Apply T until it fails or ceases to make progress. This tactical has variants analogous to the variants of `THEN`.

T `ORELSE` T'. Apply T. If it fails, apply T'.

3.5 Some Derived Rules

Many of the tactics in this section generate unwanted subgoals. These are usually membership goals, or can be proven by some trivial propositional reasoning. In practice they can almost always be proved by the autotactic, and so the tactics will be described as though they did not generate any "junk".

For each primitive inference rule in Nuprl, there is a corresponding tactic. Such tactics are useful partly because they compute many of the parameters required by the corresponding rule. Some of these parameters are easily computed, such as those giving names for new variables, and many are not, such as those which supply types for certain subterms of the goal. Many of these tactics also incorporate computation. For rules which require that a hypothesis or the conclusion be a canonical type, the corresponding tactic will head-normalize the appropriate term before attempting to apply the rule.

`ILeft`. If the conclusion is or computes to a disjoint union, then apply the introduction rule which picks the left disjunct. `IRight` is analogous.

`ITerms` l. If the conclusion is existentially quantified (using product types or set types), then introduce the terms in the list l as witnesses for the existentially quantified variables.

`I`. If the previous three tactics do not apply, and if the conclusion is a canonical type that is not an equality or `void`, then apply the introduction rule for the type.

`EqI`. If the conclusion is of the form t `in` T or t=t' `in` T, where t is not a variable and, in the latter case, t and t' have the same outermost term constructor, then apply the appropriate equality-intro rule. This will often involve computing a type. For example, when applied to the goal

$$\texttt{>>}\ f(a)\ \texttt{in}\ T,$$

`EqI` must compute a functional type for f in order to apply the equality-intro rule for application. If the type computed for f is $x:A$->B, but $B[a/x]$ is not identical to T, then no equality-intro rule applies. In this case, `EqI` will assert

$$f(a)\ \texttt{in}\ B[a/x].$$

To prove that this implies the original goal, `EqI` calls `Inclusion`, which is described in Section 3.9.

`EOn` t i. Instantiate with t the universally quantified formula in hypothesis i.

`E` i. If the type in hypothesis i is not universally quantified, apply the appropriate elimination rule.

`Unroll` i. If hypothesis i declares a variable x to be of a list or recursive type, then "unroll" the type. If the type is a recursive type, the main step is to apply the "unroll" rule for recursive types. If it is a list type, then there are two main subgoals. The first is essentially the goal with x replaced by `nil`, and the second is the goal with x replaced by the cons h`.`t for h and t new variables.

`SubstFor` t. If t is of the form a`=`b `in` A, substitute b for a in the conclusion. There is an analogous tactic, `SubstForInHyp`, which works on hypotheses.

There are also slightly generalized versions of some of the elimination rules. For example, `RepeatAndE` breaks up a hypothesis which is a conjunction, making each conjunct into a new hypothesis. The work done by some of these generalizations can be somewhat substantial. As an example, consider the elimination rule for dependent product types. If a hypothesis declaring some variable x to be in a type of the form y`:`A`#`B is eliminated, then the hypothesis list is extended by three new hypotheses:

$$y\texttt{:}A\texttt{,}\ \ z\texttt{:}B\texttt{,}\ \ x\texttt{=<}y\texttt{,}z\texttt{>}\ \texttt{in}\ y\texttt{:}A\texttt{\#}B\texttt{,}$$

and in the conclusion `<`y`,`z`>` is substituted for x. The tactic `GenProductE` generalizes this rule to an arbitrary number of repeated products. For an n-fold product, $n+1$ new hypotheses are generated, the last of which is an equation between x and an n-tuple. Simply chaining together an appropriate sequence of applications of the elimination rule will generate n equalities, which must be collapsed using substitution. The tactic uses another method which is slightly better. In either case, new membership subgoals are generated along the way, and so applications of the tactic can be much slower than one might expect.

A useful way to apply elimination rules is with `OnVar`.

`OnVar` x `(`T`:int->tactic)`. If x is declared in the hypothesis list, apply T to that hypothesis. Otherwise, repeatedly apply introduction rules as long as the conclusion is a universal quantification or an implication, or until x becomes declared, in which case apply T to the hypothesis declaring it.

There are several tactics that perform generalization, replacing a term in the conclusion by a new variable.

`LetBe` x a A. If the goal is `>>` B, then the subgoal generated is

$$x\texttt{:}A\texttt{,}\ \ x\texttt{=}a\ \texttt{in}\ A\ \texttt{>>}\ B[x/a]$$

(regarding a subgoal `>>` a `in` A as a "junk" subgoal).

`TypeSubterm` a A. If the goal is `>>` b `in` B, then the subgoal generated is

$$x\texttt{:}A\ \texttt{>>}\ b[x/a]\ \texttt{in}\ B\texttt{.}$$

`ETerm` a. This is similar to `LetBe`, except that the type A for a is computed, and elimination is done on the resulting new declaration.

There are some simple derived rules for equality reasoning.

`DestructEq` f i. If hypothesis i is of the form a=b in A, then compute a type A' and add a new hypothesis

$$f(a) = f(b) \texttt{ in } A'.$$

The term f must be a lambda-abstraction, and should be a destructor function (such as projection from pairs).

`SplitEq` t. If the conclusion is of the form a=b in A, then generate two subgoals, with conclusions a=t in A and t=b in A.

There are two frequently used tactics related to the set type.

`Unhide`. If the conclusion is a squashed type (see Chapter 2 for a definition of squashing), then the single subgoal is identical to the goal, except that no hypotheses are hidden (a hidden hypothesis is one that cannot be used until it becomes unhidden—hidden hypotheses are created only by the set-elim rule).

`Properties` l. For each term t in the list l, compute a type for t, head-normalize the type, and if a type of the form $\{x:A \mid B\}$ is obtained, then add $B[t/x]$ as a new hypotheses.

We end this section with a few miscellaneous tactics.

`AbstractConcl` t. If the conclusion is A, then create a new variable x and produce a subgoal with conclusion

$$(\lambda x.\ A[x/t])\ (t).$$

This is useful in the application of certain lemmas which quantify over predicates, such as those expressing induction schemes.

`BringHyps` l. "Bring" some hypotheses (specified by the numbers in l) to the conclusion side of the sequent, adding them to the old conclusion as antecedents of an implication or as universal quantifiers. As an example, bringing the last hypothesis of

$$x:A \texttt{ >> } B$$

results in

$$\texttt{>> } x:A\texttt{->}B.$$

`NonNegInd` x i. If the type in hypothesis i is (or computes to) a subrange of the integers, then use it to calculate an integer lower bound for the variable x declared by the hypothesis. If the lower bound is non-negative, then do induction on x, using the bound as the base case.

3.6 Chaining and Lemma Application

There is a group of tactics based on the well-known notions of backward and forward chaining. The description of these tactics requires two definitions. For terms A and B, A is an *assumption of* B if: B is of the form $x:C$->D (which may be degenerate, *i.e.*, an implication) and A is either identical to C or is an assumption of D; or B

is of the form $C\#D$ and A is an assumption of C or of D. A is a *conclusion of* B if: A and B are identical; or B is $x{:}C\text{->}D$ and A is a conclusion of D; or B is $C\#D$ and A is a conclusion of C or of D. As assumption A of B is an antecedent in B of a conclusion A' of B if: B is $x{:}C\text{->}D$ and either A is C and A' is in D or A is an antecedent of A' in D; or B is $C\#D$ and either A is an antecedent of A' in C or A is an antecedent of A' in D.

`BackThruHypUsing` l i. Do one backward chaining step using hypothesis i. This uses `match_part_in_context` (described earlier) to match the conclusion C of the sequent against a conclusion of the type A that is hypothesis i. The terms in l are used as additional instantiating terms. If the match is successful, then the tactic generates as subgoals sequents with the same hypothesis list, but with conclusions that are suitble instantiations of assumptions of A that are antecedents in A of the subterm occurrence matched with C.

Several tactics are based on `BackThruHypUsing`.

`BackchainWith` T. This is most concisely described by ML code:

```
let BackchainWith Tactic =
Try Hypothesis THEN
AtomizeConcl THENW
( ApplyToAHyp (λi. BackThruHyp i THENM BackchainWith Tactic)
  ORELSE Tactic
).
```

`Hypothesis` proves goals where the conclusion is identical to a hypothesis, `AtomizeConcl` repeatedly applies introductions until the conclusion is not a conjunction or function type, and `ApplyToAHyp` T, for T of type `int->tactic`, applies T to each hypothesis in turn until success (*i.e.*, until an application does not fail). A useful instance of this tactic is `BackchainWith Fail`, where `Fail` is the tactic which always fails. This implements a kind of Horn-clause theorem prover (based on depth-first search). It is used by the tactic `Contradiction`, which attempts to show the hypothesis list to be contradictory by asserting `void` and backchaining. Note that `BackchainWith` does not check for looping, although it would be easy to modify it so that it does (in fact the current version does check).

There are two main ways to apply lemmas. One is to apply them explicitly using the tactics described below. The other involves explicitly grouping them according to similarity of usage. The grouping is accomplished by using a naming conventions, as in the association of typing lemmas to definitions, or by using ML objects in the library to update global lists of tactics and lemma names. There are several tactics that refer to such lists; they generally use these lists by trying to apply every tactic in the list or every lemma named in the list. For example, the tactics `Autotactic`, `Member` and `Inclusion`, described in a later section, each (as a last resort) refer to an assooiated list. Below we give some other examples. This technique is rather *ad hoc*, but it has turned out to be very useful in practice. For a much better approach to knowledge management, see [47].

`LemmaUsing` *name* l. This is analogous to `BackThruHyp`, except that a lemma (*i.e.*, previously proved theorem), instead of a hypothesis, is "backed through". The tactic `Lemma` can be used when there is no term list l.

`FLemma` *name* l. If the hypotheses whose numbers appear in l can be matched to some of the assumptions of the theorem named by *name*, then a subgoal is generated that has the same conclusion and has as an additional hypothesis the smallest conclusion of the named theorem such that all its antecedent assumptions were matched. As in `match_part_in_context`, types of quantified variables are used to compute further instantiating terms. A typical example of the use of `FLemma` is in the application of a transitivity property. Suppose r has been proved, in a lemma named *foo*, to be transitive, *i.e.*,

$$\forall \mathtt{x,y,z}:A.\ r\mathtt{(x,y)} \Rightarrow r\mathtt{(y,z)} \Rightarrow r\mathtt{(x,z)}.$$

If the first and second hypotheses in some sequent are $r(x,y)$ and $r(y,z)$, then an application of `FLemma`, with arguments *foo* and `[1;2]`, will generate a subgoal which has $r(x,z)$ as a new hypothesis.

`RewriteConclWithLemma` *name*. If the named lemma has an equality as a conclusion, then try to match the left side of the equality against a subterm of the conclusion of the sequent; if successful, substitute for that subterm the appropriate instance of the right side of the equality. This may generate non-trivial subgoals, other than the one resulting from the substitution, since the equality asserted by the lemma may have preconditions. This tactic has several variants.

`Decidable`. This just repeatedly applies tactics from a global list. The members of the list are typically applications of `Lemma` to the name of a theorem that asserts the decidability of some proposition (*i.e.*, that has a conclusion of the form $P \vee \neg P$). A related tactic is `Decide` which, given P, produces subgoals for the cases P and $\neg P$, calling `Decidable` on the subgoal $P \vee \neg P$.

`Assume` P. This is similar to `Decide`, except that it deals with goals of the form $?P$, where $?P$ is defined to be $P\,|\,\mathtt{True}$. Types of this form are used to simulate ML-style failure in Nuprl. `Assume` P, if successful, generates two subgoals, one where P is a new assumption, and one which is identical to the goal. `Assume` looks for a previously defined function that either produces a proof of P or fails. The two subgoals correspond to the two possible outcomes.

3.7 A Type Constructor

For the purpose of this section, define a *module* to be a function of the form

$$\lambda\ x_1\ \ldots\ x_m.\ y_1:B_1\ \#\ \ldots\ \#\ y_{n-1}:B_{n-1}\ \#\ \{y_n:B_n\,|\,C\}.$$

that is a member of a type

$$x_1:A_1\ \mathtt{->}\ \ldots\ \mathtt{->}\ x_m:A_m\ \mathtt{->}\ \mathtt{U}i.$$

Many data types that arise in programming and in mathematics are modules by this definition (which is nonstandard). To help create and use such parameterized data types, there is a function `create_module`. The inputs to this function are the terms A_i, B_j, and C, a name M for the module, names for Nuprl functions for projecting components of members of the type, a library position at which to begin creating the associated library objects, a preferred name for a typical element of the module, and the universe level i.

Executing the function creates library objects for the following:

- The definition of the module M. This includes a definition object, and a theorem from which the module is extracted. The theorem is usually proved automatically.

- Definitions of n implicitly-polymorphic projection functions. For any members $a_1, \ldots, a_m$ of $A_1, \ldots, A_m$, respectively, the i^{th} projection function selects the i^{th} component from an n-tuple that is a member of $M(a_1, \ldots, a_m)$. The theorems that give the types of the projections are generated, and usually proved, automatically.

- A theorem that states that C, which can be thought of as an axiom about members of the module, is true of the projections of any members of any $M(a_1, \ldots, a_m)$.

- Updating the `tag_redex` function (described above). An ML object is created that has the effect of advising certain tactics that perform reduction to consider an application of a defined projection to an n-tuple to be a redex-contractum pair.

- Updating the generic module eliminator. An ML object is created that has the effect of updating the `ModE` tactic to take account of the new module. This tactic can be applied to any hypothesis whose type is an instance (*i.e.*, application) of a module that has been defined by `create_module`. The tactic works like `GenProductE`, and chooses new variable names that are based on the names of the projection functions.

3.8 Derived Rules via Pattern Theorems

Often one wishes to write a tactic that simply summarizes a pattern of inference with a fixed structure. In such cases, it is often easier to partially prove a "pattern theorem" that contains syntactic variables which get instantiated when the "derived rule" is applied. Nuprl does not directly support this kind of construction of derived rules, but it is possible to simulate it. The method of simulation is somewhat inelegant; it is presented here because it is useful, and because it was used in the formalization of Girard's Paradox (see Chapter 4).

In stating these pattern theorems, atoms are used as placeholders for irrelevant subterms. For example, in the reflection library (discussed in Chapter 5), a pattern theorem is proved for induction over a recursive type `Term0`. The statement of this theorem is

`>> ∀s:Term0. "G".`

Immediately preceding a pattern theorem in the library should be a corresponding ML object. This ML object, when checked, should use the function `set_d_tactic_args` to set certain global variables to contain the arguments that are necessary for the proof of the pattern theorem. During the proof of the pattern theorem, a tactic needing arguments that cannot be computed from the sequent it is applied to uses a special function to access the global variables. For example, a tactic may require terms to instantiate a universal quantifier with; one of the global variables refers to a list of terms from which the tactic may remove the needed number of terms by using the function `get_term_arg`.

The values given to the global variables by the ML object are just placeholders for the actual arguments that will be supplied by the tactic `Pattern` which uses the theorem as a pattern for constructing another proof. `Pattern`, given as input a set of actual arguments, assigns the arguments to the appropriate variables, and then applies, in depth-first order, the rules that occur in the pattern proof tree.

Many of the derived rules that are defined with this mechanism resemble elimination or introduction rules; that is, they analyze either a hypothesis or the conclusion. It is useful to encapsulate such tactics in a pair of tactics `DE` and `DI` (also called `DElim` and `DIntro`). These tactics try to apply members of an associated global list, continuing through the list until success. The user can have these lists updated from the library.

3.9 Type Checking and the Autotactic

As mentioned earlier, the most important single tactic is the autotactic. In this section, the autotactic and related tactics are discussed.

`Inclusion` i. Attempt to prove that a term t is in a type T' assuming that hypothesis i asserts that t is a member of a type T. This tactic will make progress in the following situations.

- One of the members of a user-defined list of tactics succeeds.
- T and T' are (or compute to) universes, with the level of T not greater than that of T'.
- T is a subtype of T'.
- T and T' "almost normalize" to identical terms. `HypModComp`, described in Section 3.3, is used here.

- T and T' are function types $x\!:\!A$`->`B and $x\!:\!A$`->`B', respectively, and `Inclusion` recursively makes progress on the goal

$$x\!:\!A,\ y\!:\!B \ \texttt{>>}\ y\ \texttt{in}\ B'$$

The last case could be generalized to other type constructors. `Inclusion` is called frequently, mostly by `EqI` (see Section 3.5).

`PolyMember`. Deal with membership goals t `in` T where t is an instance of a definition that is of the second kind mentioned in Section 3.1. The tactic uses matching to compute terms with which to instantiate the typing lemma for the definition. This results in a new hypothesis t `in` T'. If T and T' are not identical, `Inclusion` is applied. The subgoals usually produced by `PolyMember` are to show that the arguments of the definition instance are of the appropriate type. `PolyMember` can also be applied when the conclusion is a binary equality with both sides of the equality being instances of the same definition. In this case, the subgoals will be to prove that the corresponding arguments are equal in the appropriate types.

`MemberI`. Make "one step" of progress on a membership goal t `in` T. What a "step" is depends on the outermost forms of t and T. Generally, the steps are conservative, and if the step that is determined to be applicable fails, no other attempts are made. For some cases, what step to take is obvious. For example, if t is $\texttt{<}a\texttt{,}b\texttt{>}$, and T is $A\texttt{\#}B$, then the step is to apply `EqI` (*i.e.*, to apply the appropriate Nuprl equality-intro rule). In other cases, there are several reasonable alternatives. For example, if the goal is

$$\texttt{>>}\ f\texttt{(}a\texttt{)}\ \texttt{in}\ \{x\!:\!A\,|\,B\}\texttt{,}$$

then `EqI` might succeed. But one can also proceed by analyzing the set-type, getting subgoals

$$\texttt{>>}\ f\texttt{(}a\texttt{)}\ \texttt{in}\ A \qquad \text{and} \qquad \texttt{>>}\ B\texttt{[}f\texttt{(}a\texttt{)/}x\texttt{]}\,.$$

`MemberI` takes the second alternative (assuming `PolyMember` does not apply). However, if T had not been explicitly a set type, but a term that *computed*, via at least one definition expansion, to a set type, then the first alternative would have been taken. This heuristic has worked well in practice. There are a few other simple heuristics used by `MemberI`. `MemberI` is another one of the tactics that can be updated by the user; the user additions are tried first. Also, as with `PolyMember`, `MemberI` can be applied to binary equalities when the outermost forms of both sides are the same.

`Member`. Reduce the conclusion (without expanding definitions), then repeatedly apply `MemberI` to membership goals where the member is not an induction form. The reason for stopping at induction forms is that it is difficult to guess the types that are necessary to proceed. Proceeding with bad guesses can often lead to false subgoals. The conservative nature of `Member` gives typechecking in Nuprl an interactive character; most of typechecking can be handled automatically, but occasionally the tactic will stop and require the user to make a decision. Fortunately, in practice `Member` is able to automatically prove almost all membership goals that most users would consider trivial.

`Autotactic`. The autotactic cycles through the following, stopping when no further progress can be made.

- Decompose hypotheses that are conjunctions.
- Remove all top level applications of the squash operator in the hypothesis list.
- Apply `Member`.
- Analyze set types in order to expose information for `arith` and then apply it. This fails if `arith` fails.
- Analyze the conclusion (*i.e.*, apply an introduction step) if it is a conjunction or a function-type.
- Apply a member of a user-defined list.

(a second clause dealing with integer arithmetic is omitted). Except for `Member` and the last clause, all the clauses of the autotactic are strongly valid: if a goal is provable, then each subgoal generated by the clause is also provable. This property, the conservative nature of `Member`, and the fact that little backtracking is done, make it feasible to apply the autotactic everywhere desired. For example, users typically call it after almost every top-level application of one of the tactics discussed in Section 3.5.

3.10 An Example From Number Theory

One of the first substantial developments of mathematics carried out in Nuprl was a proof of the fundamental theorem of arithmetic. The complete self-contained library constructed for this theorem contains 113 objects, of which 59 are definitions and 54 are theorems. 36 of the definitions are common to most existing libraries; they are mostly definitions for logical notions. There are 15 definitions dealing with basic list and integer relations and types, and 8 definitions which are particular to the development of the fundamental theorem of arithmetic.

Of the 54 theorems, 20 are associated with definitions in the way described in Section 3.1. Most of the remaining theorems establish simple properties of the defined objects. The fundamental theorem of arithmetic is formalized as the two Nuprl theorems

```
∀n:{2..}. ∃ l:PrimeFact where eval(l) = n
```

and

```
∀ l1,l2:PrimeFact. eval(l1) = eval(l2) => l1 = l2 in Fact.
```

The first statement can be read as: *for every integer n greater than or equal to 2 there is a prime factorization that multiplies out (or* evaluates*) to n.* The second can be read as: *any two prime factorizations that evaluate to the same number are the same factorizations.*

The program extracted from the first theorem above is a function mapping numbers to prime factorizations. This program is mostly developed in the proof of the following lemma (the theorem is just an instantiation of the lemma).

```
∀k:N. ∀n,i:{2..} where i≤n & n-i≤k & ∀d:{2..(i-1)}. ¬(d|n).
    ∃ l:PrimeFact where eval(l) = n
```

This lemma is proved by induction on k and suggests the algorithm used to compute prime factorizations. If we let $P(i,n)$ be

```
i≤n & ∀d:{2..(i-1)}. ¬(d|n),
```

$t(i,n)$ be n-i, and $R(n)$ be

```
∃l:PrimeFact where eval(l) = n,
```

then the lemma is equivalent to (omitting some types)

```
∀k,n,i. P(i,n) & t(i,n)≤k => R(n).
```

If $P(i,n)$ and $t(i,n) = 0$ then $n = i$ and n is prime, so the base case of the inductive proof of this lemma follows. For the induction case, assume $P(i,n)$. If $t(i,n) = 0$ we are done, otherwise we compute n' and i' such that $t(i',n') < t(i,n)$ and $P(i',n')$, and use the induction hypothesis. The obvious choices for i' and n' are $i+1$ and n in the case that i does not divide n, and i and n/i otherwise. In the first case $R(n)$ is proven by using the factorization given by $R(n')$; in the second, i is added to that factorization.

It is possible to express as a theorem in Nuprl a general inductive schema for problems that can be phrased, as in this case, in terms of an "invariant" P, a result assertion R and a bounding function t (although such a schema was not used in our proof of the fundamental theorem of arithmetic). One way of formulating such a schema is the following.

```
∀A,B:U1. ∀P:(A#B)->U1. ∀R:B->U1. ∀t:(A#B)->Int.
  ∀x:A#B. t(x) = 0 & P(x) => R(x.2)
  & ∀x:A#B. P(x) & 0<t(x) =>
      ∃y:A#B. P(y) & 0≤t(y)<t(x) & (R(y.2) => R(x.2))
  & ∀b:B. ∃a:A. 0≤t(<a,b>) & P(<a,b>)
  => ∀b:B. R(b)
```

To apply it to the lemma discussed above, we take A and B to be {2..} and P, R and t to be suitable modifications of what was given above.

All of the proofs in the fundamental theorem of arithmetic library were constructed using an early version of the tactic collection that was described in this chapter. This early tactic collection was designed to be generally applicable; none of the tactics were designed to be specifically useful in number theory.

The total time required to complete the library was about forty hours. This includes all work relevant to the effort; in particular, it includes the time spent on entering definitions, on informal planning, on lemma discovery and aborted proof attempts, and on proving all the necessary results dealing with the basic arithmetic operators and relations. The proofs contain a total of 879 refinement steps, most of which were entered manually (some were automatically applied by a primitive analogy tactic).

The fundamental theorem of arithmetic was chosen as a first major experiment with Nuprl because it is widely recognized as a non-trivial theorem in elementary number theory, because it has interesting computational content, and because mechanical proofs of it have been performed in other systems. It took approximately 8 weeks of effort to prove the theorem in the PLCV system [20]. PLCV is a natural deduction system for reasoning about PL/C programs which has powerful built-in support for arithmetical and propositional reasoning but no tactic mechanism or proof editor. Boyer and Moore [14] also conducted a proof of the fundamental theorem of arithmetic. They do not say how long the effort took, but they do say in their book that a correctness proof of a tautology checker required twelve hours of effort, and that the fundamental theorem of arithmetic was without doubt the hardest theorem proved so far using their system. A comparison with Boyer and Moore's proof is complicated by the fact that they started at a lower level by defining the integers and developing some elementary properties of numbers which are incorporated in Nuprl's `arith` procedure. On the other hand, they also modified their system by adding heuristics for proving theorems in number theory.

3.11 Saddleback Search

In this section we will take a closer look at the programming problem that was briefly discussed in Chapter 1.

The proofs that contain the development of the saddleback search algorithm are short enough to permit a complete presentation. The proof of the main lemma is by induction on k, and the first step is shown in Figure 3.1. The tactic used here performs induction on k, using l as a new variable for the induction step. The square brackets around some of the hypotheses in the subgoals indicate that the corresponding hypotheses are *hidden*. A hidden hypothesis cannot be used in any way, and remains hidden through further rule applications until the conclusion has a computationally trivial form (*e.g.*, when it is an equality).

The first subgoal is proved in two steps. The first step, shown in Figure 3.2, is to assert that the hypotheses are contradictory and to expand types which are

```
* top
>> ∀m,n:N+. ∀B:({0..(m-1)}->{0..(n-1)}->Int) where sorted(B{m,n}).
     ∀x:Int. ∀k:N. ∀i:{0..(m-1)}. ∀j:{0..(n-1)} where i+(n-j) = k.
          x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

BY (OnVar 'k' (NonNegInd 'l') ...)

1* 1. m: N+
   2. n: N+
   3. B: {0..(m-1)}->{0..(n-1)}->Int
   4. x: Int
   [5]. sorted(B{m,n})
   6. i: {0..(m-1)}
   7. j: {0..(n-1)}
   [8]. i+(n-j)=0
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

2* 1. m: N+
   2. n: N+
   3. B: {0..(m-1)}->{0..(n-1)}->Int
   4. x: Int
   5. l: int
   6. 0<l
   7. ∀i:{0..(m-1)}. ∀j:{0..(n-1)} where i+(n-j)=l-1.
          x ϵ B{m,n}(0:i, j:(n-1)) ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))
   [8]. sorted(B{m,n})
   9. i: {0..(m-1)}
   10. j: {0..(n-1)}
   [11]. i+(n-j)=l in Int
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))
```

Figure 3.1: By induction.

```
* top 1
...
>> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

BY (Assert 'False' ...) THEN (Unsetify ...)

1* 1. m: Int
   2. 0<m
   3. n: Int
   4. 0<n
   8. i: Int
   9. j: Int
   11. 0 ≤ j
   12. j ≤ n-1
   13. 0 ≤ i
   14. i ≤ m-1
   >> False
```

Figure 3.2: The base case is contradictory.

subsets. The display of this step in Nuprl would show as part of the main goal the hypotheses numbered 1 to 8 shown in Figure 3.1. In the interest of compactness, these have been manually elided (replaced by the string "..."). In addition, the system itself suppresses the display in subgoals of hypotheses that also appear in the goal. The second step, shown in Figure 3.3, is to show that a contradiction can be inferred from the known equalities and inequalities. This is accomplished by using tactics based on the monotonicity rule to add hypothesis 10 to hypothesis 13, and then to add n to both sides of 13.

The first step of the proof of the induction step (the second subgoal in Figure 3.1) is shown in Figure 3.4. In this step, we are considering the block of the matrix that extends from the top to row i and from column j to the right. The induction hypothesis (hypothesis 7 in the second subgoal in Figure 3.1) asserts the decidability of membership of x in smaller blocks of B. As in the informal description of the algorithm, we do a case analysis on the relation between x and $B(i,j)$. The case where $B(i,j) = x$ is easy; we can justify the left disjunct by providing the coordinates where x occurs. This step is shown in Figure 3.5. For the case where $B(i,j) < x$, we want to discard column j and use the induction hypothesis. There is no point doing this if column j is the last column, however, since in that case x does not occur in the block being considered. Hence we perform the case analysis shown in Figure 3.6.

Consider first the case where $j < n-1$. As shown in Figure 3.7, we apply the induction hypothesis to the smaller block of B obtained by removing column j. Now, x is in the larger block just in case it is in the smaller block (Figure 3.8). The first case is shown in Figure 3.9. If x is in the smaller block, say at position $\langle r,s\rangle$,

```
* top 1 1
1. m: Int
2. 0<m
3. n: Int
4. 0<n
5. B: {0..(m-1)}->{0..(n-1)}->Int
6. x: Int
7. sorted(B{m,n})
8. i: Int
9. j: Int
10. i+(n-j)=0 in Int
11. 0 ≤ j
12. j ≤ n-1
13. 0 ≤ i
14. i ≤ m-1
>> False

BY (Mono 10 '+' 12 THEN MonoWithR 13 '+' 'n = n' ...)
```

Figure 3.3: Contradiction follows from known inequalities.

```
* top 2
...
>> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

BY (Cases ['B(i,j)<x'; 'B(i,j)=x'; 'x<B(i,j)'] ...)

1* 12. B(i,j)<x
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

2* 12. B(i,j)=x
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

3* 12. x<B(i,j)
   >> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))
```

Figure 3.4: The main case split.

```
* top 2 2
...
12. B(i,j) = x
>> x ∈ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∈ B{m,n}(0:i, j:(n-1)))

BY (ILeft THENW ITerms ['i';'j'] ...)
```

Figure 3.5: In this case, x has been located.

```
* top 2 1
...
12. B(i,j)<x
>> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

BY (Cases ['j<n-1'; 'j = n-1'] ...)

1* 13. j<n-1
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

2* 13. j = n-1
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))
```

Figure 3.6: Is the column to be discarded the last one?

```
* top 2 1 1
...
7. ∀i:{0..(m-1)}. ∀j:{0..(n-1)} where i+(n-j)=l-1.
      x ϵ B{m,n}(0:i, j:(n-1)) ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))
...
12. B(i,j)<x
13. j<n-1
>> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

BY (InstHyp ['i'; 'j+1'] 7 ...)

1* 14. x ϵ B{m,n}(0:i, j+1:(n-1)) ∨  ¬(x ϵ B{m,n}(0:i, j+1:(n-1)))
   >> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))
```

Figure 3.7: Try to find x in the smaller block of B obtained by removing column j.

```
* top 2 1 1 1
...
12. B(i,j)<x
13. j<n-1
14. x ∊ B{m,n}(0:i, j+1:(n-1)) ∨  ¬(x ∊ B{m,n}(0:i, j+1:(n-1)))
>> x ∊ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ∊ B{m,n}(0:i, j:(n-1)))

BY (OnLastHyp CaseHyp THENL [ILeft;IRight] ...)

1* 14. x ∊ B{m,n}(0:i, j+1:(n-1))
   >> x ∊ B{m,n}(0:i, j:(n-1))

2* 14. ¬(x ∊ B{m,n}(0:i, j+1:(n-1)))
   15. x ∊ B{m,n}(0:i, j:(n-1))
   >> void
```

Figure 3.8: x occurs if and only if it occurs in the smaller block.

```
* top 2 1 1 1 1
...
12. B(i,j)<x
13. j<n-1
14. x ∊ B{m,n}(0:i, j+1:(n-1))
>> x ∊ B{m,n}(0:i, j:(n-1))

BY (OnLastHyp (CHThen (RepeatProductE ``r s``))
    THEN ITerms ['r'; 's'] ...)
```

Figure 3.9: If x occurs in the smaller block, we're done.

then it must also be in the larger block at position $\langle r, s\rangle$. The case where x is not in the smaller block is handled in Figure 3.10. We are assuming that x must occur in the larger block, and need to obtain a contradiction. Let r and s be the position of x in the larger block. We need to know whether $\langle r, s\rangle$ is in column j or in the smaller block, so we do the case analysis shown in the figure. The first case, where $s = j$, is shown in Figure 3.11; a contradiction follows from the fact that column j is sorted (and so $B(r,j) \leq B(i,j)$). In the second case, shown in Figure 3.12, we get a contradiction to hypothesis 14 by showing that x does occur in the smaller block. This finishes the proof of the case where $j < n-1$.

In the case where $j = n-1$, we prove (Figure 3.13) that x does not occur by contradiction. If it occurred at position $\langle r, s\rangle$, say, then the fact that column j is column sorted would give us $B(r,j) \leq B(i,j)$, from which a contradiction is immediate.

This completes the proof in the case that $B(i,j) < x$. The case $x < B(i,j)$ is

```
* top 2 1 1 1 2
...
12. B(i,j)<x
13. j<n-1
14. ¬(x ∊ B{m,n}(0:i, j+1:(n-1)))
15. x ∊ B{m,n}(0:i, j:(n-1))
>> void

BY (OnLastHyp (CHThen (RepeatProductE ''r s''))
    THEN Cases ['s = j'; 'j<s'] ...)

1* 15. r: {0..(m-1)}
   16. s: {0..(n-1)}
   17. s = j
   18. j ≤ s
   19. s ≤ n-1
   20. B(r,s)  =  x
   21. 0 ≤ r
   22. r ≤ i
   >> void

2* 15. r: {0..(m-1)}
   16. s: {0..(n-1)}
   17. j<s
   18. j ≤ s
   19. s ≤ n-1
   20. B(r,s)  =  x
   21. 0 ≤ r
   22. r ≤ i
   >> void
```

Figure 3.10: x does not occur in the smaller block. Suppose it occurs at $\langle r, s\rangle$ the larger block. We need to know whether $\langle r, s\rangle$ is in column j.

```
* top 2 1 1 1 2 1
...
8. sorted(B{m,n})
...
12. B(i,j)<x
13. j<n-1
14. ¬(x ∊ B{m,n}(0:i, j+1:(n-1)))
15. r: {0..(m-1)}
16. s: {0..(n-1)}
17. s = j
18. j ≤ s
19. s ≤ n-1
20. B(r,s)  =  x
21. 0 ≤ r
22. r ≤ i
>> void

BY (ColSorted 'j' 'r' 'i' 8 ...)
```

Figure 3.11: The contradiction follows from the fact that column j is sorted.

```
* top 2 1 1 1 2 2
...
12. B(i,j)<x
13. j<n-1
14. ¬(x ∊ B{m,n}(0:i, j+1:(n-1)))
15. r: {0..(m-1)}
16. s: {0..(n-1)}
17. j<s
18. j ≤ s
19. s ≤ n-1
20. B(r,s)  =  x
21. 0 ≤ r
22. r ≤ i
>> void

BY (E 14 THENO ITerms ['r';'s'] ...)
```

Figure 3.12: $\langle r, s\rangle$ must be in the smaller block, contradicting 14.

```
* top 2 1 2
...
8. sorted(B{m,n})
...
12. B(i,j)<x
13. j = n-1
>> x ϵ B{m,n}(0:i, j:(n-1))  ∨  ¬(x ϵ B{m,n}(0:i, j:(n-1)))

BY (IRight ...)
   THEN (OnLastHyp (CHThen (RepeatProductE ''r s'')) ...)
   THEN (ColSorted 'j' 'r' 'i' 8 ...)
```

Figure 3.13: When j is the last column of B, x does not occur, since if it did, the fact that column j is sorted would be contradicted.

analogous, so it will not be discussed. Except for this analogous branch, the figures we have given cover the entire Nuprl proof of the lemma.

The main theorem follows in two steps from the lemma. The first step, shown in Figure 3.14, is to instantiate the lemma using $m+n-1$, $m-1$, and 0 for k, i, and j respectively. The final step (Figure 3.15) is to remove n from the inequality in hypothesis 6.

For completeness we show in Figure 3.16 one of the two ML objects in the lbirary. This object defines the tactic `ColSorted` which was used in the steps shown in Figures 3.11 and 3.13. This tactic was written to shorten the proof, and its function is to instantiate the property `sorted(B{m,n})` on two positions within a single column. The second ML object defines an analogous tactic `RowSorted`.

```
* top
>> ∀m,n:N+. ∀B:({0..(m-1)}->{0..(n-1)}->Int) where sorted(B{m,n}). ∀x:Int.
     x ϵ B{m,n}(0:(m-1), 0:(n-1))  ∨  ¬(x ϵ B{m,n}(0:(m-1), 0:(n-1)))

BY (Id ...) THEN
   (LemmaUsing 'saddleback_lemma' ['m+n-1'; 'm-1'; '0'] ...)

1* 1. m: N+
   2. n: N+
   3. B: {0..(m-1)}->{0..(n-1)}->Int
   4. sorted(B{m,n})
   5. x: Int
   6. m+n-1<0
   >> void
```

Figure 3.14: Use the main lemma, with $m+n-1$ for the bound, $m-1$ for starting row i, and 0 for the starting column j.

```
* top 1
1. m: N+
2. n: N+
3. B: {0..(m-1)}->{0..(n-1)}->Int
4. sorted(B{m,n})
5. x: Int
6. m+n-1<0
>> void

BY (MonoWithR 6 '-' '0<n' ...)
```

Figure 3.15: Subtract `0<n` from hypothesis 6, getting `m-1<0`, contradicting `m>0`.

```
let ColSorted col row1 row2 i p =
  let n = number_of_hyps p  in
  ( CH i
    THEN (E i)
    THEN InstantiateHyp [col; row1; row2] (n+1)
    THEN Thinning [n+1; n+2]
  ) p
;;
```

Figure 3.16: One of two simple tactics.

Chapter 4

Girard's Paradox

This chapter gives a detailed account of our use of Nuprl in obtaining and analyzing the computational content of Girard's paradox. See Chapter 1 for an introduction to the problem. Section 4.1 presents the variant of $\lambda^{\tau\tau}$ that corresponds to the subset of the Nuprl type theory that was used in the formalization of the paradox. In Section 4.2 is a proof of Girard's paradox and a discussion of its formalization. In Section 4.3 is an analysis of the computational behaviour of the resulting term. At end are a few concluding remarks.

4.1 The Nuprl Subset

In this section we present a polymorphic typed λ-calculus $\nu^{\tau\tau}$ that has dependent types and a type of all of types. It is a simple matter to check that the Nuprl proof of Girard's paradox that we constructed yields a proof in $\nu^{\tau\tau}$; the details of this are uninteresting and are given in Appendix A. It seems certain that the Nuprl proof also yields a proof in $\lambda^{\tau\tau}$, but this has not been verified.

The set of terms of $\nu^{\tau\tau}$ is the smallest set containing an infinite set of variables, the constant τ, $a(b)$ whenever a and b are terms, $\lambda x.\, b$ whenever x is a variable and b is a term, and $\Pi\, x : A \,.\, B$ whenever x is a variable and A and B are terms. The usual notions of binding and substitution apply. We write $t[a/x]$ to denote the result of substituting a for x in t. The rules of $\nu^{\tau\tau}$ deal with *sequents*

$$x_1 : A_1, \ldots, x_n : A_n \vdash t \in T,$$

where the variables x_i are all distinct, for each i the set of free variables of A_i is contained in $\{x_1, \ldots, x_{i-1}\}$, and the set of free variables of $t \in T$ is contained in $\{x_1, \ldots, x_n\}$. We call the portion of a sequent to the left of the turnstyle a *context* or an *assumption list*. The extensions of a context $\mathcal{A}$ by an assumption $x : A$ or by another context $\mathcal{A}'$ are denoted by $\mathcal{A}, x : A$ and $\mathcal{A}, \mathcal{A}'$ respectively. We write $t \to t'$ if t' can be obtained from t by a sequence of β-reductions (*i.e.*, a sequence of replacements of subterms $(\lambda x.\, b)(a)$ by $b[a/x]$). Two terms will be considered identical if they are the same up to renaming of bound variables.

The rules of $\nu^{\tau\tau}$ are as follows.

τ-formation
$$\mathcal{A} \vdash \tau \in \tau$$

Π-formation
$$\frac{\mathcal{A} \vdash A \in \tau \quad \mathcal{A},\, x\!:\!A \vdash B \in \tau}{\mathcal{A} \vdash \Pi\, x\!:\!A\,.\,B \in \tau}$$

λ-intro
$$\frac{\mathcal{A} \vdash A \in \tau \quad \mathcal{A},\, x\!:\!A \vdash b \in B}{\mathcal{A} \vdash \lambda x.\, b \in \Pi\, x\!:\!A\,.\,B}$$

function-elim
$$\frac{\mathcal{A} \vdash f \in \Pi\, x\!:\!A\,.\,B \quad \mathcal{A} \vdash a \in A}{\mathcal{A} \vdash f(a) \in B[a/x]}$$

assumption
$$\mathcal{A},\, x\!:\!A,\, \mathcal{A}' \vdash x \in A$$

reduction-1
$$\frac{\mathcal{A} \vdash t \in T' \quad \mathcal{A} \vdash T \in \tau \quad T \to T'}{\mathcal{A} \vdash t \in T}$$

reduction-2
$$\frac{\mathcal{A},\, x\!:\!A',\, \mathcal{A}' \vdash t \in T \quad A \to A'}{\mathcal{A},\, x\!:\!A,\, \mathcal{A}' \vdash t \in T}$$

4.2 Girard's Paradox

Girard's paradox is an adaptation by Girard [29] of the well-known Burali-Forti paradox. The proof of the paradox that we use is somewhat different from Girard's.

Informally, the argument proceeds as follows. An *ordering* is a type together with a transitive binary relation. An ordering is *well-founded* if it has no infinite descending chains of elements. If the collection of all types is itself a type, then we can form the collection of all well-founded orderings. This collection and the relation of embedding together form an ordering. This ordering, call it *Uord*, is well-founded. Any well-founded ordering can be embedded in *Uord*, so *Uord* is embedded in itself. Thus the constant chain whose elements are *Uord* is an infinite descending chain in *Uord*, contradicting well-foundedness.

Formalizing this argument in $\nu^{\tau\tau}$ involves the propositions-as-types correspondence, whereby a proposition is associated with a type of $\nu^{\tau\tau}$ in such a way that the members of the type correspond to constructive proofs of the proposition. In this scheme, universal quantification corresponds directly to the Π type. Most of the other concepts of logic must be coded. Falsity is represented by the type $\Pi\, t\!:\!\tau\,.\,t$; a member of this type would be a function taking any type and producing a member of that type. A formalization of the paradox in $\nu^{\tau\tau}$ yields a term that is a member of $\Pi\, t\!:\!\tau\,.\,t$, so $\nu^{\tau\tau}$ is inconsistent when viewed as a logic.

This term does not normalize. Meyer and Reinhold conjectured that for hypothetical $T \in \tau$ and $F \in T \to T$, the construction of the non-normalizing term could be modified in such a way as to make it the application of a fixed-point combinator to F. We used Nuprl and a subset of the tactic collection described in Chapter 3 to formally prove a suitably modified version of the paradox. The term extracted from this formal proof can be show in $\nu^{\tau\tau}$ to have the type T. We then analyzed this term with the help of Nuprl's symbolic computation facilities, showing that it is an application of a *looping* combinator to F, but *not* an application of a fixed-point combinator. An important point here is that Nuprl made it possible to quickly construct in a natural way a formal argument that looks very much like a detailed version of the informal one and that produces via extraction the term to be analyzed. The complete Nuprl proof comprises eleven lemmas, with an average of about ten steps per lemma.

We now give a detailed version of the proof of Girard's paradox that was sketched earlier. We will point out the modifications that are made to obtain the application of the looping combinator to F. We will then briefly discuss the translation of the modified proof into Nuprl. This translation is sufficiently direct that the interested reader should be able to determine the correspondence between the detailed argument below and the terms discussed in the next section.

The definitions of embedding, U, and $<_U$ given below are variants of those given by Coquand [24]. We will not present the well-known encodings for logic; as an example, existential quantification is defined by

$$\exists\, x \in A\,.\, B \;\equiv\; \Pi\, C{:}\tau\,.(\Pi\, x{:}A\,.\,B \rightarrow C) \rightarrow C$$

where $A \rightarrow B$ abbreviates $\Pi\, x{:}A\,.\,B$ when x does not occur free in B. We will use *Prop* and *Type* as synonyms for τ.

An *ordering* is a type A together with propositional functions $r \in A \rightarrow A \rightarrow Prop$ and $d \in A \rightarrow Prop$. r is to be understood as a relation over A, and d as a subset of A. Typical names for orderings will be (A, r, d) and (B, s, e). We will be somewhat sloppy about notation for function application, writing, for example, $f(x, y)$ for $f(x)(y)$. We will often abbreviate by writing R for (A, r, d) and S for (B, s, e); we will occasionally use subscripts with these notations.

If $f \in A \rightarrow B$, $b \in B$ and

$$\begin{aligned}
& e(b)\ \&\ \forall\, x \in A\,.\ d(x) \Rightarrow e(f(x)) \\
& \&\ \forall\, x, y \in A\,.\ d(x) \Rightarrow d(y) \\
& \qquad \Rightarrow r(x,y) \Rightarrow s(f(x), f(y)) \\
& \&\ \forall\, x \in A\,.\ d(x) \Rightarrow s(f(x), b)
\end{aligned}$$

then (f, b) *embeds* R in S. The first ordering is *embedded* in the second, written $R < S$, if such an f and b exist. We will refer to f and b as the *order-preserving map* and the *bound* of the embedding, respectively.

Define transitivity, *trans*(R), in the obvious way. R is *well-founded* if there are no subsets P of A such that P is non-empty and such that for every y in P there is an x in P with $r(x, y)$. We call such subsets *chains*. More formally, $P \in A \rightarrow Prop$

is a chain in R if

$$\exists\, x \in A\,.\, P(x)\ \&\ d(x)$$
$$\&\ \forall\, y \in A\,.\, P(y) \Rightarrow \exists\, x \in A\,.\, P(x)\ \&\ r(x,y).$$

Well-foundedness is defined by:

$$wf(R)\ \equiv\ \forall\, P \in A \to Prop\,.\, (P\ chain\ in\ R \to false).$$

The definition of $wf(R)$ is the only one that needs to be changed in order to obtain the looping combinator; we replace *false* by the assumed type T. A more usual definition of well-foundedness is a positive one:

$$\forall\, P \in A \to Prop\,.\, (\forall\, y\,.\, (\forall\, x\,.\, r(x,y) \Rightarrow P(x)) \Rightarrow P(y)) \Rightarrow \forall\, x\,.\, P(x)$$

However, I do not see how to carry out the construction and analysis of a looping combinator using this definition.

Define the type U to be

$$(\Pi\, B\colon Type.(B \to B \to Prop) \to (B \to Prop) \to Prop) \to Prop$$

and the function i of type

$$\Pi\, B\colon Type.(B \to B \to Prop) \to (B \to Prop) \to U$$

by $i(A,r,d) = \lambda\, x\,.\, x(A,r,d)$. U can be viewed as the collection of all sets of sets of orderings, and i as the function which associates to each ordering the set of all sets of orderings that contain it. Define

$$a = b\ \equiv\ \forall\, P \in U \to Prop\,.\, P(a) \Rightarrow P(b).$$

This defines an equivalence relation. The function i is injective in the sense that if $i(R) = i(S)$ then the two orderings have the same properties (see Lemma 1 below). The ordering on U is defined by $u <_U v$ if there are R and S such that $u = i(R)$, $v = i(S)$, and $R < S$. Finally, define $d_U(u)$ if there is an R such that $u = i(R)$, $trans(R)$ and $wf(R)$. We will refer to the ordering $(U, <_U, d_U)$ as *Uord*.

Our proof of the paradox consists of eight lemmas, to each of which except the first corresponds a lemma in the Nuprl formalization. The parenthesized numbers in the lemma statements below give the number of steps in the corresponding Nuprl proof. There are three additional lemmas in the Nuprl development; they contain a total of ten steps. In what follows we will omit type information when it is clear from the context.

Lemma 1 *If $i(R_1) = i(R_2)$ and P is such that $P(R_1)$, then $P(R_2)$.*

Proof. Instantiate the definition of equality with $\lambda\, u\,.\, u(P)$. □

Lemma 2 *(9). If $R_1 < R_2$ and (f_2, b_2) embeds R_2 in R_3 then there are f_3 and b_3 which embed R_1 in R_3 and such that $r_3(b_3, b_2)$.*

Proof. Let f_1, b_1 embed R_1 in R_2. Take f_3 to be $\lambda\, x\, .\, f_2(f_1(x))$, and b_3 to be $f_2(b_1)$. We check that this forms the required embedding. From the definition of embedding, $d_2(b_1)$ and f_2 maps d_2 to d_3, so $d_3(b_3)$. The second and third conjuncts of the definition are easily verified. For the fourth, suppose $d_1(x)$. Then $d_2(f_1(x))$ and $r_2(f_1(x), b_1)$. $d_2(b_1)$, so $r_3(f_2(f_1(x)), f_2(b_1))$. Finally, $r_3(f_2(b_1), b_2)$ follows from $d_2(b_1)$. □

Lemma 3 *(4+4). If $d_U(i(R))$ then trans(R) and wf(R).*

Proof. Use Lemma 1. □

Lemma 4 *(8). trans(Uord).*

Proof. This follows from lemmas 1 and 2. □

Lemma 5 *(31). wf(Uord).*

Proof. Suppose P is a chain in *Uord.* Then there is an R_0 such that $wf(R_0)$, $trans(R_0)$ and $P(i(R_0))$. Let Q be

$$\lambda\, a\, .\, \exists\, R\, .\, \exists\, f\, .\, P(i(R))\ \&\ (f, a)\ embeds\, R\ in\, R_0.$$

We show Q is a chain in R_0; this will complete the proof since $wf(R_0)$. $P(i(R_0))$ so there is an R with $R < R_0$ and $P(i(R))$. Let a_0 be the embedding bound. Then $Q(a_0)$ and, by the definition of embedding, $d_0(a_0)$.

Now suppose $Q(y)$. There are R_1 and f_1 where $P(i(R_1))$ and (f_1, y) embeds R_1 in R_0. $P(i(R_1))$ so there is an R_2 with $P(i(R_2))$ and $R_2 < R_1$. By Lemma 2 there are f_2 and x embedding R_2 in R_0 with $r_0(x, y)$. $Q(x)$ is immediate. □

The only non-trivial modification of the proof of the paradox that is required to obtain the looping combinator is in the proof of Lemma 5. The basic idea of the proof just given is to assume that there is a chain in *Uord* and obtain from it a chain in a *member* of *Uord.* With the definition of well-foundedness modified in the way described earlier (that is, by replacing *false* by T), to prove this lemma we must show how to obtain a member of T (instead of a member of *false*) whenever there is a chain in *Uord.* Given a chain P in *Uord,* we can obtain a chain Q in some R as in the unmodified proof. Since $wf(R)$, there is a member x of T. Instead of using x directly, we first apply F to it.

Denote by R_a the ordering

$$(A, r, \lambda\, x\, .\, d(x)\ \&\ r(x, a))$$

Lemma 6 *(9). If $d_U(i(R))$ and $d(a)$ then $d_U(i(R_a))$.*

Proof. It is straightforward to check that a chain in R_a is also a chain in R. □

Lemma 7 *(16). If $d_U(i(R))$ then $R < Uord$.*

Proof. Let f be $\lambda\, a \,.\, i(R_a)$ and b be $i(R)$. We check that f and b embed R in *Uord*. The first requirement follows by hypothesis, and the second by Lemma 6. If $d(a_1)$, $d(a_2)$ and $r(a_1, a_2)$ then the pair $(\lambda x.\, x, a_1)$ embeds R_{a_1} in R_{a_2}. Finally, if $d(a)$ then $(\lambda x.\, x, a)$ embeds R_a in R. □

Lemma 8 *(13). Contradiction.*

Proof. By Lemmas 4 and 5 and the assumption $Type \in Type$, $d_U(i(Uord))$. By Lemma 7, $Uord < Uord$, so

$$\lambda\, u \,.\, (u = i(Uord))$$

is a chain in *Uord*, contradicting $wf(Uord)$. □

With the modified definition of well-formedness, the conclusion of this lemma is T instead of *false*. The term t that is extracted from the Nuprl proof of this lemma is the term that is analyzed in the next section, and is the application of a looping combinator to F. As shown in Appendix A, it can be proved in $\nu^{\tau\tau}$ that $t \in T$ follows from the assumptions $T \in \tau$ and $F \in T \to T$.

The Nuprl library that formalizes the preceding argument consists of definitions of the basic notions of logic, definitions of the concepts particular to the paradox, and finally the sequence of lemmas forming the proof. The latter two groups correspond rather directly to the preceding account. Associated with most of the definitions are "pattern" theorems. These theorems are only partially proved and serve as patterns to two general tactics which are used to treat defined objects abstractly (see Chapter 2 for details).

The first object defined in the library is `Type`. This is defined to be the Nuprl term `U1`, which is the first universe of types in a cumulative hierarchy. We assume proven the lemma

```
>> Type in Type.
```

We also make assumptions concerning the function which is to be the argument to the looping combinator. Specifically, we assume proven the theorems `>> T in Type` and `>> F in T->T`, where `T` and `F` are Nuprl constants. The only reason that these last two assumptions were made was to avoid cluttering up the definitions and theorems with explicit parameterizations.

Some of the definitions for logic are shown in Figure 4.1. Some of the definitions are made directly using the Nuprl definition mechanism (see Chapter 2 for a description of this). Other objects are defined indirectly as being extractions from theorems (see Chapter 3). For these objects, the figure shows the extracted term and the associated definition.

In Figure 4.2 we show the rest of definitions of the library (except for some variations on the basic definitions for logic given earlier). Each definition is a pair

```
all:
∀<x>:<T>. <P> == <x>:<T> -> <P>

some:
∃<x>:<A>. <B> == ∀C. (∀<x>:<A>. <B>  =>  C)  =>  C

term_of(and_):
λ A B. ∀C:Type. (A  =>  B  =>  C)  =>  C

and:
<P> & <Q> == term_of(and_) (<P>) (<Q>)

or:
<P>∨<Q> == ∀C. (<P> => C)  =>  (<Q> => C)  =>  C

term_of(eq_):
λ A a b. ∀P:A->Type. P(a) => P(b)

eq:
<a>≡<b> ϵ <A> == term_of(eq_) (<A>) (<a>) (<b>)
```

Figure 4.1: Definitions for logic.

consisting of an extracted term (labelled by the name of the theorem it is extracted from) and a Nuprl definition with its right-hand side omitted (being the obvious application of a `term_of` term).

There are thirteen theorems in the library other than the ones related to definitions. Three of these are trivial (containing a total of 10 steps), and the others have statements which are almost identical to the statements of corresponding lemmas of the proof we gave above. A few figures should suffice to give a general idea of the character of the formal proofs. Later sections do not depend on any knowledge of the formal proofs, so the rest of this section can be skipped.

Figure 4.3 shows the first step in the Nuprl proof of Lemma 7. The step involves repeatedly breaking down implications and universal quantifications, and then introducing two terms (they appear within single quotes as arguments to `DIntro`—this tactic is described in Chapter 3) as the order preserving function and bound which form the asserted embedding. Proving the resulting subgoal requires showing that the term introduced as a bound is indeed a bound. The main step in proving this subgoal, as shown in Figure 4.4, is to introduce another embedding, then break down the resulting subgoals, and then to reduce the result. The two resulting subgoals are each proved in a single step. Finally, we show a step where a more general kind of tactic was used. In Figure 4.5 is a step from the Nuprl proof of Lemma 2, where a tactic incorporating a degenerate kind of resolution was used to completely

```
U_:  ( B:Type -> (B->B->Type) -> (B->Type) -> Type ) -> Type
U:  U

i_:  λ A r d. λx. x(A,r,d)
i:  i(<A>, <r>, <d>)

trans_:   λ A r d. ∀x,y,z:A. r(x,y) => r(y,z) => r(x,z)
trans:  trans(<A>,<r>,<d>)

chain_in_:    λ A r d. λP. ∃x:A. P(x) & d(x)
                            & ∀y:A. P(y) => ∃x:A. P(x) & r(x,y)
chain_in:    <P> chain in (<A>,<r>,<d>)

wf_:   λ A r d. ∀P:A->Type. P chain in (A,r,d) => T
wf:  wf(<A>,<r>,<d>)

Uless_:   λ u v. ∃A:Type. ∃r:A->A->Type. ∃d:A->Type.
                 ∃B:Type. ∃s:B->B->Type. ∃e:B->Type.
                 u≡i(A, r, d) ϵ U & v≡i(B, s, e) ϵ U
                 & (A,r,d) < (B,s,e)

Uless:  Uless

dU_:  λu. ∃A:Type. ∃r:A->A->Type. ∃d:A->Type.
          u≡i(A, r, d) ϵ U & trans(A,r,d) & wf(A,r,d)

dU:  dU
```

Figure 4.2: More definitions.

```
>> ∀A:Type. ∀q:A->A->Type. ∀c:A->Type.
      dU(i(A, q, c)) => (A,q,c) < (U,Uless,dU)


BY (Repeat ArrowIntro ...)
   THEN
   (DIntro [ 'λa. i(A, q, λx. c(x) & q(x,a))';
             'i(A, q, c)'] [] []  ...)

1* 1. A: Type
   2. q: A->A->Type
   3. c: A->Type
   4. dU(i(A, q, c))
   >> (λa. i(A, q, λx. c(x) & q(x,a)), i(A, q, c))
          embeds (A,q,c) in (U,Uless,dU)
```

Figure 4.3: First step of Lemma 7

```
1. A: Type
2. q: A->A->Type
3. c: A->Type
4. dU(i(A, q, c))
5. a1: A
6. c(a1)
>> (A,q,(λa. c(a) & q(a,a1))) < (A,q,c)

BY (DIntro ['λw. w'; 'a1'] [] [] ...)
   THEN (DIntro [][][] ...)
   THEN (Reduce ...)

1* 7. x: A
   8. c(x) & q(x,a1)
   >> c(x)

2* 7. x: A
   8. c(x) & q(x,a1)
   >> (q(x))(a1)
```

Figure 4.4: Another step from Lemma 7.

```
10. f2: A2->A3
11. b2: A3
12. f1: A1->A2
13. b1: A2
14. d3(b2)
15. ∀x:A2. d2(x) => d3(f2(x))
16. ∀x,y:A2. d2(x) => d2(y) => r2(x,y) => r3(f2(x),f2(y))
17. ∀x:A2. d2(x) => r3(f2(x),b2)
18. d2(b1)
19. ∀x:A1. d1(x) => d2(f1(x))
20. ∀x,y:A1. d1(x) => d1(y) => r1(x,y) => r2(f1(x),f1(y))
21. ∀x:A1. d1(x) => r2(f1(x),b1)
>> d3(f2(b1))

BY (Backchain ...)
```

Figure 4.5: A step from Lemma 2.

prove a subgoal. (Hypotheses giving the types of the variables `A1`, `r1`, etc., have been edited out.)

The rest of the steps of the proofs are fairly similar to the those just shown. They all use general-purpose tactics, the only tailoring of tactics to this particular effort being in the construction of the pattern theorems mentioned earlier. The time required to enter all the definitions and prove all the theorems was about two working days.

4.3 Analysis of the Extracted Term

The formalization of Lemma 8 yields a term t such that $\vdash t \in \mathtt{T}$ can be proved in $\nu^{\tau\tau}$. The term we will examine, call it L, is a reduced (via β-reductions) form of t. The term L actually contains constants which denote extractions from other Nuprl theorems. Since there is a linear ordering of theorem dependencies, these constants can be eliminated by successively replacing them by the terms they denote. The actual Nuprl terms to which we apply symbolic computation will retain these constants, since they provide convenient labels for certain subterms of interest. These constants are replaced by their denotations when necessary for computation.

A *head* β-reduction contracts the leftmost redex of a term of the form

$$(\lambda x.\, b)(a_1)\ldots(a_n).$$

Write $t \xrightarrow{h} t'$ if t reduces to t' via a sequence of head β-reductions, and $t \rightarrow t'$ if t reduces to t' via any sequence of β-reductions. Write $t \equiv t'$ if t and t' are the same except for the names of bound variables. Define $t_1 = t_2$ if t_1 and t_2 are the same up to β-conversion.

In what follows, we will be interested in the results of computations in which certain subterms are irrelevant. We will therefore consider generalized terms which are terms which contain a special new variable ι. We say that t is an *instance* of t', and write $t \leq t'$, if t can be obtained from t' by replacing some occurrences of ι by other generalized terms. A generalized term will be called *ground* if it does not contain ι. We also generalize the notion of reduction. If t and t' are generalized terms, then $t \rightarrow t'$ ($t \xrightarrow{h} t'$) if for all ground instances u and u' of t and t', respectively, $u \rightarrow u'$ ($u \xrightarrow{h} u'$). To verify $t \rightarrow t'$ using Nuprl's computation facilities we can proceed as follows. Replace by some distinct Nuprl variable all occurrences of ι in t, apply some reduction steps to obtain t'', and check that t'' is an instance of t'. In what follows we will treat generalized terms as just terms, using *ground* and *non-ground* to make the necessary distinctions.

The Looping Property

The basic idea, due to Meyer and Reinhold, behind the proof that L has the looping property is as follows. The proof of the paradox involves constructing a chain in *Uord* and deriving a contradiction (member of T) from *Uord*'s well-foundedness. This is reflected in the form of L, which is

$$wf(P^0)(pf^0)$$

where wf is the term extracted from the Nuprl proof of Lemma 5, P^0 is the chain in *Uord* from the proof of Lemma 8, and pf^0 is a term which constitutes evidence, or a proof, that P^0 is a chain. *Uord*'s well-foundedness gives the procedure wf which when applied to P^0 and pf^0, first produces a chain P^1 (with proof pf^1) in a *member* of the chain P^0 (which, by construction of P^0 is *Uord*) and then applies this member's well-foundedness proof (which is also wf). This is reflected in the reduction sequence

$$wf(P^0)(pf^0) \rightarrow \mathtt{F}(wf(P^1)(pf^1)).$$

P^1 and pf^1 are similar to P^0 and pf^0, so we can repeat the above, obtaining P^k and pf^k for $k > 1$. This establishes the required looping property.

The problem in turning the above idea into a proof is that it is necessary to characterize the reduction behaviour of a massive term. It seems likely that this task is not feasible for a human without the aid of a computer. We have solved the problem by reducing it to a small set of machine-checkable assertions.

The proof we give below involves verifying certain properties of certain terms using symbolic computation. In order to be clear about what facts were checked using a machine, we will use a special notation. When we assert $a \Rightarrow b$ ($a \overset{h}{\Rightarrow} b$), we mean that $a \to b$ ($a \xrightarrow{h} b$) was verified by applying a (head) reduction sequence to $a[x/\iota]$, where x is some new Nuprl variable, and checking by inspection that the result, with x replaced by ι, was an instance of b. The reduction strategy generally used to check these assertions was normalization while holding constant a small number of terms that were extractions from theorems. (A transcript of the session in which these computations were done is available from the author.)

Theorem 1 *For each k there is a ground term L_k such that $L_0 \equiv L$ and for all k, $L_k \to \mathtt{F}(L_{k+1})$.*

Proof. We first make some simple definitions. For terms a and b, let $\langle a, b\rangle$ denote $\lambda tf.\, f(a)(b)$, and let $\pi_1(a)$ and $\pi_2(a)$ denote $a(\iota)(\lambda xy.\, x)$ and $a(\iota)(\lambda xy.\, y)$, respectively. We assume that the variables in the definition of $\langle a, b\rangle$ are always named so as to avoid capturing free variables of a and b. We extend the pairing notation to n-tuples by right-associating. Next, let *eq*, *id*, pf_1, pf_2 and pf_3 be the terms $\lambda P.\, \lambda x.\, x$, $\lambda x.\, x$, $\lambda xy.\, \pi_1(y)$, $\lambda uvwxy.\, y$ and $\lambda xy.\, \pi_2(y)$, respectively. Finally, let d and wf be the terms extracted from, respectively, the Nuprl proof that $i(\mathit{Uord})$ is in d_U and the Nuprl proof of Lemma 5 (actually, these terms are just Nuprl constants denoting other terms, as discussed earlier).

We will inductively define (non-ground) terms f^k, p_n^k and s_n such that

$$L \to \mathtt{F}(wf(\iota)(\langle\langle \iota, p_0^1, d\rangle, f^1\rangle))$$

and for all $k > 0$ and $n \geq 0$,

$$wf(\iota)(\langle\langle \iota, p_0^k, d\rangle, f^k\rangle) \to \mathtt{F}(wf(\iota)(\langle\langle \iota, p_0^{k+1}, d\rangle, f^{k+1}\rangle))$$

and

$$f^k(\iota)(p_n^k) \to \langle \iota, p_{n+1}^k, s_n\rangle.$$

Intuitively, the tuple $\langle \iota, p_0^k, d\rangle$ is evidence that the chain P^k is non-empty (with ι taking the place of *Uord* and with p_0^k being evidence that *Uord* is in P^k), and f^k is a function which takes a member of P^k and returns another member which is smaller in the embedding ordering. In the tuple $\langle \iota, p_{n+1}^k, s_n\rangle$, ι takes the place of this other member, p_{n+1}^k is evidence that this member is a member of P^k, and s_n is evidence that this member is smaller than the first member. The three properties above suffice to prove the theorem, since we can apply the first property and then repeatedly apply the second to obtain L_k, for $k \geq 1$, as a ground instance of

$$wf(\iota)(\langle\langle \iota, p_0^k, d\rangle, f^k\rangle).$$

Define, for t a term,

$$\begin{aligned} \mathcal{S}_0[t] &\equiv \langle \iota, \iota, \iota, \iota, \iota, \iota, eq, eq, id, \iota, t, pf_1, pf_2, pf_3\rangle \\ \mathcal{S}[t] &\equiv \langle \iota, \iota, \iota, \iota, \iota, \iota, eq, eq, id, \iota, \langle \iota, t\rangle, \lambda xp.\, \langle \pi_1(p), \iota\rangle, pf_2, pf_3\rangle \end{aligned}$$

Define the terms s_n by $s_0 = S_0[d]$ and $s_{n+1} = S[s_n]$. Next, for terms t and t' let $\mathcal{P}_1[t, t']$ be the term

$$\langle \iota, \iota, \iota, \iota, eq, \iota, \iota, t, t' \rangle.$$

We will define the terms f^k and p_n^k by induction on $k \geq 1$ and simultaneously show that they satisfy the required properties. For the case $k = 1$, we will define by induction on $n \geq 0$ terms op_n and bpf_n such that for all $n, j \geq 0$

$$op_n(\iota)(\iota)(\iota)(\iota)(s_j) \to s_{n+j+1}$$

and

$$bpf_n(\iota)(d) \to s_n.$$

For $n \geq 0$, p_n^1 will be

$$\mathcal{P}_1[op_n, bpf_n].$$

Now

$$L \Rightarrow \mathtt{F}(wf(\iota)(t))$$

where t is ground and

$$t \Rightarrow \langle\langle \iota, \mathcal{P}_1[op, bpf], d\rangle, f\rangle$$

for some op, bpf and f. Define op_0, bpf_0 and f^1 to be op, bpf and f respectively. The required properties of op_0 and bpf_0 follow from

$$op_0(\iota)(\iota)(\iota)(\iota)(\mathtt{sj}) \Rightarrow \mathcal{S}[\mathtt{sj}]$$

and

$$bpf_0(\iota)(\mathtt{d}) \Rightarrow \mathcal{S}_0(\mathtt{d})$$

(where sj and d are Nuprl variables). Suppose that we have defined op_n and bpf_n, and that they have the required properties. We have

$$f^1(\iota)(\mathcal{P}_1[\mathtt{opn}, \mathtt{bpfn}]) \Rightarrow \langle \iota, \mathcal{P}_1[op, bpf], \mathtt{bpfn}(\iota)(d)\rangle$$

for some terms op and bpf (where opn and bpfn are Nuprl variables). Define op_{n+1} and bpf_{n+1} to be $op[op_n/\mathtt{opn}]$ and $bpf[bpf_n/\mathtt{bpfn}]$ respectively. By the induction hypothesis, $bpf_n(\iota)(d) \to s_n$, so if we can verify the required properties for op_{n+1} and bpf_{n+1} then we will have completed the construction of the terms p_n^1 and shown that for all $n \geq 0$

$$f^1(\iota)(p_n^1) \to \langle \iota, p_{n+1}^1, s_n\rangle.$$

The required properties follow from

$$op(\iota)(\iota)(\iota)(\iota)(\mathtt{sj}) \Rightarrow (\mathtt{opn})(\iota)(\iota)(\iota)(\iota)(\mathcal{S}[\mathtt{sj}])$$

and

$$bpf(\iota)(\mathtt{d}) \Rightarrow (\mathtt{opn})(\iota)(\iota)(\iota)(\iota)(\mathcal{S}_0[\mathtt{d}])$$

and the induction hypothesis.

Suppose now that $k \geq 1$ and that the terms f^k and p_n^k have been defined and satisfy

$$f^k(\iota)(p_n^k) \rightarrow \langle \iota, p_{n+1}^k, s_n \rangle. \quad (*)$$

For t, t' terms, denote by $\mathcal{P}[t, t']$ the term

$$\langle \iota, \iota, \iota, id, t, t', pf_1, pf_2, pf_3 \rangle.$$

For all $n \geq 0$, define p_n^{k+1} to be

$$\mathcal{P}[p_{n+1}^k, u],$$

where u is d if $n = 0$ and ι otherwise. For t a term denote by $\sigma(t)$ the term

$$t[p_0^k, f^k/\mathtt{pk0}, \mathtt{fk}]$$

Now

$$wf(\iota)(\langle\langle \iota, \mathtt{pk0}, d\rangle, \mathtt{fk}\rangle) \overset{h}{\Rightarrow} \mathtt{F}(w(\iota)(u))$$

for some w and u, where $w \overset{h}{\Rightarrow} wf$ and $u \overset{h}{\Rightarrow} \langle t, f\rangle$ for some t and f. Define f^{k+1} to be $\sigma(f)$. t is

$$\mathtt{fk}(\iota)(\mathtt{pk0})(v)(v').$$

This term, under σ, contains a subterm to which, by induction, $(*)$ can be applied. We have

$$(\langle \iota, \mathtt{pk1}, s_0\rangle)(v)(v') \Rightarrow \langle \iota, \mathcal{P}[\mathtt{pk1}, d], d\rangle.$$

This proves that

$$\sigma(t) \rightarrow \langle \iota, p_0^{k+1}, d\rangle.$$

It only remains to show that f^{k+1} has the required property. Suppose $n \geq 0$. Let r be $\mathcal{P}[\mathtt{p}, \iota]$ and r' be $\mathcal{S}[\mathtt{sn}]$. For t a term, let $\tau(t)$ be

$$t[p_0^k, f^k, p_{n+1}^k, s_n/\mathtt{pk0}, \mathtt{fk}, \mathtt{p}, \mathtt{sn}].$$

It suffices to show that

$$\tau(f(\iota)(r)) \rightarrow \langle \iota, p_{n+1}^{k+1}, s_n\rangle.$$

We have

$$f(\iota)(r) \Rightarrow \mathtt{fk}(\iota)(\mathtt{p})(a)(a')$$

for some a and a'. Now by induction

$$\tau(\mathtt{fk}(\iota)(\mathtt{p})) \rightarrow \langle \iota, p_{n+2}^k, s_{n+1}\rangle.$$

Since

$$(\langle \iota, \mathtt{q}, r'\rangle)(a)(a') \Rightarrow \langle \iota, \mathcal{P}[\mathtt{q}, \iota], \mathtt{sn}\rangle,$$

we are done. □

Not a Fixed-Point

We will now show that L is not an application of a fixed-point combinator. Let L_k, for $k \geq 0$, be the terms constructed as the proof of Theorem 1. Define u^k to be

$$\langle\langle \iota, p_0^k, d\rangle, f^k\rangle.$$

For $k \geq 1$, L_k is an instance of $wf(\iota)(u^k)$.

Figure 4.6 contains a printout of the term wf with some subterms elided. The identifiers ending in "_" are constants that name the theorems whose extracted terms they denote. From wf we obtain a term wf' by replacing the (unique) subterm of the form $\lambda \mathtt{y}.\, \mathtt{F}(t)$ by $\mathtt{X}(\lambda \mathtt{y}.\, \mathtt{F}(t))$, where $\mathtt{X}$ is a fresh variable. Note that t is headed by $\mathtt{y}$. The idea here is to block certain reductions in order to show that all reduction sequences starting from L must repeatedly satisfy a certain property. We will call a term t $\mathtt{F}$-*normal* if it has no subterm of the form $\mathtt{F}(t')$ such that $t \xrightarrow{h} \mathtt{F}(t')$. Define the term $\mathtt{F}^{(n)}(t)$ by $\mathtt{F}^{(0)}(t) = t$ and $\mathtt{F}^{(n+1)}(t) = \mathtt{F}(\mathtt{F}^{(n)}(t))$.

In the remainder of this section, all terms will be ground unless indicated otherwise.

Lemma 9 *If w, u, c and t are terms such that u and c do not contain $\mathtt{X}$, t is $\mathtt{F}$-normal, $w \xrightarrow{h} wf$, $u \to u^k$ for some $k \geq 1$, and $w(c)(u) \to \mathtt{F}^{(i)}(t)$ for some $i \geq 0$, then $t \xrightarrow{h} \mathtt{F}((\iota)(\iota)(u'))$ for some u' equal to an instance of u^{k+i+1} (i.e., where there exists $u'' \leq u^{k+i+1}$ with $u' = u''$).*

Proof. The proof is by induction on i. We will do the induction step first. Suppose, then, that $i > 0$. By standardization[1] we have

$$w(c)(u) \xrightarrow{h} wf(c)(u) \xrightarrow{h} \mathtt{F}(t')$$

for some t' with $t' \to \mathtt{F}^{(i-1)}(t)$. Therefore, to prove the induction step it suffices to show that t' has the form $w'(c')(u')$ where c' and u' do not contain $\mathtt{X}$, $w' \xrightarrow{h} wf$ and $u' \to u^{k+1}$.

Using Nuprl we find that

$$wf'(\iota)(\langle\langle \iota, \mathrm{pk0}, d\rangle, \iota\rangle) \stackrel{h}{\Rightarrow} \mathtt{X}(\lambda \mathtt{y}.\, \mathtt{F}(w_0(c_0)(u_0)))(v_0)$$

for some terms w_0, c_0, u_0, and v_0 that are not ground terms, are minimal with respect to $\leq$ and do not contain $\mathtt{X}$. It follows that

$$wf'(\iota)(u^k) \xrightarrow{h} \mathtt{X}(\lambda \mathtt{y}.\, \mathtt{F}(w_1(\iota)(u_1)))(v_1)$$

[1]Roughly, for every reduction sequence, there is a left-to-right reduction sequence with the same initial and final terms. For details, see Barendregt [4].

```
(λ P v0.
  (v0
    . . .
    (λ v3 v4.
      v3)
    T
    (λ x v9.
      (v9
        (dU_ x)
        (λ v3 v4.
          v4)
        T
        (λ A0 v18.
          (v18
            T
            (λ r0 v21.
              (v21
                T
                (λ d0 y.
                  (F
                    (y
                      (and_ (trans_ A0 r0 d0)
                             (wf_ A0 r0 d0))
                      (λ v3 v4.
                        v4)
                      (wf_ A0 r0 d0)
                      (λ v3 v4.
                        v4)
                      . . .
                      . . .)))))))))))
```

Figure 4.6: *wf* with some subterms elided.

for some terms w_1, u_1, and v_1 that are not necessarily ground terms, do not contain $\mathtt{X}$, and are minimal with respect to $\leq$. Note that, since w_1, u_1, and v_1 do not contain $\mathtt{X}$, for any c', w', u', v' and u, if $u \leq u^k$ and u and c' do not contain $\mathtt{X}$, and

$$wf'(c')(u) \xrightarrow{h} \mathtt{X}(\lambda \mathtt{y}.\,\mathtt{F}(w'(\iota)(u')))(v'),$$

then w', u' and v' do not contain $\mathtt{X}$. Removing the $\mathtt{X}$ and doing the final β-conversion in the reduction defining w_1 we get

$$wf(\iota)(u^k) \xrightarrow{h} \mathtt{F}(w_1[v_1/\mathtt{y}](\iota)(u_1[v_1/\mathtt{y}])).$$

In the proof of Theorem 1, we showed that

$$wf(\iota)(u^k) \xrightarrow{h} \mathtt{F}(w_2(\iota)(u_2))$$

where w_2 and u_2 are not necessarily ground, $w_2 \xrightarrow{h} wf$ and $u_2 \rightarrow u^{k+1}$. By the minimality of w_1, u_1, and v_1, we have

$$w_1[v_1/\mathtt{y}] \leq w_2 \quad \textit{and} \quad u_1[v_1/\mathtt{y}] \leq u_2.$$

Now,

$$wf'(c)(u) \rightarrow wf'(c)(u_0^k) \xrightarrow{h} \mathtt{X}(\lambda \mathtt{y}.\,\mathtt{F}(w_3(\iota)(u_3)))(v_3)$$

where u_0^k, w_3, u_3, and v_3 are instances of u^k, w_1, u_1, and v_1, respectively. Standardizing, we get

$$wf'(c)(u) \xrightarrow{h} \mathtt{X}(\lambda \mathtt{y}.\,\mathtt{F}(w_4(\iota)(u_4)))(v_4) \quad (*)$$

where, since w_4 must be in head normal form by the construction of wf', $w_4 \rightarrow w_3$, $u_4 \rightarrow u_3$, and $v_4 \rightarrow v_3$. Therefore,

$$w_4[v_4/\mathtt{y}] \rightarrow w_3[v_3/\mathtt{y}] \leq w_1[v_1/\mathtt{y}] \leq w_2 \xrightarrow{h} wf.$$

Note that wf was actually defined to be a constant denoting an extraction, so we must have

$$w_4[v_4/\mathtt{y}] \xrightarrow{h} wf.$$

Also,

$$u_4[v_4/\mathtt{y}] \rightarrow u_3[v_3/\mathtt{y}] \leq u_1[v_1/\mathtt{y}] \leq u_2 \rightarrow u^{k+1},$$

so $u_4[v_4/\mathtt{y}] \rightarrow u^{k+1}$. Removing the $\mathtt{X}$ in $(*)$, then, we have

$$w(c)(u) \xrightarrow{h} wf(c)(u) \xrightarrow{h} (\lambda \mathtt{y}.\,\mathtt{F}(w_4(\iota)(u_4)))(v_4) \xrightarrow{h} \mathtt{F}(w_4[v_4/\mathtt{y}](\iota)(u_4[v_4/\mathtt{y}])).$$

This completes the proof of the induction case.

We now prove the base case. We have $w(c)(u) \rightarrow t$, and must show that there is a u' such that $t \xrightarrow{h} \mathtt{F}((\iota)(\iota)(u'))$ and u' is equal to an instance of u^{k+1}. For v a

$$\begin{array}{ccccc}
w'(c)(u) & \xrightarrow{h} & wf'(c)(u) & \xrightarrow{h} & \mathtt{X}(\lambda \mathtt{y}.\, \mathtt{F}((s_1)(s_2)(u')))(v) \\
\downarrow & & & & \downarrow \\
t' & & \xrightarrow{h} & & r \equiv \mathtt{X}(\lambda \mathtt{y}.\, \mathtt{F}((s_3)(s_4)(u'')))(v') \\
\Downarrow & & & & \Downarrow \\
t & & \xrightarrow{h} & & r' \equiv (\lambda \mathtt{y}.\, \mathtt{F}((\iota)(\iota)(u''')))(v'')
\end{array}$$

Figure 4.7: Reductions in the base case argument.

term, let $\varepsilon(v)$ be the term obtained by repeatedly replacing subterms of the form $\mathtt{X}(a)$ by a.

Figure 4.7 contains a diagram which summarizes the reductions discussed in the following. Let w' be w with all occurrences of wf replaced by wf'. Using the argument in the induction case, we have

$$w'(c)(u) \xrightarrow{h} wf'(c)(u) \xrightarrow{h} \mathtt{X}(\lambda \mathtt{y}.\, \mathtt{F}((s_1)(s_2)(u')))(v)$$

where

$$u'[v/\mathtt{y}] \to u^{k+1},$$

and u' and v do not contain $\mathtt{X}$. Now consider the given reduction sequence from $w(c)(u)$ to t. By inspection of wf' (see Figure 4.6), we see that if b is such that $w'(c)(u) \to b$, and if, for some a and a', $\mathtt{X}(a)(a')$ is a subterm of b, then $a(a')$ is a β-redex and has no free variables, and if this redex is contracted, the resulting term is head normal (being headed by $\mathtt{F}$). It follows that we can obtain a reduction sequence from $w'(c)(u)$ to some t' where t can be obtained from t' by replacing maximal subterms of the form $\mathtt{X}(a_1)(a_2)$ or $\mathtt{X}(a_1)$ by a term a_3 where $\varepsilon(a_1(a_2)) \to a_3$ or $\varepsilon(a_1) \to a_3$, respectively. (Such a set of replacements is indicated in the diagram in Figure 4.7 by a double arrow.) By Church-Rosser and standardization,

$$t' \xrightarrow{h} \mathtt{X}(\lambda \mathtt{y}.\, \mathtt{F}((s_3)(s_4)(u'')))(v')$$

for some u'' and v'. By standardization and the fact that s_1 is head-normal, $u' \to u''$ and $v \to v'$. Let r be the term above that t' reduces to. The head reduction sequence from t' to r gives, step for step, a sequence $t \xrightarrow{h} r'$ for some r'. The replacements taking t' to t give rise to a replacement taking r to r'. With this replacement,

$$\varepsilon(\,(\lambda \mathtt{y}.\, \mathtt{F}((s_3)(s_4)(u'')))\,(v')\,) \to r'.$$

Suppose that this reduction involves contracting the outermost redex. Then, since also any subterm of t' of the form $\mathtt{X}(a)(a')$ must be closed, we must have that

$$\mathtt{X}(\lambda \mathtt{y}.\, \mathtt{F}((s_3)(s_4)(u'')))(v')$$

is a subterm of t', and that the replacements which take t' to t involve replacing this subterm by r'. This contradicts the assumption that t is F-normal. Since u'' and v' do not contain X, and since s_3 is head-normal, r' must be of the form

$$(\lambda \mathtt{y}.\, \mathtt{F}((\iota)(\iota)(u''')))(v'')$$

for some u''' and v'' where $u'' \to u'''$ and $v' \to v''$. Now

$$u'[v/\mathtt{y}] \to u''[v'/\mathtt{y}] \to u'''[v''/\mathtt{y}].$$

Since also $u'[v/\mathtt{y}]$ reduces to an instance of u^{k+1}, we are done. □

Lemma 10 *Any term t can be reduced to a term $\mathtt{F}^{(i)}(t')$ where t' is* F*-normal.*

Proof. The proof is a simple induction on the size of terms. Write t as $\mathtt{F}^{(i)}(t')$ with i maximal, and suppose t' is not F-normal. Then t' reduces to a term $\mathtt{F}(t'')$ which appears in t'. This term is smaller than t'. □

Theorem 2 *If $1 \le k < k'$, then $L_k \ne L_{k'}$.*

Proof. Suppose $L_k = L_{k'}$. Then L_k and $L_{k'}$ have a common reduct t. By Lemma 10 we may assume that t is of the form $F^{(i)}(t')$ where t' is F-normal. Applying Lemma 9, we get

$$t' \xrightarrow{h} \mathtt{F}((\iota)(\iota)(u)),$$

where u is equal to an instance of u^{k+i+1} and equal to an instance of $u^{k'+i+1}$. This means that there are instances of u^{k+i+1} and $u^{k'+i+1}$ that have a common reduct, which implies that there are instances of p_0^{k+i+1} and $p_0^{k'+i+1}$ that have a common reduct. From the definition of these latter two terms, it follows that there are instances of p^1_{k+i} and $p^{k'-k+1}_{k+i}$ that have a common reduct. But this is impossible, since the first of these terms is a nine-tuple whose fifth component is *eq*, while the second is a nine-tuple whose fifth component is also a nine-tuple. □

The following is immediate.

Theorem 3 *For all $i \ge 0$, $L_i \ne \mathtt{F}(L_i)$.*

4.4 Conclusion

We formalized one particular proof of Girard's paradox. It is possible that another proof would yield a term which is a fixed-point combinator, but it seems unlikely. The focus of the looping structure of the paradox seems to be in the proof that *Uord* is well-founded, where, to show that a particular entity does not exist (a chain in our case, and evidence that some member is less than itself in the case of Girard's original proof), one assumes that it does, transforms it into an analogous entity in a *member* of *Uord*, and applies that member's well-foundedness property. It seems certain that if the transformation adds any structure, then the fixed-point cannot be obtained.

Nuprl was used in an essential way in this effort. The formal proof of Girard's paradox was carried out at a fairly high level and in a reasonably short time (about two working days), and involved very little explicit reasoning about the components of the term that was being constructed. Nuprl's collection of tools for term manipulation were then used to verify certain properties of the huge resulting term. Nuprl also played a valuable role in the discovery of parts of the argument, since the straightforward formalization of the informal proof yielded a term in whose correctness one could have confidence, and on which one could quickly test certain kinds of conjectures using the various tools provided by the system.

The result of this effort indicates a possible role for mechanized formal problem solving environments such as Nuprl in mathematical research. There may arise other problems such as this one, where the computer plays an indispensable role in the solution. If this is the case, then systems like Nuprl, which can support high-level formal reasoning in a variety of problem domains and which provide a large set of facilities for manipulation and analysis of proofs and terms, will prove to be valuable tools.

Chapter 5

Partial Reflection

This chapter contains a detailed account of the partial-reflection library that was discussed in Chapter 1. A complete listing of this library can be found in Appendix C.

The best way to present the formal arguments contained in the library would involve the use of the Nuprl system. One of the main goals in the design of Nuprl and of the tactic collection of Chapter 3 was to allow the construction of proofs that could be easily comprehended through use of the system. Unfortunately, these proofs are not well-suited to presentation on paper. Therefore this chapter mostly contains explanations of the main definitions and theorems of the library. We also provide a few details of a few of the proofs in order to give some idea of the typical level of inference.

In the first section of this chapter is an overview of the library and brief discussion of an initial segment of the library that contains objects not particular to the partial reflection work. In the following section is a discussion of an implementation of a fragment of Bishop's set theory. The next four sections are devoted to the library objects and tactics that form the partial-reflection mechanism. The following two sections describe some implemented examples of the use of the mechanism, and the final section contains some general comments.

5.1 Overview and Preliminaries

The complete library contains about 1300 objects, of which about 800 are theorems, 400 are definitions and 100 are ML objects. The library can be roughly divided according to topic:

- Basics.
- Lists.
- Sets.

- The representation of Nuprl terms.
- Association lists and type environments.
- Function environments and combined environments.
- Booleans and "partial" booleans.
- Well-formedness and evaluation of meta-terms.
- Relations and operations on environments; monotonicity with respect to environment extension.
- Term rewriting.
- Applications.

The parts of the library corresponding to the first two topics are discussed in this section. Most of the objects in these two parts are simple and not particular to the partial-reflection library, so the discussion of them will be brief.

Our presentation of the library will be somewhat in the style of conventional mathematics. Usually, explicit references to the existence of library objects will not be made. Unless stated otherwise, whenever a definition is made, there is a corresponding definition in the library, and when a statement about defined objects is asserted to be true or provable, then there is a corresponding theorem in the library. In order to make the presentation smoother, we will sometimes use English for parts of Nuprl definitions or theorems. This will involve the use of metavariables, and will sometimes omit some details of the actual Nuprl version. For example, we might define the predicate `null` as follows.

For l a list, let `null(`l`)` be `list_ind(`l`; true; h,t,v.false)`.

Also, a theorem about the decidability of `null`, whose statement in the library is

$$\forall\mathtt{A:U1.}\ \forall\mathtt{l:A\ list.}\ \mathtt{null}(l)\vee\neg\mathtt{null}(l),$$

might be expressed as follows.

For all lists l, `null(`l`)`$\vee\neg$`null(`l`)`.

Hopefully these conventions will not cause any confusion. If they do, the reader can refer to Appendix C.

The "basics" section of the library includes definitions for logical concepts and definitions related to the integers. There are also definitions that provide alternate notations for simple combinations of base terms of the type theory. For example, we define a "let" construct in the usual way:

$$\mathtt{let}\ x\ \mathtt{=}\ t\ \mathtt{in}\ t' \equiv (\lambda\ x.t')(t).$$

We use it, together with the definitions for projection from pairs, in the definition `let2`:

`let` x`,`y `=` p `in` t $\equiv$ `let` x `=` p`.1 in let` y `=` p`.2 in` t.

Nuprl's cumulative hierarchy `U1`, `U2`, ... of universes of types is not internally indexed, and so one is confronted with a "universe polymorphism" problem: many theorems that mention universes do not depend on the particular universe levels mentioned. For example,

`∀A,B:U`i`. A&B in U`i

is true for any $i \geq 1$. This problem usually does not arise in applications of Nuprl since it is usually not necessary to mention any universe other than `U1`. As will be seen later, the partial-reflection mechanism requires higher universes, and so it becomes important to have a way to state a fact such as the above that does not involve duplicating the theorem for every universe level which might later arise. Allen [3] gives a semantic basis for solving this problem, but no general solution has as yet been implemented. For our purposes a rather simple-minded solution requiring no logical extensions is sufficient. Define ∞ to be the subtype of integers between 1 and 12, `Type` to be `U13`, and `U` to be the function in `∞->Type` which associates `U`i to each i in ∞. We can prove lemmas relating the type constructors to `U`. For example,

`∀i:∞. ∀A:Ui. ∀B:A->Ui. x:A->B(x) in Ui.`

These lemmas are "added" to the autotactic using the mechanism described in Chapter 3, and never have to be explicitly applied.

This universe polymorphism feature is extensively used in the development of list theory. For example, define

`∀∈` l`:` A `list.` P $\equiv$ `[nil → True; h.t,v →` P`(h) & v; @`l`]`

(where the right-hand side uses an alternate notation for the `list_ind` form). This is a proposition which is true if and only if P is true of every element of the list l. We can prove that

`∀i:∞. ∀A:Ui. ∀P:A->Ui. ∀l:A list. (∀∈ l: A list. P) in Ui.`

The "lists" section of the library has definitions for many of the standard functions over lists. For example, we define

`accumulate` f `over` l `from` x $\equiv$ `[nil →` x`; h.t,v →` f`(h,v); @`l`]`

and prove that

```
∀A,B:Type. ∀f:A->B->B. ∀x:B. ∀l:A list.
        (accumulate f over l from x) in B.
```

Another useful function is `com`, which combines two lists l_1 and l_2 and is defined by

```
[nil→ λl2.nil; h.t,f→ λl2. <h,hd(l2)>.f(tl(l2)); @l1] (l2).
```

There are also theorems which provide useful forms of list induction. We can perform a simultaneous induction over lists of the same length by using the following theorem.

```
∀A,B:Type. ∀P: l1:(A list) -> {l2:B list||l1|=|l2| in N} -> Type.
  P(nil,nil) =>
  ( ∀h1:A. ∀h2:B. ∀t1:A list. ∀t2:B list.
      |t1|=|t2| in N => P(t1,t2) => P(h1.t1, h2.t2) ) =>
  ∀l1:A list. ∀l2:B list. |l1|=|l2| in N => P(l1,l2).
```

Elimination of a list of length two can be done using the following.

```
∀A:Type. ∀l:A list. |l|=2 in N => ∀P:{l:A list||l|=2 in N}->Type.
        (∀a,b:A. P([a;b])) => P(l).
```

5.2 Set Theory

A basic problem in formalizing constructive analysis is in dealing with sets. According to Bishop [8], a set is "the totality of all mathematical objects constructed in accordance with certain requirements", together with an equality relation. To define a set one specifies how to construct members of the set and when two members are to be considered equal. This differs sharply from classical set theory, where there is a single global notion of equality. There, to define a set one just defines its members, possibly forming equivalence classes to achieve the desired equality.

There is (by design) a correspondence between the types of Nuprl and the sets of Bishop's mathematics. Types are also specified by defining the members and giving an equality relation. In order to allow the definition of types that have equalities other than the ones inherited from the basic type constructors, Nuprl includes quotient types, written $(x,y):A//E$ for $E(x,y)$ an equivalence relation. This type has the same members as A, but its equality is given by E. Although it might seem that this type-constructor would permit the construction in Nuprl of most of the sets appearing in elementary analysis, it turns out that it is not even adequate for the definition of the real numbers. See Chapter 10 of the Nuprl book [23] for a discussion of this.

We are thus forced into dealing with equality explicitly. We will consider a set to be a pair consisting of a type together with an equivalence relation on that type. In order to define the basic set constructors, we must first decide whether equality is to be computationally significant. If it is, we would define an equivalence relation to be a pair consisting of the relation together with the evidence, or proof, that the relation satisfies the transitivity, symmetry and reflexivity axioms. A function from a set S_1 to a set S_2 would be a *pair* consisting of an operation f from the underlying type of S_1 to that of S_2, together with a proof that f preserves equality. However, in much of Bishop's analysis, including calculus and the theory of metric spaces, equalities are of no computational interest. We will therefore use the set type to "hide" equality information. Thus functions on sets are not pairs, but particular Nuprl functions.

We now give some details of our formalization of set theory in Nuprl. We define a hierarchy of collections of sets that parallels the universe hierarchy. For $i \geq 0$, let `Set(`i`)` be

$$\texttt{A:Ui \# \{ r:(A\#A)->U}i\texttt{ | eq\_reln\{A:U}i\texttt{\}(r) \},}$$

where `eq_reln{A:U`i`}(`r`)`, which asserts that r is an equivalence relation, is defined in the obvious way. If S is a set, then its first and second components (S`.1` and S`.2`) are denoted by `|`S`|` and `=_(`S`)` respectively. If a and b are members of `|`S`|`, then we will denote `=_(`S`)(`a`)(`b`)` by a`=`b `in` S.

We also define some particular sets that will be used later. `Set` and `SET` are `Set(6)` and `Set(7)`, respectively, and `Prop` is

$$\texttt{<U6, } \lambda \texttt{ P,Q. P <=> Q>.}$$

Note that this last definition is stronger than necessary: it is convenient to have equality of propositions and `<=>` be the same, but if we were to be strictly consistent in our treatment of equality as having no computational significance, we would weaken the equality by applying the squash operator and would define

$$\texttt{Prop} \equiv \texttt{<U6, } \lambda \texttt{ P,Q. } \downarrow\texttt{P <=> } \downarrow\texttt{Q>.}$$

The choice of universe levels 6 and 7 in the definitions just given is rather arbitrary. They only requirement we have is that they be high enough to accommodate anticipated applications.

It is straightforward to define products and function spaces. The set S_1`->`S_2 of functions from S_1 to S_2 has as its type component

$$\texttt{\{ f:|}S_1\texttt{|->|}S_2\texttt{| | fnl\{}S_1\texttt{,}S_2\texttt{\}(f) \}}$$

and as its equality

$$\lambda \texttt{ f,g. } \forall \texttt{x:|}S_1\texttt{|. f(x)=g(x) in } S_2$$

(where the predicate `fnl` is defined in the obvious way). It is also easy to define the cartesian product S_1`#`S_2 of S_1 and S_2, and a generalization `#(`L`)` for non-empty lists of sets L.

Our definition of subset is a direct translation of Bishop's. He defines a subset of a set X to be a set S together with an injection from S to X. As an example consider the set $\boldsymbol{R}^+$ of positive real numbers. Constructively, to be given a positive real number means to be given a real number x together with a proof of positivity, that is, a positive integer n such that $1/n \leq x$. A Nuprl set representing $\boldsymbol{R}^+$ is the collection of all pairs consisting of an x together with an appropriate n. This set can be realized as a subset of $\boldsymbol{R}$ with the injection which projects the first component. To formalize this notion of subset, we define

$$\texttt{injective}(i\in S_1\texttt{->}S_2) \equiv \forall\texttt{x,y:}|S_2|\texttt{. i(x)=i(y) in } S_2 \texttt{ => x=y in } S_1$$

and

$$S_1 \subset S_2 \equiv \exists\texttt{i:}|S_1\texttt{->}S_2|\texttt{ where injective(i}\in S_1\texttt{->}S_2\texttt{)}.$$

5.3 Representation

In this section we discuss first the representation in Nuprl of a simple class of terms and then the representation of contexts (*i.e.*, of definitions and hypotheses).

Terms

The type `Term0` of "raw" terms which do not necessarily represent a *well-formed* object of the type theory is defined using the recursive type constructor as

```
rec(T. (Atom#T list) | (T#T#Atom) | (Atom#T#T) | Int ).
```

Members of this type, which will be referred to as *meta-terms*, or just *terms* when no confusion can result, can be thought of as trees with four kinds of nodes. The meaning of the different kinds of nodes is suggested by the notations for the injections:

$$f\texttt{(}l\texttt{)} \equiv \texttt{inl(<}f\texttt{,}l\texttt{>)}$$
$$x \texttt{ = } y \texttt{ in } A \equiv \texttt{inr(inl(<}x\texttt{,}y\texttt{,}A\texttt{>))}$$
$$y\{i\ x\} \equiv \texttt{inr(inr(inl(<}i\texttt{,}x\texttt{,}y\texttt{>)))}$$
$$n \equiv \texttt{inr(inr(inr(}n\texttt{)))}.$$

The above are all members of `Term0` whenever f, A, i are in `Atom`, x and y are in `Term0`, l is a list of members of `Term0` and n is an integer. These kinds of nodes will be referred to, respectively, as function-application nodes, equality-nodes, i-nodes, and integer nodes. The first will represent terms which are function applications; the second, terms which are equalities (in the set sense); the third, terms which are members of subsets, where i represents an injection such that $y = i(x)$; and the last, terms which are integers. We will also use the term *meta-constant* to refer to function-application nodes of the form f`(nil)`.

Using the induction rules for recursive types, we can prove a lemma which encapsulates a more natural form of induction over `Term0` than is provided by

Nuprl's primitive rule.

```
∀j:∞. ∀P:Term0->Uj.
      ∀f:Atom. ∀l:Term0 list. (∀∈l:Term0 list.P) => P(f(l))
   => ∀t,u:Term0. ∀A:Atom. P(t) => P(u) => P(t=u in A)
   => ∀i:Atom. ∀t,u:Term0. P(t) => P(u) => P(u{i t})
   => ∀n:Int. P(n)
   => ∀t:Term0. P(t).
```

We can also prove an analogous induction lemma for certain subsets of Term0. More specifically, if Q is *hereditary* over Term0, that is,

```
∀f:Atom. ∀l:Term0 list. Q(f(l)) => (∀∈ l : Term0 list. Q)
& ∀t,u:Term0. ∀A:Atom. Q(t=u in A) => Q(t) & Q(u)
& ∀i:Atom. ∀t,u:Term0. Q(u{i t}) => Q(t) & Q(u),
```

then a version of the previous induction lemma holds where {x:Term0|Q(x)} is substituted for Term0.

The main components of the partial reflection mechanism account for i-nodes, but all of the applications we have developed using the mechanism ignore them. They will be useful in future work that is more particular to constructive analysis. We will motivate their inclusion by considering a small example, but we will mostly ignore them in the rest of this chapter.

Consider the example of $\boldsymbol{R}^+$ given in the previous section. Let R and R^+ be Nuprl sets representing $\boldsymbol{R}$ and $\boldsymbol{R}^+$ respectively, and let i be the natural injection such that injective(i∈R^+->R). Computing the inverse of a real number requires knowing that the number is bounded away from 0; limiting to non-negative numbers, it requires a member of R^+. Thus the inverse of an $x \in R$ might be written inv(<x,n>) where n is such that <x,n>∈R^+. Using an i-node, we can represent this expression as

"inv"(["x"{"i" t}])

where t represents <x,n>. In Nuprl we could use a definition to suppress the display of the portion {"i" t}. The important point here is that in a meta-term $u\{i\ v\}$, the only part we will be interested in for the purpose of equality reasoning is u. The partial reflection mechanism is designed so that rewriting functions, for example, can be applied without concern for v.

Association Lists and Failure

The representation of contexts uses association lists, or "a-lists" for short. These are objects of type Atom#(A list) for A a type, and are used to associate Nuprl objects to the atoms that appear in meta-terms. In dealing with a-lists, and in the term-rewriting system described later, extensive use is made of a Nuprl version of

ML-style *failure*. For A a type, define $?A$ to be A|True, where True is a single-element type. A value inl(a) $\in$ $?A$ is denoted by s(a) (s for "success"), and the unique value of the form inr(a) is denoted by fail. In later sections, the distinction between a and s(a) will often be glossed.

The library has many objects related to failure and a-lists; we will only give a few examples. We define a *catch* operator by

$$t?t' \quad \equiv \quad \texttt{decide}(t\texttt{;u.s(u);v.}t')\,.$$

If t succeeds (*i.e.*, has a value of the form s(a)) then $t?t'$ succeeds with value a (*i.e.*, the value of $t?t'$ is s(a)). If t fails, it has whatever value t' has. We will need two simple definitions for accessing the value of a successful computation. First, for A and B types, f in A->$?B$, and a in $?A$, define

$$\texttt{slet}(B,f,a) \quad \equiv \quad \texttt{decide}(a\texttt{;x.}f\texttt{(x);x.fail)}\,.$$

We also define a useful notation for a certain instances of slet:

$$\texttt{let s(}x\texttt{) = }a\texttt{ in }t\texttt{: }?B \quad \equiv \quad \texttt{slet(}B\texttt{, }\lambda x.t\texttt{, }a\texttt{)}\,.$$

Secondly, for a a succeeding member of $?A$, define

$$\texttt{ds}(a) \quad \equiv \quad \texttt{d}(a\texttt{;x.x;x."uu")}\,.$$

Many of the definitions related to failure are analogous to ones that do not involve failure. For example, there is a definition for application of a function f of type A->$?B$ to an argument a of type ?A: if a fails then $f(a)$ fails; if a succeeds with value b then $f(a)$ is $f(b)$ using the usual definition of application.

The value in $?A$ associated to an atom a by an a-list l is

$l\{A\}(a)$ $\equiv$
[nil $\longrightarrow$ fail; h.t,v $\longrightarrow$ if h.1=x then s(h.2) else v; @ l],

where the part headed by if is an alternate notation for an atom_eq form (see Chapter 2). We use this definition to define the relation of containment between a-lists. Informally, $l \subset l'$ if for every atom a, if $l\{A\}(a)$ succeeds then $l'\{A\}(a)$ succeeds and has the same value.

Type Environments

A context is represented as a pair of a-lists, the first of which is a type environment and the second of which is a function environment. Type environments associate atoms with members of SET and are members of the type

$$\texttt{TEnv} \quad \equiv \quad \texttt{(Atom \# S:Set \# ?triv\_eq(S)) list}\,.$$

The actual Nuprl definition of `TEnv` uses several intermediate definitions. `triv_-eq(`S`)` is defined as

```
∀x,y:|S|. ↓(x=y in S) => x=y in S
```

and is true when S has a computationally trivial equality. The importance of this component of type environments will be pointed out in a later section. Define the initial type environment $\gamma 0$ to be the list:

```
  <"Int", <↑Int, s(Int_eq_triv)>>
. <"False", <↑False, s(False_eq_triv)>>
. <"True", <↑True, s(True_eq_triv)>>
. nil
```

where for A a type, $\uparrow A$ is `<`A`, λx,y.  x=y in `A`>`.

The initial type environment is defined only as a convenience. It allows us to assume that certain associations are present in all contexts. This simplifies the statements of many theorems; for example, for the main evaluation function to have its desired type it is required that `"Int"` represent the set of integers. We also want to treat `"Prop"` in this way, but `Prop` is not a member of `Set` since `|Prop|` is `U6` and `Set` is `Set(6)`. We cannot lower the level of `|Prop|` since we will need the proposition a`=`b `in` S to be a member of `|Prop|` whenever S is in `Set`. Hence `"Prop"` is treated as a special case in some of the definitions below.

Note that our definition of type environments does not account for universe polymorphism. Instead of parameterizing with respect to universe level, the "types" (sets, actually) in type environments all come from a fixed level of the set hierarchy. This kind of simplification continues throughout the remainder of the library, and has the disadvantage that the reflection mechanism cannot be directly applied to itself. Thus, for example, we cannot always directly use term-rewriting functions in reasoning about the construction of other term-rewriting functions. This situation is alleviated somewhat by the fact that many of the theorem-proving procedures that we will want to write will operate on members of `Term0`, which is a member of `U1`.

If γ is a type environment and A is an atom, define `type_atom@(`γ`,`A`)` to be

```
A="Prop" in Atom
∨ ∃∈ γ0 : TEnvUnit list. (λu. A=u.1 in Atom)
∨ ∃∈ γ : TEnvUnit list. (λu. A=u.1 in Atom).
```

This is true if and only if A is `"Prop"` or is associated a value by $\gamma 0$ or γ. Using the tactic `Decidable` that was described in Chapter 3, we can prove that this proposition is decidable. Using the algorithm extracted from the proof, we can define a useful characterization `type_atom(`γ`,`A`)` of `type_atom@(`γ`,`A`)` that has the property that if γ and A are sufficiently concrete (*e.g.*, if they do not contain any free variables), it can be evaluated to either `True` or `False`.

Define

```
AtomicMType(γ)  ≡  { a: Atom | type_atom(γ,a) }
```

(`M` is for "meta"). Also, define

```
MType(γ)  ≡  { p: Atom list # Atom | all_type_atoms(γ,p) },
```

where `all_type_atoms` has a natural definition. This last type is the type of "meta-types" for functions; the first component of such a meta-type represents a function's argument types, and the second represents the result type.

The definition of application of a type environment to a member of `Atomic-MType(γ)` is straightforward. Define `γ(a)` to be

```
if a="Prop" then <Prop,fail>
else outl( γ0{TEnvVal}(a) ? γ{TEnvVal}(a) )
```

(where `outl` maps `s(a)` to a). Define `val(γ,a)` to be `γ(a).1`, the member of `SET` that is associated with a. Note that `Prop` is the value of `"Prop"`, even though `Prop` cannot appear in a type environment. An important property of any set S (`Prop` included) that is `val(γ,a)` for some a is expressed in the following.

```
∀γ:TEnv. ∀a:AtomicMType(γ). ∀x,y:|val(γ,a)|.
   (x=y in val(γ,a)) in |Prop|.
```

We can now define evaluation for members of `MType(γ)`. If *mt* is a member of `MType(γ)` such that *mt*`.1` is non-empty, then the value of *mt*'s "domain type" is

```
dom_val(γ,mt)  ≡
   let l,b = mt in
   #(map λa. val(γ,a) on l to SET list).
```

If *mt* is any member of `MType(γ)`, then its value in γ is

```
val(γ,mt)  ≡
   let l,b = mt in
   if null(l) then val(γ,b) else dom_val(γ,mt) -> val(γ,b).
```

Environments

Function environments associate an atom, called a *meta-function*, with a meta-type, with a value that is a member of the value of the meta-type, and with some additional information about the value. More specifically, for γ a type environment, define

```
FEnv(γ)  ≡
(  Atom # mt:MType(γ) # f:|val(γ,mt)| # val_kind(γ,mt,f) ) list.
```

The type val_kind(γ,mt,f) is a disjoint sum, and a member of it indicates whether f is an injection, a trivial predicate (in the sense of triv_eq as discussed above), a "semi-trivial" predicate, or none of the three. More will be said later about the use of this component. The initial function environment δ0 is

```
  <"True", <nil,"Prop">, True, no_kind>
. <"False", <nil,"Prop">, False, no_kind>
. nil.
```

The type whose members represent contexts is

$$\texttt{Env} \;\equiv\; \gamma\texttt{:TEnv \# FEnv(}\gamma\texttt{)}.$$

Members of Env will be called simply *environments*, and the variable α will always denote an environment. For most of the definitions presented so far that take a type environment as an argument, there is a new version which takes an environment as an argument. For example, instead of writing val(α.1,a) for the value of an atomic meta-type a in the type environment α.1, we will just write val(α,a).

Definitions analogous to those made for type environments are made for function environments. Thus define function-environment application, the predicates fun_atom@ and fun_atom, and the type MFun(α) of meta-functions. For f in MFun(α), mtype(α,f) is the meta-type assigned to f by α.2, and val(α,f) is the value assigned to f by α.2.

5.4 Well-Formedness and Evaluation

In this section we present the core of the reflection mechanism. We formalize a notion of well-formedness for members of Term0, and construct an evaluation function for well-formed terms.

Roughly, a meta-term t is well-formed in an environment α if "the meta-types match up". For example, suppose that t is a function-application $f(l)$, where l is a list of well-formed terms and f has a value in α. Let $A_1, \ldots, A_m$ be the members of the domain component of f's meta-type. Assume that each well-formed term has an atomic meta-type, and let $B_1, \ldots, B_n$ be the meta-types of the members of l. For $f(l)$ to be well-formed, we will require that $m = n$ and that for each i, A_i and B_i are appropriately related. It is much too restrictive to take the latter requirement to be equality as atoms.

To see this, consider two examples. First, suppose that during the course of a Nuprl proof we have a hypothetical function f declared to be of type Int->Int and a hypothetical x declared to be of type N (the type of natural numbers). If we construct an environment α where "f", "x", "N" and "Int" represent f, x, N and Int, respectively, and where "f" and "x" have the natural meta-types, then the representation "f"("x") of f(x) would not be well-formed in α. For the second example, suppose instead that f has type N->N, x has type Int, "f" and "x" have the natural meta-types, and that we know x$\geq$0. Again, "f"("x") would not be well-formed.

This leads us to consider a notion of matching of atomic meta-types which allows the associated values to stand in a subtype relationship. For simplicity, consider the case where f has domain meta-type the singleton list [A] and t has the meta-type B. Then f([t]) is well-formed in α if

- $A = B$, or
- val(α,B) is a subtype (in a sense to be made precise later) of val(α,A), or
- val(α,A) is a subtype of val(α,B) and the value of t in α satisfies the P such that

$$|\texttt{val}(\alpha,A)| = \{ \texttt{x: } |\texttt{val}(\alpha,B)| \mid P(\texttt{x}) \}.$$

Thus well-formedness depends on evaluation which in turn is only defined on well-formed terms. Therefore we need to deal with these notions in a mutually recursive fashion.

We will define the function which computes meta-types of terms, the evaluation function, and the well-formedness predicate, and then prove that they are well-defined (*i.e.* have the appropriate types) simultaneously by induction. Proceeding in this way relies on several particular properties of Nuprl's type theory. First, terms are untyped. For example, the domain type is not, as in other type theories, part of the notation for functions; that is, lambda abstractions are of the form $\lambda x.\, b$ and not $\lambda x : A.\, b$. This allows us to give direct expression to the evaluation function. Secondly, under some circumstances expressions a in A may be treated as propositions; more will be said about this shortly.

The meta-type of a term is simple to compute. Define mtype(α,t) to be

```
rec_ind(t; h,t. case t to AtomicMType(α)
                  f,args -> mtype(α,f).2
                  x,y,A -> "Prop"
                  i,x,y -> h(x)
                  n -> "Int").
```

This definition uses Nuprl's primitive form for induction over recursive types and a definition for case analysis of meta-terms. The part to the right of to is the result type of the case-analysis form. Informally the definition of mtype(α,t) can be written as

if t *is* f(args) *then* mtype(α,f).2
if t *is* x=y in A *then* "Prop"
if t *is* y{i x} *then* mtype(α,x)
if t *is* n *then* "Int".

The *type* of a meta-term is

$$\texttt{type}(\alpha,t) \equiv \texttt{val}(\alpha,\ \texttt{mtype}(\alpha,t)).$$

Note that this is actually a *set.*

To define evaluation, we need an auxiliary function that applies a function to a list and returns a tuple of the results. Define

$$g\{\alpha\}(l) \quad \equiv \quad (\texttt{[a]} \rightarrow g(\texttt{a});\ \lambda\ \texttt{h t v.}\ \texttt{<}g(\texttt{h}),\texttt{v>};\ \texttt{@}l),$$

where the term on the right is a form of list recursion where the base case is a singleton list. The value $\texttt{val}(\alpha,t)$ of a meta-term t is

```
rec_ind(t; g,t. case t to |type(α,t)|
                f,args -> if null(mtype(α,f).1) then val(α,f)
                          else val(α,f) (g{α}(args))
                x,y,A -> g(x)=g(y) in val(α,A)
                i,x,y -> g(x)
                n -> n).
```

We now proceed to formalize the notion of type matching for function application terms that was discussed earlier. The definition of *subtype* is based on Nuprl's subtype constructor, but since it applies to sets, we need to explicitly assert that the equalities are respected. So, if S_1 and S_2 are sets, then S_1 is a *subtype* of S_2, written $S_1 \subset\subset S_2$, if

```
∃P:|S₂->Prop| where
  |S₁| = {x:|S₂| | P(x)} in SET_U
  & ∀x,y:|S₁|. x=y in S₁ <=> x=y in S₂
```

(where SET_U is the universe component of SET). The type-matching described earlier involves comparing an atomic meta-type with a pair consisting of an atomic meta-type and the recursively computed value of an argument meta-term. It is convenient to define a type of such pairs:

$$\texttt{Val}(\alpha) \quad \equiv \quad \texttt{a:AtomicMType}(\alpha)\ \texttt{\#}\ |\texttt{val}(\alpha,\texttt{a})|.$$

We will call members of $\texttt{Val}(\alpha)$ simply *values.* A value v is a *member* of an atomic meta-type B, written $v \in\{\alpha\}\ B$, if

```
let A,a = v in
A=B in Atom ∨ val(α,A)⊂⊂val(α,B)
∨ ∃Q:val(α,B)⊂⊂val(α,A) where Q(a).
```

If *aml* is a list of atomic meta-types and *vl* is a list of values then define $vl \in\{\alpha\}\ aml$ by

```
|aml|=|vl| in N &
∀∈ com(vl,aml) : Val(α)#AtomicMType(α) list. (λ v,B. v ∈{α} B).
```

(Recall that `com` combines two lists of the same length into a list of pairs.)

With the definitions just given we can define the well-formedness of a function-application $f(l)$. Define `wf_fun_ap@(α,f,l)` by

```
fun_atom@(α,f) &
(map λt. <mtype(α,t), val(α,t)> on l to Val(α) list)
∈{α} mtype(α,f).1.
```

The well-formedness of the other kinds of meta-terms is straightforward to define. Well-formedness for an equality meta-term `x=y in A`, written `wf_eq_ap@(α,x,y,A)`, is

```
type_atom(α.1,A) & <mtype(α,x),val(α,x)> ∈{α} A
& <mtype(α,y),val(α,y)> ∈{α} A.
```

Well-formedness for an i-node $y\{i\ x\}$, written `wf_i_pair@(α,i,x,y)`, is defined by

```
fun_atom@(α,i) & is_injection(α,i) &
[mtype(α,x)] = mtype(α,i).1 in Atom list &
mtype(α,y) = mtype(α,i).2 in Atom &
val(α,i)(val(α,x)) = val(α,y) in val(α,mtype(α,y)).
```

Note that in this latter definition we have simplified matters by having a stricter matching requirement for the types of i, x and y.

We can now define the well-formedness of a meta-term t. Define `wf@(α,t)` to be

```
rec_ind(t; P,t.
  case t to U
    f,l -> (∀∈ l: Term0 list. P) & wf_fun_ap@(α,f,l)
    x,y,A -> P(x) & P(y) & wf_eq_ap@(α,x,y,A)
    i,x,y -> P(x) & P(y) & wf_i_pair@(α,i,x,y)
    n -> True)
```

where `U` is defined to be `U8`.

The central theorem of the reflection mechanism is the following:

```
∀α:Env. ∀t:Term0. wf@(α,t) in U &
     ↓(wf@(α,t)) => (mtype(α,t) in AtomicMType(α)
                     & val(α,t) in |type(α,t)|).
```

Finally, we define the type `Term(α)` of well-formed meta-terms as

```
{x:Term0 | wf@(α,x)}.
```

The proof of the main theorem is by induction on `t`, and illustrates a feature of Nuprl's type theory that distinguishes it from other type theories such as [25, 46]:

one can use induction to simultaneously prove the well-formedness and truth of a proposition. For example, to prove

$$\texttt{n:N >> } P[\texttt{n}]$$

it suffices to show that >> P[0] and

$$\texttt{n:N, } P[\texttt{n}] \texttt{ >> } P[\texttt{n+1}].$$

It is *not* also required that

$$\texttt{n:N >> } P[\texttt{n}] \texttt{ in U}i$$

for some i. If well-formedness were a prerequisite to the use of induction, our main theorem would not be provable.

For our partial-reflection method to be useable, the lifting procedure outlined in Chapter 1 must be reasonably fast. Since each lifting of a sequent will require establishing that the meta-terms representing the components of the sequent are well-formed, the well-formedness predicate `wf@` is not directly suitable. Proving `wf@(`α`,`t`)` for a typical concrete meta-term t and concrete environment α would require the analysis of a rather large proposition. However, many components of the proposition are decidable. For example, there are trivial algorithms for deciding the truth or falsity of the component

$$|aml| \texttt{ = } |vl| \texttt{ in N}$$

of the assertion $vl \in\{\alpha\}$ aml, and for the atom-equality case of the assertion $v \in\{\alpha\}$ B. In order to make it easier to prove well-formedness, we construct a characterization `wf` of `wf@` such that the simplification via the evaluation rules of instances `wf(`α`,`t`)` will remove these decidable components. For terms t, `wf(`α`,`t`)` will often simplify to `True` or `False`.

The characterization is built using operations over the "partial booleans". Define

$$\texttt{Bool} \equiv \texttt{True}\vee\texttt{True} \quad \text{and} \quad \texttt{PBool} \equiv \texttt{Bool}\vee\texttt{U}.$$

Thus a partial boolean is either a boolean or a proposition (that is, a type). For p a partial boolean, the proposition `true(`p`)` is true if p is the true boolean, or if p is a true member of U. Many of our proposition-forming operations have `PBool` analogues. For example, the conjunction $p_1 \& p_2$ of two partial booleans p_1 and p_2 is computed as follows (ignoring the injections for the disjoint unions). If p_1 and p_2 are propositions P_1 and P_2, then the result is the proposition $P_1 \& P_2$; if one is a proposition P and the other is a boolean b, then the result is P if b is true, and the boolean `false` if b is false; and if both are booleans then the result is their boolean conjunction. Thus, if `bool` and `prop` are the injections from `Bool` and `U`, respectively, into `PBool`, then the proposition

$$\texttt{true(bool(false) \& prop(}P\texttt{))}$$

simplifies to `False` regardless of what P is.

The definition of `wf` is straightforward. We simply replace the components of the definition of `wf@` by their `PBool` analogues, replacing by `prop(`P`)` those components P for which it is too difficult to get at the "decidable portions". The proof of the equivalence of `wf` and `wf@` is done almost automatically.

The "central theorem" given earlier establishes the partial reflection mechanism. In order to give some idea of what the formal arguments leading to this theorem are like, we will give a detailed account of one of the lemmas that was most difficult to prove. It is not necessary to understand this account in order to follow the rest of this chapter, so the remainder of this section may be skipped.

The lemma we will discuss is called `arg_tuple_typing` in Appendix C, and its statement is

```
∀α,P,wf,mtype,val. ∀mt:MType(α). ∀l:({t:Term0|P(t) & wf(t)} list).
   ¬(null(mt.1))
   => (map λt. <mtype(t),val(t)> on l to Val(α) list) ϵ{α} mt.1
   => ([a] → val(a); λ h t v. <val(h),v>; @ l) in |dom_val(α,mt)|
```

This statement uses a definition to suppress the types of the quantified variables α, ..., `val`. This lemma is applied once in the proof of the main theorem, to prove the function-application case. The P in the lemma statement is a predicate on `Term0`, and arises from the use of induction on `Term0` in the proof of the main theorem. (See Appendix B for a description of the induction rule for recursive types.)

The proof of the lemma has 44 steps. Snapshots of the steps are too bulky for presentation here, so we will give an informal proof and annotate it with references to the Nuprl proof. In order to make the presentation here cleaner, we will make a few definitions that have no analogues in the library. Suppose that $\alpha \in$ `Env`, `P` $\in$ `Term0->U`, and

`wf` $\in$ `{t:Term0|P(t)} -> U`.

Let W be

`{ t:Term0 | P(t) & wf(t) }`.

Suppose further that

`mtype` $\in$ W `-> AtomicMType(`α`)`

and

`val` $\in$ `t:`W `-> |val(`α`,mtype(t))|`.

For x in W `list`, let $vals[x]$ denote

`map λt. <mtype(t),val(t)> on` x `to Val(`α`) list`,

and for $x \in$ `AtomicMType(`α`) list`, let $vals[x]$ denote

`map λa. val(`α`,a) on` x `to SET list`.

Finally, for $x \in W$ list, let *tup*[x] denote

$$\texttt{([a]}\rightarrow\texttt{val(a);}\ \lambda\ \texttt{h t v. <val(h),v>;}\ \texttt{@}x\texttt{).}$$

We are now ready to give the proof of the lemma. In the argument that follows, comments within square brackets relate steps of the argument with the formal proof. When we refer to a "step" of the Nuprl proof, we mean a node of the proof tree; the refinement rule of the node is usually a combination (using tacticals) of several tactics. "Computation" as part of a step in a proof refers to the use of one or more of the computation tactics described in Chapter 3.

The proof is by induction on the list `mt.1`. [Expanding the definition of `dom_val` and setting up the induction are done in a single step] The base case is trivial [one step]. For the induction step, suppose that `aml` ∈ `AtomicMType(`α`) list`, `a` ∈ `AtomicMType(`α`)`, and `l` ∈ W `list`, and assume that

1) ∀`l:`W `list. ¬(null(aml)) =>`
vals`[l]` ∈ `aml => ` *tup*`[l] in |#(`*vals*`[aml])|`

and

2) *vals*`[l]` ∈`{`α`} (a.aml).`

We must show *tup*`[l] in |#(`*vals*`[a.aml])|`.

We do a case analysis on `l` [the analysis is set up in one step, using the tactic `Unroll`]. The case where `l` is `nil` follows from *2)* above. [This requires three steps. A contradiction is obtained from the component of *2)* which asserts that the lists have the same length. This requires some unfolding and folding of definitions.]

Assume, then, that `l` is `t.l'` [and, in one step, thin out some irrelevant hyptheses]. Suppose `null(aml)` [one step to do the case analysis on `aml`]. By *2)*, we may assume `l'` is `nil`. [This requires six steps. We `Unroll l'`, and prove the non-`nil` case by some low-level reasoning. This includes supplying a typing that `Autotactic` could not infer, unrolling the definitions of `map` and `length`, and applying a lemma about the length of empty lists.] We must show

`val(t) in |val(`α`,a)|.`

[Reducing the goal to this requires three steps. Some computation is done, `aml` is unrolled, the non-`nil` case of the unrolling is proved by expanding a definition, and finally more computation is done.] This follows from *2)* by a previous lemma [in one step we apply the lemma called `val_member_char` and use computation]. This finishes the case where `null(aml)`.

Suppose `¬null(aml)`. We can write `l'` as `t'.l''`. [Unroll `l'`, and prove the `nil` case by expanding *2)*, unrolling the definitions `map` and `length`, and using a lemma about the length of empty lists. This requires four steps.] We must show that

`<val(t), `*tup*`[t'.l'']> in |val(`α`,a)| # |#(`*vals*`[aml])|.`

[Assert a typing to aid the Autotactic, unroll map, and use a lemma to do a simplification of the term of the form |*A*#*B*| to |*A*|#|*B*|. This requires three steps.] We must show that the components of the pair are in the respective components of the product type [one step]. First,

$$\texttt{val(t) in |val(}\alpha\texttt{,a)|.}$$

follows from *2)*. [We use computation and the lemma val_member_char (two steps).] For the second component, by *1)* it suffices to show

$$\mathit{vals}[\texttt{t'.1''}] \in\{\alpha\} \texttt{ aml}$$

[one step to instantiate the induction hypothesis, one to prove a simple membership goal, and a computation step]. By definition it now suffices [two trivial steps] to prove that

$$|\mathit{vals}[\texttt{t'.1''}]| = |\texttt{aml}| \texttt{ in Int}$$

and

$$\forall\in \texttt{ com(}\mathit{vals}[\texttt{t'.1''}]\texttt{, aml) (}\lambda\texttt{p. p.1 }\in\{\alpha\}\texttt{ p.2),}$$

and both of these follow from *2)*. [To prove that the first of these follows from *2)* requires eight steps. The first step is computation. Six of the remainder are to prove three identical membership subgoals that Autotactic was unable to prove. These subgoals are $|\mathit{vals}[\texttt{1''}]|$ in Int and they are proved by folding and unfolding definitions. The remaining step proves $|\texttt{aml}|\geq 0$ using the tactic Properties. The proof that the second of the goals above follows from *2)* requires four steps, and uses computation together with several introductions and eliminations.]

5.5 Relations and Operations on Environments

Inference procedures proved correct with respect to a single environment would not be very useful. Usually the objects we reason about will contain variables and other components whose identity is not known in advance. For example, we will want a procedure for normalizing arithmetic expressions to apply to the conclusion of the sequent

```
1: Atom list >> |l|-|l| = 0 in Int.
```

What we desire, then, is that any procedure which is correct for meta-terms that are well-formed in some environment α is also correct in any extension of α. We will achieve this property by building it into the definition of correctness.

An environment α_1 is a *sub-environment* of α_2, written $\alpha_1 \subset \alpha_2$, if $\alpha_1.1$ and $\alpha_1.2$ are sub-a-lists of $\alpha_2.1$ and $\alpha_2.2$, respectively. Most of the functions defined so far that take an environment as an argument are *monotonic* with respect to environment extension. For example, we have

$$\forall\alpha 1\subset\alpha 2.\ \forall \texttt{t:Term0. wf@(}\alpha 1\texttt{,t) => wf@(}\alpha 2\texttt{,t)}$$

and

$$\forall \alpha 1 \subset \alpha 2.\ \forall \mathtt{t}:\mathtt{Term}(\alpha 1).\ \mathtt{val}(\alpha 1,\mathtt{t}) = \mathtt{val}(\alpha 2,\mathtt{t})\ \mathtt{in}\ |\mathtt{type}(\alpha 1,\mathtt{t})|.$$

A substantial portion of the library is devoted to theorems such as these.

We will often want to combine environments. Several different environments might be relevant to a particular sequent, and we will want to lift the sequent with respect to the union of these environments. If each of the individual environments is to be a sub-environment of the union, then they must agree on common atoms. Two a-lists l_1 and l_2 over a type A are *consistent*, written $\mathtt{cst}\{A\}(l_1,l_2)$, if

$$\forall \mathtt{a}:\mathtt{Atom}.\ \mathtt{succeeds}(l_1\{A\}(\mathtt{a})) \Rightarrow \mathtt{succeeds}(l_2\{A\}(\mathtt{a})) \Rightarrow l_1\{A\}(\mathtt{a}) = l_2\{A\}(\mathtt{a})\ \mathtt{in}\ ?A.$$

Environments α_1 and α_2 are consistent, written $\mathtt{cst}(\alpha_1,\alpha_2)$, if their respective type and function environments are consistent a-lists. Define $\alpha_1 @ \alpha_2$ to be the environment obtaining by appending the respective components. This operation is only defined when the two type environments are consistent. When $\alpha_1 @ \alpha_2$ is defined,

$$\alpha_1 \subset \alpha_1 @ \alpha_2 \quad \text{and} \quad \alpha_2 \subset \alpha_1 @ \alpha_2.$$

The above definitions have natural extensions to lists of environments. Such a list l is consistent, written $\mathtt{cst}(l)$, if its members are pairwise consistent. For consistent lists l, define $@(l)$ to be the result of appending all the environments in l. The most important property of $@(l)$ is that every member of l is a subenvironment of it.

Lifting a sequent will often generate a subgoal to prove that some list of environments is consistent. There are several theorems that help in dealing with these subgoals.

To show that two environments α_1 and α_2 are consistent we can apply the following theorem about a-lists:

```
∀A:U1. ∀.l1,l2:Atom#A list.
  ∀∈l2: Atom#A list. (λy. if s(x) = l1{A}(y.1). s=y.2 in A)
  => cst{A}(l1,l2).
```

(This statement differs from the Nuprl version in the expansion of a definition.) If α_1 and α_2 are concrete, then by applying the theorem to the respective type and function environment components and then normalizing the $(\forall\in\ \ldots)$ terms, showing α_1 and α_2 are consistent reduces to proving a conjunction of typechecking goals. In practice, however, the function environment components of different defined environments will usually not overlap. In such a case, we can use the above theorem to prove consistency of the type environments, and the following for the function environments:

```
∀A:U1. ∀.l1,l2:Atom#A list.
  ∀∈l1: Atom#A list. (λa. fails(l2{A}(a.1)))
  => cst{A}(l1,l2)
```

(where again a definition has been expanded). The sufficient condition for consistency given by this theorem, for concrete a-lists, will normalize to `True` (or `False`).

As discussed earlier, respect for environment extension will be built into the definition of correctness of inference procedures. A consequence of this is that we will often be interested in proving assertions of the form

$$\forall\alpha\texttt{:Env. } \alpha_0 \subset \alpha \texttt{ => P(}\alpha\texttt{)}$$

where α_0 is some concrete environment. A potential practical problem in proving such assertions is in reasoning about functions that take environments as arguments. Consider the following example. Suppose that "T" is an atom that is bound to a type T in α_0. Since α_0 is concrete, the term $\texttt{val(}\alpha_0\texttt{,"T")}$ can be simplified to T. In the context of a hypothesized α where $\alpha_0 \subset \alpha$, $\texttt{val(}\alpha\texttt{,"T")}$ cannot be simplified since α is a variable, although other kinds of reasoning can be used to deduce

$$\texttt{val(}\alpha\texttt{,"T") = } T \texttt{ in SET.}$$

These other kinds of reasoning are much harder than direct simplification. To avoid this problem we use a simple trick. For any α such that $\alpha_0 \subset \alpha$, both $\alpha \subset \alpha_0 @ \alpha$ and $\alpha_0 @ \alpha \subset \alpha$. Since our basic functions are monotonic with respect to $\subset$, for most P arising in practice, to prove

$$\forall\alpha\texttt{:Env. } \alpha_0 \subset \alpha \texttt{ => P(}\alpha\texttt{)}$$

it will suffice to show

$$\forall\alpha\texttt{:Env. P(}\alpha_0 @ \alpha\texttt{).}$$

For α a variable, terms like $\texttt{val(}\alpha_0 @ \alpha\texttt{,"T")}$ can still be simplified since the functions which look up values in environments use the natural procedure. For convenience, this trick is built into the definition of correctness for rewriting functions to be given later.

5.6 Lifting

A sequent

$$A_1\texttt{, } A_2\texttt{, ..., } A_n \texttt{ >> } A_{n+1}$$

is lifted by applying the tactic

`LiftUsing` *envs*

where *envs* is an ML list of Nuprl terms that are members of `Env`. Let α_1, α_2, ..., α_m be the members of the list. All but one of the subgoals generated by the tactic will usually be proved by the autotactic. The remaining subgoal is

$$\alpha\texttt{:Env, (...), val(}\alpha\texttt{,}A'_1\texttt{), val(}\alpha\texttt{,}A'_2\texttt{), ..., val(}\alpha\texttt{,}A'_n\texttt{)}$$
$$\texttt{>> val(}\alpha\texttt{,}A'_{n+1}\texttt{)}$$

where for each i, A'_i is a meta-term representing A_i, and where `(...)` is an instance of a special Nuprl definition that causes the system to suppress the display of the hypothesis. This elided hypothesis is a large conjunction that stores information about α and the meta-terms A'_i. In particular, it contains the assertion that α is equal to the combination of $\alpha_1, \alpha_2, \ldots, \alpha_m$ together with other bindings derived from the variables and other components of the unlifted sequent. It also contains assertions that each A'_i is a well-formed meta-term whose meta-type is `"Prop"`. It is easy to maintain the information in this hypothesis to reflect changes to the lifted sequent caused by the application of inference procedures. As a result, applications of inference procedures usually will not generate any subgoals to prove meta-terms well-formed. See Section 5.7 for more on this.

The preceding description of lifting contains several simplifications. First, the hypothesis list of the unlifted sequent may contain hypotheses that declare variables. The lifting procedure does not compute representations for such hypotheses. Secondly, the lifting procedure may fail to compute non-trivial representations for some of the A_i. Such A_i are left in the hypothesis list. Finally, the conclusion of the lifted sequent may be squashed; more will be said about this later. In the remainder of this section is a detailed account of the lifting procedure.

The first stage in the procedure is to compute the meta-terms A'_i and the necessary environment additions. This is done with a straightforward recursive algorithm. A meta-term t' representing a term t is computed as follows.

- If t is of the form f`(<`$a_1,\ldots,a_n$`>)` or f`(`a_1`)...(`a_n`)`, if $a'_1 \ldots a'_n$ are meta-terms representing $a_1 \ldots a_n$ respectively, and if some atom f' is bound to f in the function environment component of some α_j, then t' is

  $$f'(\texttt{[}a'_1\texttt{;}\ldots\texttt{;}a'_n\texttt{]})$$

  (using our notations for injections into `Term0`).

- If t is a`=`b `in` S for some set S, if a' and b' represent a and b, and if an atom S' is bound to S in the type environment component of some α_j, then t' is a'`=`b' `in` S'.

- If t is an integer, then t' is the injection of that integer into `Term0`.

- If none of the above apply, then generate a new atom a and an environment addition that associates a with t. The atom a is chosen to appear textually similar to t; for example, the atom chosen for the term `x+y` would be `"x+y"`.

The lifting procedure currently does not account for i-nodes. If only the last clause is applicable to a top-level term, the procedure fails. Through the use of Nuprl definitions, the terms A'_i can be made to appear to the user as identical to the terms A_i. An example of this is given in Chapter 1. One of the definitions takes advantage of the fact that definitions need not respect term structure; using it, a token `"x"` appears as just `x`.

The successful applications of the procedure just described to the terms $A_1, \ldots,$ A_{n+1} give a list of environment additions and representatives for some of the A_i. The α_i and the environment additions are assembled into a single environment $\overline{\alpha}$, and a Nuprl variable α is created. Next, hypotheses are added concerning α, $\overline{\alpha}$ and the computed representives. ("Adding" hypotheses, of course, generates proof obligations.) The first hypothesis added asserts that the list *envs* is consistent. This fact is usually proved by a special tactic which is programmed to make use of the techniques mentioned in the previous section. Next, a declaration α:`Env` and a hypothesis

$$\alpha \texttt{ = } \overline{\alpha} \texttt{ in Env}$$

are added. Adding the equality requires a proof that $\overline{\alpha}$ is in `Env`; this will usually be done automatically. The next hypothesis to be added asserts that each α_i is a subenvironment of α; this will always be proved automatically.

For each computed meta-term t, the following hypotheses are added:

$$t \texttt{ in Term0,} \quad \texttt{wf(}\alpha\texttt{,}t\texttt{),} \quad \texttt{mtype(}\alpha\texttt{,}t\texttt{) = "Prop" in Atom.}$$

The first of these will always be proved automatically. For the other two, lemmas are first used to substitute $\overline{\alpha}$ for α. For example, for `wf(`α`,`t`)` we use the lemma

$$\forall\alpha\texttt{1,}\alpha\texttt{2:Env. }\forall\texttt{t:Term0. }\alpha\texttt{1=}\alpha\texttt{2 in Env => wf(}\alpha\texttt{2,t) => wf(}\alpha\texttt{1,t).}$$

Each term `wf(`$\overline{\alpha}$`,`t`)` is simplified with the evaluation rule. The evaluation is controlled by specifying that some definitions, including those that occur in $\overline{\alpha}$, not be expanded. The simplification may result in subgoals whose proof requires user assistance; see Section 5.4 for more on this. To prove the assertion about `mtype` simple evaluation is used; if `mtype(`$\overline{\alpha}$`,`t`)` does not evaluate to the atom `"Prop"`, then the lifting procedure fails.

The next stage of the lifting procedure is to replace the terms A_i for which meta-terms A_i' were successfully computed by `val(`α`,`A_i'`)`. This is accomplished using two lemmas: one for the A_i that are hypotheses, and one for the conclusion. The one for hypotheses is

$$\begin{array}{l}\forall\alpha\texttt{1,}\alpha\texttt{2:Env. }\forall\texttt{t:Term0. }\alpha\texttt{1=}\alpha\texttt{2 in Env => wf(}\alpha\texttt{1,t) =>}\\ \quad\texttt{mtype(}\alpha\texttt{1,t) = "Prop" in Atom => val(}\alpha\texttt{1,t) => val(}\alpha\texttt{2,t).}\end{array}$$

Using the lemma, to justify the replacement of the hypothesis A_i by `val(`α`,`A_i'`)` it suffices to show `val(`$\overline{\alpha}$`,`A_i'`)`. The evaluation rule can be used to compute this term to A_i.

The next step is to clean up the hypothesis list. This involves conjoining all the new hypotheses except for those of the form `val(`α`,`t`)`, and eliding the resulting conjunction (that is using a definition that causes it to be displayed as `(...)`).

The final step is to attempt to squash the conclusion. Many of the operations we will want to perform on lifted sequents will involve equality reasoning, and we have chosen to regard equalities as having no computational significance. Thus a value-preserving transformation of A'_{n+1} to A''_{n+1} will only allow us to conclude in general that

$$\downarrow(\mathtt{val}(\alpha,A'_{n+1}))\ \mathtt{<=>}\ \downarrow(\mathtt{val}(\alpha,A''_{n+1})).$$

Since $\mathtt{val}(\alpha,A'_{n+1})$ does not follow from $\downarrow(\mathtt{val}(\alpha,A'_{n+1}))$, such a transformation cannot be applied if the conclusion of the sequent is just $\mathtt{val}(\alpha,A'_{n+1})$.

To replace $\mathtt{val}(\alpha,A'_{n+1})$ by $\downarrow(\mathtt{val}(\alpha,A'_{n+1}))$, the lifting procedure attempts to use information from the environment. Suppose that A'_{n+1} is an equality meta-term u=v in A. The value associated with A in the type environment component of $\overline{\alpha}$ is a pair $\mathtt{<}S\mathtt{,}p\mathtt{>}$ where $p \in$?triv_eq(S). If p is of the form s(p') then we know that

$$p' \ \in\ \forall\mathtt{x,y:}|S|\mathtt{.}\ \downarrow(\mathtt{x=y\ in\ }S)\ \mathtt{=>}\ \mathtt{x=y\ in\ }S$$

and we can use this fact to squash the conclusion (actually applying this fact involves the use of several lemmas). If p is fail, then the attempt to squash fails. If A'_{n+1} is a function application, then an analogous attempt is made using the val_kind component of function environments (see Section 5.3).

5.7 Term Rewriting

The remainder of the library is devoted to applying the partial reflection mechanism to equality reasoning. In this section we discuss definitions and theorems that constitute an implementation of a term rewriting system. This system supports the construction of value-preserving functions on meta-terms. Such functions can be based directly on object-level equations, or can be defined in terms of other rewriting functions by using a set of combinators. In practice, the verification of the correctness of particular rewriting functions will often involve only simple typechecking.

The Type of Rewriting Functions

A function f of type Term0->?Term0 is a rewrite with respect to an environment α if for any term t that is well-formed in some extension α' of α, the computation of the application $f(t)$ either fails or results in a term t' such that:

- t' is well-formed in α',
- t' and t have the same meta-type, say A, and
- the values of t and t' are equal in the set that is the value of A.

To define this formally, we first define equality of values for terms t and t' that are well-formed in an environment α. Define t =$\{\alpha\}$ t' if

```
mtype(α,t1) = mtype(α,t2) in Atom
& val(α,t1)=val(α,t2) in type(α,t1).
```

If f is a function of type `Term0->?Term0`, then f is *value invariant*, written `val_inv(`α`,`f`)`, if

```
∀t1:Term0 where wf@(α,t1).
  if s(t2)=f(t1) . ↓(wf@(α,t2)) & ↓(val(t1) ={α} val(t2)).
```

We use the squash operator in this definition because we do not want to require the existence of a function that computes proofs of well-formedness and value-invariance. Finally, define `Rewrite(`α`)`, the type of rewriting functions that are correct in the environment α, as

```
{f: Term0->?Term0 | ∀α2:Env where cst(α,α2). val_inv(α@α2,f)}.
```

Note that this definition incorporates the convenient characterization of environment extension that was discussed in Section 5.5. A crucial fact about rewrites is the following.

```
∀α1,α2:Env. α1⊂α2 => ∀f:Rewrite(α1). f in Rewrite(α2).
```

Basic Rewriting Functions

The basic units of conventional term rewriting are *rewrite rules.* These are ordered pairs $\langle t, t'\rangle$ of terms containing variables, where all the variables of t' occur in t. A term u can be rewritten to a term u' via the rule $\langle t, t'\rangle$ if there is a substitution σ for the free variables of t such that u' is obtained by replacing an occurrence of $\sigma(t)$ in u by $\sigma(t')$. We will view a rewrite rule as a member of `Rewrite(`α`)`. Corresponding to $\langle t, t'\rangle$ will be the function which, given u, if there is a substitution σ such that $\sigma(t) = u$, returns $\sigma(t')$, or else fails. Such functions will usually be obtained by "lifting" a universally quantified equation.

Since rewriting is based on term-matching, we need to develop a theory of substitution for meta-terms. For this we need to consider meta-terms with variables. In order to avoid introducing a new type for terms with variables, we will take variables to be meta-terms of the form v`(nil)` for v an atom. Not all such terms will be variables; concepts that concern variables will usually be parameterized by a list *vars* of atoms, and v`(nil)` will be a variable just if $v \in$ *vars.* When v`(nil)` is a variable, we will refer to it as the *variable* v.

A variable v *occurs* in a meta-term t if

```
rec_ind(t; Q,t.
case t to U1
  f,l -> (null(l) & v=f in Atom) ∨ ∃∈ l : Term0 list. Q
  x,y,A -> Q(x) ∨ Q(y)
  i,x,y -> Q(x) ∨ Q(y)
  n -> False).
```

A *substitution* is a member of the type

```
(Atom # Term0) list.
```

For the purpose of applying a substitution s to a term t, the atoms of s are taken to define the variables of t. The function `subst` takes a substitution s and a meta-term t and simultaneously replaces subterms v`(nil)` by u for each `<`v`,`u`>` in s. Applications `subst(`s`,`t`)` will be written s`(`t`)`. Suppose *vars* is a list of atoms. A substitution s is *limited* to *vars*, written

s `limited to` *vars*,

if for every member `<`x`,`u`>` of s, x is a member of *vars*. If t is a term, then s *covers* t with respect to *vars*, or

s `covers` t `wrt` *vars*,

if for every member x of *vars* that occurs as a variable in t, x is the first component of some pair in s. Finally, s is *complete* on t with respect to *vars* if it covers t and is limited to *vars*. The significance of completeness is that it is required for monotonicity of substitution: for substitutions s_1 and s_2, if s_1 is a sub-a-list of s_2, and both s_1 and s_2 are complete on t with respect to *vars*, then

s_1`(`t`)=`s_2`(`t`) in Term0.`

This property is need for the matching function, which puts together recursively computed substitutions.

The function `match` is defined in terms of the program extracted from the proof of the following theorem.

```
∀vars:Atom list. ∀t1,t2:Term0. ?(∃s:Atom#Term0 list
   where (s{vars} complete on t1) & s(t1)=t2 in Term0).
```

The proof of this theorem is rather typical of proofs in the "applications" portion of the library, and so we will briefly discuss it here. The first step is to do induction on `t1`. This gives four subgoals corresponding to four kinds of meta-terms. We will only discuss the hardest of the subproofs, which is the one corresponding to function application. The 45-step subproof is given in its entirety in Appendix C.

The goal of the subproof is to show

```
?{ s | s complete on f(l) and s(f(l)) = t2 }
```

where it is assumed that for any `x` in the list `l` and any `t` in `Term0`

```
?{ s | s complete on x and s(x) = t }.
```

(We are taking some liberties with our notation here.) Applying the tactic

```
(Decide 'null(l)' ...)
```

gives two subgoals, one where we assume null(l) and the other where we assume ¬(null(l)). In the first case, we apply

```
(Decide '∃∈ vars: Atom list λx. f=x in Atom' ...).
```

This again gives two subgoals. In the first of these, we know that f(l) is a variable, so we introduce the list [<f,t2>] for s. This gives two subgoals:

```
[<f,t2>]{vars} complete on f(l)
```

and

```
[<f,t2>](f(l)) = t2 in Term0.
```

The first is proved in six steps by computation and unrolling l, and the second in five steps by computation, unrolling l and a substitution.

In the case where null(l) but f is not in vars, f(l) is not a variable and has no sub-meta-terms. Here we first unroll t2 using

```
(OnVar `t2` Term0Unroll ...).
```

This gives four subgoals. Only the case where t2 is a function application is non-trivial. The other three cases are proved in one step each by introducing a failure via

```
(IRight ...)
```

(recall that ?A is A|True). Thus assume that t2 is f1(l1). We now apply

```
(Decide 'f=f1 in Atom & null(l1)' ...)
```

and fail in the negative case. In the remaining case, f(l) and t2 are the same, so we can introduce the empty substitution nil for s. This produces two subgoals:

```
nil{vars} complete on f(l)
```

and

```
nil(f(l)) = inl(<f1,l1>) in Term0.
```

The first is proved in six steps using computation, unrolling l, and a substitution. The second is proved in five steps by unrolling l and l1 and using computation.

Finally, we have the case where ¬null(l). We unroll t2, failing on the non-function-application cases. We also assume that f=f1 (else failure). By applying the lemma list_subst_lemma to the induction hypothesis we get an s such that s is complete on each t in l and such that

```
(map subst(s) on l to Term0 list) = l1 in Term0 list.
```

Our subproof is completed in four more steps by using computation, some explicit equality introductions, and the lemmas complete_subst_on_fun_ap and subst-_on_fun_ap which relate substitution to function applications.

Our representation of a rewrite rule is defined directly in terms of `match`. If u and v are meta-terms and if *vars* is a list of atoms then define `rewrite{`*vars*`}(`u`->`v`)` to be

```
λt. let s(subst) = match(u,t,vars) in s(subst(v)): ?Term0.
```

This is a function that matches u against a given term t and either fails or applies the resulting substitution to v.

To make simpler the proofs of correctness of instances of this rewrite constructor, we introduce the notion of a *full* substitution. A substitution s is *full* over *vars* if

```
∃l:Term0 list where |vars|=|l| in Int &
        s = com(vars,l) in Atom#Term0 list.
```

This allows us to express a useful sufficient condition:

```
rewrite{vars}(u->v) in Rewrite(α1)
```

if for every α_2 such that `cst(`α_1`,`α_2`)`,

```
∀s:Atom#Term0 list. s full over vars => wf@(α1@α2,s(u)) =>
    ↓(wf@(α1@α2,s(v))) & ↓(val(s(u)) ={α1@α2} val(s(v)))).
```

This theorem does not get applied directly. Instead, we use specializations of it where the number of variables is fixed. The length two specialization is the following. For atoms x and y,

```
rewrite{[x;y]}(u->v) in Rewrite(α1)
```

if for every α_2 such that `cst(`α_1`,`α_2`)` and for all terms t_1 and t_2,

```
let s = [<x,t1>;<y,t2>] in
wf@(α1@α2,s(u)) =>
↓(wf@(α1@α2,s(v))) & ↓(val(s(u)) ={α1@α2} val(s(v))).
```

An equational fact can be turned into a verified rewriting function by using the tactic `LemmaToRewrite`, which makes use of these specialized theorems. We describe the operation of the tactic by considering the one-variable case. An invocation of the tactic in such a case has the form:

```
>> rewrite{[x]}(u->v) in Rewrite(α1),
BY LemmaToRewrite T.
```

The purpose of the tactic argument T will be described shortly. The tactic first applies the appropriate specialized theorem and does some simplifications. This gives two subgoals (all other entailed proof obligations will be proved by the autotactic). The first of these is

$$\alpha 2\texttt{:Env, t:Term0, wf@(}\alpha_1\texttt{@}\alpha 2\texttt{,}u'\texttt{) >> wf@(}\alpha_1\texttt{@}\alpha 2\texttt{,}v'\texttt{),}$$

where u' and v' are u and v with the all occurrences of x(nil) replaced by t. Simplifying the hypothesis wf@(α_1@α2,u') yields a hypothesis wf@(α_1@α2,t) and hypotheses t $\in\{\alpha_1 @ \alpha 2\}$ A for some atomic meta-type A. In practice, these hypotheses will often be sufficient to allow the autotactic to completely prove the simplification of the conclusion. The second subgoal is

$$\alpha 2\texttt{:Env, t:Term0, wf@(}\alpha_1\texttt{@}\alpha 2\texttt{,}u'\texttt{) >> }\downarrow\texttt{(}u'\texttt{ =\{}\alpha_1\texttt{@}\alpha 2\texttt{\} }v'\texttt{).}$$

The first step in proving this is to simplify both the hypothesis wf@(α_1@α2,u') and the conclusion. The proof that u' and v' have the same meta-types (if they in fact do) will always be done automatically, so what remains to be shown is that an object-level equality holds. For example, if u is the meta-term x-x, α_1 is α_Q (an environment for rational arithmetic) and v is 0, then u' is t-t, and the simplified form of the remaining subgoal has conclusion

$$\texttt{val(}\alpha\texttt{\_Q@}\alpha 2\texttt{,t) - val(}\alpha\texttt{\_Q@}\alpha 2\texttt{,t) = 0 in Q.}$$

The last step of LemmaToRewrite is to apply the tactic T to this subgoal. Before applying it, though, a lemma is applied to add, for each hypothesis t $\in\{\alpha_1 @ \alpha 2\}$ A, a new hypothesis

$$\texttt{val(}\alpha_1\texttt{@}\alpha 2\texttt{,t) in |val(}\alpha_1\texttt{@}\alpha 2\texttt{,}A\texttt{)|.}$$

Thus, if T simply applies an equational lemma, then the autotactic can complete the proof.

There are two simple basic rewrites that are used frequently. The first is the trivial rewrite

$$\texttt{Id} \equiv \lambda\texttt{t. s(t).}$$

The second is true_eq, which rewrites an equality meta-term a=b in A to True when a and b are "syntactically" equal. This notion of equality is the same as identity, (*i.e.*, equality in the type Term0), except when i-nodes are involved. Recall that the value of an i-node $u\{i\ \ v\}$ is defined to be the value of v. We therefore take $u\{i\ \ v\}$ to be syntactically equal to $u'\{i\ \ v'\}$ when v is syntactically equal to v'.

Building Other Rewriting Functions

The approach to term rewriting of Paulson [53] is easily adapted to our setting. His approach involves using analogues of the LCF *tacticals* [30] as combinators for building rewrites, where his rewrites produce LCF proofs of equalities. Using these combinators, we can concisely express a variety of rewriting strategies (Paulson used his rewriting package to prove the correctness of a unification algorithm [53]). Furthermore, since we can prove that each combinator is correct (*i.e.*, it combines members `Rewrite`(α) into a member of `Rewrite`(α)), proving that a complex rewriting function is correct often involves only simple typechecking.

If f and g are in `Rewrite`(α), then so are the following.

- f `THEN` g. This rewriting function, when applied to a term t, fails if $f(t)$ fails, otherwise its value is $g(f(t))$.
- f `ORELSE` g. When applied to t, this has value $f(t)$ if $f(t)$ succeeds, otherwise it has value $g(t)$.
- `Progress` f. This fails if $f(t)$ succeeds and is equal (as a member of `Term0`) to t, otherwise it has value $f(t)$.
- `Try` f. If $f(t)$ fails then the value is t, otherwise it is $f(t)$.
- `Sub` f. This applies f to the immediate subterms of t, failing if f failed on any of the subterms.
- `Repeat` f. This is defined below.

In Nuprl, all well-typed functions are total. However, many of the procedures, rewrites in particular, that we will want to write will naturally involve unrestricted recursion, and in such cases we will not be interested in proving termination. For example, many theorem-proving procedures perform some kind of search, and searching continues as long as some criteria for progress are satisfied. In such a case, it will often be difficult or impossible to find a tractable condition on the inputs to the procedure that will guarantee termination. It appears that the effort to incorporate partial functions into the Nuprl type theory has been successful [22], but a final set of rules has not yet been formulated and implemented.

For our purposes, there is a simple-minded solution to this problem which should work in all cases of practical interest. We simply use integer recursion with a (very) large integer. Thus, for A a type, a a member of A, and f a function of type A->A, we define `fix{`A`}(`a`,`f`)`, an approximation to the fixed-point of f, as

$$\texttt{[0} \longrightarrow a \texttt{ ; n,y} \longrightarrow f\texttt{(y) ; @ a\_big\_number].}$$

One difference between this and a true fixed-point operator is that we must take into account the effectively bogus base case a. Using this definition it is easy to define `Repeat`:

$$\texttt{Repeat(}f\texttt{)} \;\equiv\; \texttt{fix\{Term0->?Term0\}(Id, }\lambda\texttt{g. (f THEN g) ORELSE Id).}$$

We can also directly define, using `Id` as the "base case", a fixed-point operator `rewrite_letrec` for rewriting functions that satisfies

$$\begin{array}{l}\forall\alpha\texttt{:Env. }\forall\texttt{F:Rewrite(}\alpha\texttt{)->Rewrite(}\alpha\texttt{).}\\ \qquad\texttt{rewrite\_letrec(F) in Rewrite(}\alpha\texttt{).}\end{array}$$

As an example of the use of our combinators, we define a rewrite that does a bottom-up rewriting using f:

$$\texttt{BotUp}(f) \;\equiv\; \texttt{letrec g = (Sub(g) THEN Try(}f\texttt{))}.$$

Normalization with respect to f could be accomplished with

$$\texttt{Repeat (Progress (BotUp(}f\texttt{)))}.$$

Applying a Rewriting Function

In Chapter 1 we gave an example of an application of a rewriting function to a lifted sequent. The application was done with a tactic `RewriteConcl` which takes as an argument the rewriting function to be applied. Below we describe how `RewriteConcl` works. The analogous tactic `RewriteHyp` is very similar.

Assume that we have a lifted sequent as described in Section 5.6. The application of (`RewriteConcl '`f`' ...`) to the sequent, where f is a Nuprl term, proceeds as follows. First, if the conclusion is not already squashed, an attempt is made to squash it in the same way as described in Section 5.6. If the conclusion cannot be squashed, then rewriting cannot proceed. Thus assume that the conclusion is $\downarrow$`val(`α`,`t`)`. Next, the following lemma is instantiated with α, $f[\overline{\alpha}/\alpha]$ and the meta-term t.

```
∀α:Env. ∀f:Rewrite(α). ∀x:Term0.
  ↓(wf(α,x)) => mtype(α,x) = "Prop" in Atom =>
  if s(y)=f(x) . y in Term0 & ↓(wf(α,y)) &
    mtype(α,y) = "Prop" in Atom & ↓(val(α,x)) <=> ↓(val(α,y)).
```

Proving that α and t have the appropriate type and that the two antecedants in the implication hold is trivial since all these properties except the first are conjuncts of the elided hypothesis of the lifted sequent. The subgoal to show that $f[\overline{\alpha}/\alpha]$ is in `Rewrite(`α`)` is handled by a special purpose tactic that will not always return a complete proof.

The remaining subgoal has conclusion $\downarrow$`val(`α`,`t`)` and a hypothesis of the form "`if s(y)=f(t). ...`". Evaluation is applied to this hypothesis to compute f`(`t`)` as far as possible. If f is well-typed, then this evaluation will result in f`(`t`)` being replaced either by `s(`t'`)` for some meta-term t' (the rewriting of t), or by `fail`. In the latter case, `RewriteConcl` fails, so assume that the former holds.

Simplifying and restoring some definitions, we have a new meta-term t' and new hypotheses t' `in Term0`, $\downarrow$`wf(`α`,`t'`)`,

$$\texttt{mtype(}\alpha\texttt{,}t'\texttt{) = "Prop" in Atom,}$$

and

$$\downarrow\texttt{(val(}\alpha\texttt{,}t\texttt{)) <=> }\downarrow\texttt{(val(}\alpha\texttt{,}t'\texttt{))}$$

The last of these allows us to replace the conclusion with `↓(val(`α`,`t`))`. The rest of the new hypotheses are added to the elided hypothesis (and the components of the elided hypothesis relating to t are removed). This completes the rewriting.

5.8 Other Applications

In this section we describe two smaller applications of the partial-reflection mechanism.

A Procedure for Equality

Rewriting functions deal with individual terms. We have implemented a simple procedure that applies to representations of sequents. This procedure decides whether an equality follows from hypothesized equalities by symmetry, transitivity and reflexivity. The procedure is a member of the type `Complete(`α`)` defined as

`{t:PropTerm(`α`)|t{`α`}} list -> concl:PropTerm(`α`) -> ?↓(concl{`α`})`

where `PropTerm(`α`)` is

`{ t:Term(`α`) | mtype(`α`,t)="Prop" in Atom }`

and where `t{`α`}`, for t in `PropTerm(`α`)`, is defined as `val(`α`,`t`)`. Members of `Complete(`α`)` are applied using the tactic `ApplyCompleteTac`.

For the purpose of term-rewriting it was convenient to take equality of value for meta-terms to include identity of meta-types. Here we relax this constraint, and define t`=`t' `∈{`α`}` A, for t and t' in `Term(`α`)`, by

t `∈{`α`}` A `&` t' `∈{`α`}` A `& val(`α`,`t`) = val(`α`,`t'`) in val(`α`,`A`)`.

The equality procedure is extracted from the following theorem.

α`:Env -> Complete(`α`)`.

We will briefly discuss the proof of this theorem. The main step in the proof is shown in Figure 5.1. In this step we are assuming that `concl` is an equality meta-term `x=y in A`, and we must construct a program that succeeds only if the equality is true. To do this, we specify a subprogram that takes a list of pairs of meta-terms with equal values in `A`, and a meta-term `t`, and succeeds only if `t` and `y` have the same value. To use this subprogram to prove the goal, we apply it to the list obtained from the `A`-equalities that are in the list `hyps`, and to `x`. Proving that this application of the subprogram suffices requires 23 steps.

The main step in constructing the subprogram is to apply the following fixed-point induction lemma (which is simply the theorem giving the type of the approximate fixed-point operator defined in the previous section).

`A:Type -> A -> (A->A) -> A`

```
* top 1 1 3
1. α: Env
2. hyps: {t:PropTerm(α)|t{α}} list
3. concl: PropTerm(α)
4. x: Term(α)
5. y: Term(α)
6. A: Atom
[7]. wf@(α,x=y in A) & concl=(x=y in A) in Term(α)
[8].  A in AtomicMType(α) & y ϵ{α} A
>> ?↓(concl{α})

BY Assert
   '∀l: { q:Term(α)#Term(α) | q.1 = q.2 ϵ{α} A } list.
      ∀t:Term(α) where t ϵ{α} A. ?↓(t = y ϵ{α} A)'


1* >> ∀l: { q:Term(α)#Term(α) | q.1 = q.2 ϵ{α} A } list.
        ∀t:Term(α) where t ϵ{α} A. ?↓(t = y ϵ{α} A)

2* 9. ∀l: { q:Term(α)#Term(α) | q.1 = q.2 ϵ{α} A } list.
        ∀t:Term(α) where t ϵ{α} A. ?↓(t = y ϵ{α} A)
   >> ?↓(concl{α})
```

Figure 5.1: A step from the development of the equality procedure.

One of the subgoals resulting from the application of the induction lemma is identical to the goal; this is the "bogus base case" which we prove by simply introducing a failure. The other subgoal has the form

$$\ldots\ P \gg P$$

where P is the specification of the subprogram. The proof of this requires 25 steps. An outline of this proof follows.

Assume that we are given an `l` and a `t`. The first step is to check whether `t` is identical to `y`. If not, then we find a pair in `l` that contains `t`, failing if none exists. Let `t2` be the other component of the pair, and let `l2` be `l` with the pair removed. We first apply the induction hypothesis (P) to `l2` and `t2`, and then do a case analysis on whether the result fails or not. If not, we are done. If it does, then we instantiate the induction hypothesis on `l2` and `t`. If the result fails, then we prove

$$?\downarrow(\mathtt{t}=\mathtt{y}\ \in\{\alpha\}\ \mathtt{A}$$

by introducing a failure; otherwise, we are done.

The proof of this theorem illustrates a new aspect taken on by the "proofs-as-programs" paradigm [7] when the program being constructed is *partial* (that is, possibly either failing or effectively not terminating). In this case the sequent no longer captures the specification of the program. For example, in the subproof just described, it was necessary during its construction in Nuprl for the user to remember that P could only be applied to "smaller" inputs. This proof turned out to be surprisingly difficult. The number of requirements not specified by the sequents proliferated as the proof was developed, and it became difficult to prevent the tactics from automatically applying hypotheses incorrectly (sometimes in subtle ways). The development of partial programs in Nuprl is an area requiring much more exploration.

An Expression Normalizer

A *monoid* is a set together with an associative binary operation and an identity element. A *commutative* monoid is a monoid where the binary operation is commutative. We have implemented a term-rewriting function which normalizes expressions over a commutative monoid. The normal form of an expression is right-associated, has no occurrences of the identity element (unless it is the whole expression), and all of its maximal subterms that are not applications of the monoid's binary operator are sorted from left to right according to a lexicographic ordering. In the remainder of this subsection we discuss the main definitions and theorems in the development of the normalizing algorithm.

Monoids are straightforward to define. If S is a set, e is a member of $|S|$ and o is a member of $|S\#S\text{->}S|$, then define `monoid(`S`,`e`,`o`)` by

$$\forall \mathtt{x},\mathtt{y},\mathtt{z}\!:\!|S|.\ \mathtt{x}\ o\ (\mathtt{y}\ o\ \mathtt{z}) = (\mathtt{x}\ o\ \mathtt{y})\ o\ \mathtt{z}\ \mathtt{in}\ S\ \&\ \forall \mathtt{x}\!:\!|S|.$$
$$\mathtt{x}\ o\ e = \mathtt{x}\ \mathtt{in}\ S\ \&\ e\ o\ \mathtt{x} = \mathtt{x}\ \mathtt{in}\ S.$$

This definition uses a Nuprl definition that allows function application to appear as infix. Our definition of commutative monoid is the obvious one.

Atoms A, e and o represent the components of a monoid in an environment α if their values have the appropriate type and satisfy the monoid axioms. More precisely, o represents a binary operation in α, or o `bin op over` $A\{\alpha\}$, if

$$\texttt{type\_atom}(\alpha.1,A)\ \texttt{\&}\ \texttt{fun\_atom}(\alpha,o)\ \texttt{\&}\ \texttt{mtype}(\alpha,o)\ \texttt{=}\ \texttt{A\#A->A in MType}(\alpha)$$

Here we have used a definition for meta-type constants. A, e and o represent a monoid, or `monoid(`α`,`A`,`e`,`o`)`, if

$$\begin{aligned}&e\ \in\{\alpha\}\ A\ \ \texttt{\&}\ \ o\ \texttt{bin op over}\ A\{\alpha\}\\ &\texttt{\& monoid(val(}\alpha,A\texttt{), val(}\alpha,e\texttt{), val(}\alpha,o\texttt{)).}\end{aligned}$$

We have the following characterization property (using the definition of value-equality for meta-terms given in the last subsection).

$$\begin{aligned}&\forall\alpha\texttt{:Env. }\forall A,e,o\texttt{:Atom. monoid(}\alpha,A,e,o\texttt{)}\\ &\ \texttt{=>}\ \forall\texttt{x,y,z:Term(}\alpha\texttt{). x }\in\{\alpha\}\texttt{ A \& y }\in\{\alpha\}\texttt{ A \& z }\in\{\alpha\}\texttt{ A =>}\\ &\quad\texttt{x o (y o z) = (x o y) o z }\ \in\{\alpha\}\ \texttt{A}\\ &\quad\texttt{\& }\forall\texttt{x:Term(}\alpha\texttt{). x }\in\{\alpha\}\texttt{ A =>}\\ &\qquad\texttt{x o e = x }\ \in\{\alpha\}\ \texttt{A \& e o x = x }\ \in\{\alpha\}\ \texttt{A}\end{aligned}$$

A Nuprl definition is used here to allow function applications in meta-terms to appear as infix.

The first step in the normalization algorithm is to compute a list of meta-terms representing the meta-term to be normalized. In order to specify this part of the algorithm, we need to define an auxiliary function that combines a list of meta-terms into a single one. Thus for l a list of meta-terms, define `(implode` l `using` e`,`o`)` by

$$\texttt{[nil }\rightarrow\ e\texttt{; h.t,v }\rightarrow\texttt{ h o v; @}l\texttt{].}$$

This is a member of `Term(`α`)` whenever $e\ \in\{\alpha\}\ A$, o `bin op over` A and for all members t of l, t is well-formed and $t\ \in\{\alpha\}\ A$. We write this last property (of l) as

$$\forall\texttt{t}\in l\texttt{. wf(t) \& t }\in\{\alpha\}\ A.$$

Two lists l_1 and l_2 of meta-terms *equal* with respect to a monoid, written

$$l_1\texttt{=}l_2\texttt{ (mod }e,o\texttt{) }\in\{\alpha\}\ A\texttt{ list,}$$

if their implosions have the same value, or, more precisely, if

$$\begin{aligned}&o\texttt{ bin op over }A\{\alpha\}\texttt{ \& }e\ \in\{\alpha\}\ A\\ &\texttt{\& }\forall t\in l_1\texttt{. wf(t) \& t }\in\{\alpha\}\ A\texttt{ \& }\forall t\in l_2\texttt{. wf(t) \& t }\in\{\alpha\}\ A\\ &\texttt{\& (implode }l_1\texttt{ using }e,o\texttt{) = (implode }l_2\texttt{ using }e,o\texttt{) }\in\{\alpha\}\ A.\end{aligned}$$

We can now specify the portion of the normalization procedure that "explodes" a meta-term into a list of meta-terms.

```
∀A,e,o:Atom. ∀t:Term0. ∃l:Term0 list where
  ∀α:Env. monoid(α,A,e,o) => wf@(α,t) => t {α} A
  => ↓([t]=l (mod e,o) ∈{α} A list).
```

The next part of the algorithm is to sort the members of the list of meta-terms with respect to a given ordering. This part is specified by the following theorem.

```
∀A,e,o:Atom. ∀ord:Term0->Term0->Bool. ∀l1:Term0 list.
   ∃l2:Term0 list where
     ∀α:Env. com_monoid(α,A,e,o) => ∀t∈l1. wf(t) & t ∈{α} A
             => ↓(l1=l2 (mod e,o) ∈{α} A list)
```

The last part of the algorithm is to implode the list into a single meta-term. However, since we want this algorithm to act as a rewriting function, there is a complication. Suppose that the meta-term to be rewritten using the procedure is (x *o* *e*). The term will normalize to x. The definition of the type Rewrite(α) requires that the meta-types of (x *o* e) and x be the same atom, but in general this need not be true, since all that can be deduced from the fact that (x *o* *e*) is well-formed is that x$\in\{\alpha\}A$. More will be said about this problem in the next section. The solution here is to do run-time typechecking. The last step of the normalization procedure is to check that the resulting normal form of a term has meta-type A, and fail if it does not. The typechecking step is implemented by the extraction from the following theorem.

```
∀A:Atom. ∀α:Env. ∀t:Term0.
  ?↓(∀α2:Env. α⊂α2 => wf@(α2,t) => mtype(α2,t) = A in Atom)
```

The specification of the last part of the algorithm is the following

```
∀A,e,o:Atom. ∀α1:Env. ∀l:Term0 list. ?∃t:Term0 where
 ∀α2:Env. monoid(α2,A,e,o) => α1⊂α2 => ∀t∈l. wf(t) & t ∈{α2} A =>
   wf@(α2,t) & t ∈{α2} A & ↓(val(implode l using e,o) ={α2} val(t))
```

All that remains is to piece together the parts. To do this, we need an ordering on meta-terms to supply to the sorting procedure. We can use the given environment to order the meta-function and meta-type atoms by taking one atom to precede another if it occurs earlier in the appropriate component of the environment. This ordering is lifted to meta-terms lexicographically. Finally, the whole normalizing procedure is extracted from the proof of the following.

```
∀α1:Env. ∀A,e,o:Atom. ∀t1:Term0. ?∃t2:Term0 where
  ∀α2:Env. α1⊂α2 => com_monoid(α2,A,e,o) =>
    wf@(α2,t1) => ↓(wf@(α2,t2) & val(t1) ={α2} val(t2))
```

```
* top 1 1
1. w: |Q|
2. x: |Q|
3. y: |Q|
4. z: |Q|
5. α: Env
6. (...)
7. val(α,y+0+y+z+z=0 in Q)
8. val(α,x+y+w+z=y+z+y+z in Q)
>> ↓(val(α,0+0=z+x+w+y in Q))

BY (Rewrite 'norm_wrt_Q_plus(α)' ...)

1* 6. val(α,y+y+z+z=0 in Q)
   7. val(α,x+y+w+z=y+y+z+z in Q)
   8. (...)
   >> ↓(val(α,0=x+y+w+z in Q))
```

Figure 5.2: Applying the normalizer for commutative monoids.

Using `norm` to denote the extraction of this theorem, we have the following typing theorem.

```
∀α1:Env. ∀A,e,o:Atom. com_monoid(α1,A,e,o) =>
  ∀α2:Env. α1⊂α2 => norm{α2,A,e,o} in Rewrite(α2).
```

The last theorem in the library is an example of the use of this procedure and of the equality procedure described earlier. We define `norm_wrt_Q_plus(`α`)` to be

```
Topmost(norm{α,Q,Q_0,Q_plus})
```

where `TopMost(`f`)` applies f to all maximal subterms on which f succeeds. Figure 5.2 shows an application of this to a lifted sequent. The single subgoal generated by this is proved using the equality procedure; this is shown in Figure 5.3.

5.9 Conclusion

In this section we discuss a few flaws in the design of the partial-reflection mechanism, point out some aspects with the Nuprl theory and system that hindered the development of this library, and suggest some ways to extend the library.

One of the most serious problems with our partial-reflection mechanism is the restrictive notion of type-matching in the definition of well-formedness for meta-terms. What we really want $t\in\{\alpha\}A$ to assert is simply that the value of t is a member of the type component of the set that is the value of A. The Nuprl type

```
* top 1 1 1
1. w: |Q|
2. x: |Q|
3. y: |Q|
4. z: |Q|
5. α: Env
6. val(α,y+y+z+z=0 in Q)
7. val(α,x+y+w+z=y+y+z+z in Q)
8. (...)
>> ↓(val(α,0=x+y+w+z in Q))

BY (ApplyCompleteTac 'equality(α)' ...)
```

Figure 5.3: Applying the equality procedure.

a in T cannot be used to express this since this type is just a degenerate instance of the equality type a=a in T, which is only well-formed when a is a member of T. It is currently not known whether there is a satisfactory way to add to Nuprl a type that directly encodes membership propositions. For more on this issue, see Allen [3].

Our definition of $t \in \{\alpha\}A$ is a only a weak approximation of a membership proposition. If, for example, we know that $t \in \{\alpha\}A$ and val(α,A)⊂⊂val(α,A'), then it is not in general true that $t \in \{\alpha\}A'$. (The definition of ⊂⊂ is given in Section 5.4.) In fact, it is not even true that if S_1 ⊂⊂ S_2 and S_2 ⊂⊂ S_3 then S_1 ⊂⊂ S_3. This is because of Nuprl's strong type-equality: for T and $\{x:A\,|\,B\}$ to be equal types, T must be a set type $\{x:A'\,|\,B'\}$ where A is equal to A' and for all x in A, B is equal to B'.

These problems can be solved by the introduction of extensional type equality (two types are extensionally equal if they have the same members and same equality relation). Allen's work [2] shows that such an addition is sound. One way of doing this is to directly add a new type constructor for it. Another way, which is suggested by Mendler [48] to express the monotonicity requirement for the introduction of his recursive types, is to add a type constructor ⊂ such that the type $T \subset T'$ is inhabited exactly when all the members of T are also members of T' (and that also the equalities of the two types agree). Using, say, the second approach, we can give a correct definition of $t \in \{\alpha\}A$. First, for a meta-term t and environment α we define a minimal type containing the value of t. Define <$t\{\alpha\}$> by

{ v:|type(α,t)| | val(α,t)=v in type(α,t) }

The new definition for $t \in \{\alpha\}A$ is

<$t\{\alpha\}$> ⊂ |val(α,A)| & ∀x,y:<$t\{\alpha\}$>. x=y in val(α,A).

A related problem has to do with the restriction in the definition of `Rewrite(`α`)` that a meta-term and the result of rewriting must have meta-types which are the same atoms. An example of the problems that can be caused by the restriction was given in the previous section. The addition of the $\subset$ operator here too gives a solution. We can redefine t `=`$\{\alpha\}$ t' to be

$$\texttt{<}t\{\alpha\}\texttt{>} \subset \texttt{<}t'\{\alpha\}\texttt{>} \;\; \texttt{\&} \;\; \texttt{<}t'\{\alpha\}\texttt{>} \subset \texttt{<}t\{\alpha\}\texttt{>}.$$

A less serious problem with the partial-reflection mechanism is an error in the definition of meta-term evaluation. This error is trivial to fix, but it was not discovered until a substantial amount of work had been based on the definition. The clause of the definition that deals with function-application meta-terms is

```
f,args -> if null(mtype(α,f).1) then val(α,f)
          else val(α,f) (g{α}(args)).
```

The problem is that we often will want to reason about applications of the evaluation function where the structure of the meta-term is manifest but the environment is abtract (*e.g.*, it is a variable). It would be convenient in such cases to unfold the application of `val` simply by computation, rather than by using substitution and thus having to reason about the types of the components. For example, we would like to transform a meta-term `val(f(x,y))` (omitting α) into `(val(f))(val(x),val(y))`. However, the clause shown above blocks this, since symbolic computation of `mtype(`α`,f).1` will not (since α is a variable) produce `nil` or a cons. This error seriously complicated the proofs of the theorems relating to the expression normalizer for monoids discussed in the last section.

Several problems with the Nuprl system hindered the development of this library. First, Nuprl's definition mechanism has turned out to be inadequate. As explained in Chapters 2 and 3, Nuprl definitions need not respect the structure of the terms of the type theory. Definitions can supply very good display forms for terms, but they can not always be maintained; substitution, for example, can result in their being lost. Furthermore, the part of the system that attempts to maintain display forms is extremely inefficient, slowing down operations such as normalization of terms in sequents by an order of magnitude. As a consequence, most of the library was constructed with the display maintenance mechanism disabled. Not only did this make proofs harder to construct, but it also seriously detracts from the capability of the proofs to act as explanations of the theory. Solutions are known and awaiting implementation.

Another problem is with the amount of time taken by tactic execution. Typical steps take one to two minutes, which is quite long since most tactics involve no search and usually apply a fairly small number of primitive Nuprl rules. Part of the problem is with the implementation of Nuprl. The version of ML used is an old one and quite slow in comparison with newer ML implementations. The rest of the system could also be considerably sped up with some recoding. Probably the most significant contribution to this problem, though, is the burden of continually reproving typechecking subgoals. Typechecking is recursively undecidable in Nuprl, so the task is left to the user. In practice, the tactic `Member` described in Chapter 2 has been able to prove the vast majority of the typechecking proof obligations. The problem is the number of such requirements that arise. In a single proof, one may have to prove that the same term has a certain type dozens of times. A possible solution to this is for the system to maintain an annotation of terms by types, so that typing information is not continually discarded. Some work was done along these lines by Harper [32], but the approach requires more investigation.

The partial-reflection mechanism can support other kinds of theorem proving procedures than we have implemented so far. Conditional rewrites are an example. Conditional rewriting functions, instead of being just value-preserving functions on meta-terms, would, when given a meta-term t, produce a meta-term t' and a list of meta-terms l such that the members of l represent propositions and such that t and t' have equal values under the assumption that the values of the members of l are true propositions. A theory similar to that developed for ordinary rewriting functions could be developed for conditional rewriting. Another example is congruence closure [42, 52]. This is an efficient procedure for deciding whether an equality follows from a set of equalities using symmetry, reflexivity, transitivity or substitution. This algorithm could be implemented in Nuprl as a graph algorithm parameterized by the congruence relation. However, I do not know if the Nuprl program could be made adequately efficient. Applying this general algorithm via the partial-reflection mechanism would be straightforward.

As a final example, it is possible to use the partial reflection mechanism to implement decision procedures for intuitionistic or classical propositional logic. The latter is possible when computational content is not of interest because it is sound to add to Nuprl the axiom scheme $\downarrow(P\vee\neg P)$.

A more ambitious extension to this work would be to expand the represented language to include constructs with binding variables. This will probably take place in the context of a complete reflection of the Nuprl theory along the lines of [41, 40].

Bibliography

[1] Luigia Aiello and Richard W. Weyhrauch. Using meta-theoretic reasoning to do algebra. In W. Bibel and R. Kowalski, editors, *5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1980.

[2] Stuart F. Allen. A non-type theoretic definition of Martin-Löf's types. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE, 1987.

[3] Stuart F. Allen. A non-type-theoretic semantics for type-theoretic language. Technical Report 87-866, Department of Computer Science, Cornell University, September 1987. Ph.D. Thesis.

[4] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.

[5] Joseph L. Bates. *A Logic For Correct Program Development*. PhD thesis, Cornell University, 1979.

[6] Joseph L. Bates and Robert L. Constable. The nearly ultimate prl. Computer Science Technical Report TR-83-551, Cornell University, 1983.

[7] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Trans. Prog. Lang. Sys.*, 7(1):113–136, 1985.

[8] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967.

[9] Errett Bishop. Mathematics as a numerical language. In et al. J. Myhill, editor, *Intuitionism and Proof Theory*, pages 53–71. North-Holland, 1970.

[10] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer-Verlag, 1985.

[11] W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9:1–36, 1977.

[12] W. W. Bledsoe and Peter Bruell. A man-machine theorem-proving system. *Artificial Intelligence*, 5:51–72, 1974.

[13] R. S. Boyer and J Strother Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J Strother Moore, editors, *The Correctness Problem in Computer Science*, chapter 3. Academic Press, 1981.

[14] Robert S. Boyer and J Strother Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, New York, 1979.

[15] Douglas S. Bridges. *Constructive Functional Analysis.* Pitman, London, 1979.

[16] Alan Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, New York, 1983.

[17] Rance C. Cleaveland. *Type-Theoretic Models of Concurrency.* PhD thesis, Cornell University, 1987.

[18] Avra J. Cohn. *Machine Assisted Proofs of Recursion Implementation.* PhD thesis, University of Edinburgh, 1980.

[19] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233, Ljubljana, 1971.

[20] Robert L. Constable, Scott D. Johnson, and Carl D. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*, volume 135 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1982.

[21] Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates. Writing programs that construct proofs. *Journal of Automated Reasoning*, 1:285–326, 1985.

[22] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *Proceedings of the Second Annual Symposium on Logic in Computer Science.* IEEE, 1987.

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

[24] Thierry Coquand. An analysis of Girard's paradox. In *First Annual Symposium on Logic in Computer Science.* IEEE, 1986.

[25] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, pages 151–184. Springer-Verlag, 1985.

[26] Martin Davis and Jacob T. Schwartz. Metamathematical extensibility for theorem verifiers and proof-checkers. *Computers and Mathematics with Applications*, 5:217–230, 1979.

[27] N. G. de Bruijn. The mathematical language AUTOMATH, its usage and some of its extensions. In *Symposium on Automatic Demonstration, Lecture Notes in Mathematics vol. 125*, pages 29–61, New York, 1970. Springer-Verlag.

[28] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[29] J.-Y. Girard. Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur, 1972. Thèse de Doctorat d'Etat, University of Paris, France.

[30] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[31] David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[32] Robert Harper. *Aspects of the Implementation of Type Theory*. PhD thesis, Cornell University, 1985.

[33] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *The Second Annual Symposium on Logic in Computer Science*. IEEE, 1987.

[34] Douglas J. Howe. Reflected term rewriting in type theory. unpublished manuscript, December 1985.

[35] Douglas J. Howe. Implementing number theory, an experiment with nuprl. In *Proceedings of the Eighth International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 404–415. Springer-Verlag, 1986.

[36] Douglas J. Howe. The computational behaviour of Girard's paradox. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 205–214. IEEE, 1987.

[37] Douglas J. Howe. *Nuprl Release 2.0 Documentation*. Department of Computer Science, Cornell University, March 1988.

[38] Gerard P. Huet. A unification algorithm for the typed lambda-calculus. *Theoretical Computer Science*, 1(1):27–58, 1975.

[39] Richard A. Kemmerer. Verification assessment study, final report. Technical report, National Computer Security Center, 1986.

[40] Todd B. Knoblock. *Metamathematical Extensibility in Type Theory.* PhD thesis, Cornell University, 1987.

[41] Todd B. Knoblock and Robert L. Constable. Formalized metareasoning in type theory. In *Proceedings of the First Annual Symposium on Logic in Computer Science.* IEEE, 1986.

[42] Dexter C. Kozen. *Complexity of Finitely Presented Algebras.* PhD thesis, Cornell University, 1977.

[43] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis.* North-Holland, Amsterdam, 1978.

[44] Per Martin-Löf. An intuitionistic theory of types. Mimeographed manuscript, 1972.

[45] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. F. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118, Amsterdam, 1973. North-Holland.

[46] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[47] David A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics.* PhD thesis, MIT Artificial Intelligence Laboratory, 1987. AI-TR 979.

[48] Paul F. Mendler. *Inductive Definition in Type Theory.* PhD thesis, Cornell University, 1987.

[49] Albert R. Meyer and Mark B. Reinhold. 'Type' is not a type. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages.* Association for Computing Machinery, SIGACT, SIGPLAN, 1986.

[50] B. Q. Monahan. *Data Type Proofs Using Edinburgh LCF.* PhD thesis, University of Edinburgh, 1984.

[51] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[52] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, 1980.

[53] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[54] Lawrence C. Paulson. Tactics and tacticals in Cambridge LCF. Technical Report Report 39, University of Cambridge, Computer Lab., 1983.

[55] Lawrence C. Paulson. Lessons learned from LCF: a survey of natural deduction proofs. *Computer Journal*, 28:197–214, 1985.

[56] Mark Reinhold. Personal communication.

[57] Mark Reinhold. Looping with Girard's paradox. Unpublished manuscript, 1986.

[58] D. Schmidt. A programming notation for tactical reasoning. In R. E. Shostak, editor, *7th Conference on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, pages 445–459. Springer-Verlag, 1984.

[59] N. Shankar. A mechanical proof of the church-rosser theorem. Technical Report 45, Institute for Computing Science. The University of Texas at Austin, 1984.

[60] N. Shankar. Towards mechanical metamathematics. Technical Report 43, University of Texas at Austin, Institute for Computing Science, 1984.

[61] Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

[62] L Wos, R. Overbeek, L. Ewing, and J. Boyle. *Automated Reasoning*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

Appendix A

Translation Details

In Chapter 4 we claimed that the Nuprl proof constructed for Girard's paradox could be translated into a proof in $\nu^{\tau\tau}$. In this appendix we give some justification for this claim. In the first section we discuss the subtheory of Nuprl that was used in the proof. In the second section, we show that any proof in this subtheory has a direct translation into $\nu^{\tau\tau}$. In what follows, we will use the $\nu^{\tau\tau}$ notation for sequents, product types and membership.

A.1 A Subtheory of Nuprl

In this section we discuss a subtheory of Nuprl that we will call Nuprl_0. The rules of Nuprl_0 are just specializations or slight variants of actual Nuprl rules. The verification that the specializations were respected in the Nuprl proof of the paradox was carried out with the aid of several simple ML programs. Tactics that are used in Nuprl proofs must generate subproofs in terms of the primitive inferences rules. Furthermore, these subproofs are retained and are accessible. The ML programs were used to completely scan the primitive proofs and make straightforward checks concerning rule usage.

The language of Nuprl_0 is the language of $\nu^{\tau\tau}$ together with constants `term_of(`*id*`)` for *id* a string. The formation of terms and sequents is as in $\nu^{\tau\tau}$.

Nuprl rules involve not only sequents, but also extractions. Take, for example, the Nuprl rule called *function-introduction* (specialized to the first universe):

$$\frac{\mathcal{A} \vdash A \in \mathtt{U1} \qquad \mathcal{A},\, x\!:\!A \vdash B \quad [b]}{\mathcal{A} \vdash \Pi\, x\!:\!A\,.\,B \quad [\lambda x.\, b].}$$

The extractions are written in square brackets. This rule specifies that if b is the extraction of the second premise, then $\lambda x.\, b$ is the extraction of the conclusion. All the Nuprl rules specify how to compute an extraction for the conclusion from extractions for the premises. This gives an algorithm for computing extractions in any complete Nuprl proof. We may therefore eliminate the extractions by regarding the Nuprl rules as dealing with judgments of the form $\mathcal{A} \vdash a \in A$. For example, we rewrite the *function-intro* rule as

$$\frac{\mathcal{A} \vdash A \in \tau \quad \mathcal{A},\, x\!:\!A \vdash b \in B}{\mathcal{A} \vdash \lambda x.\, b \in \Pi\, x\!:\!A\,.\,B.}$$

We call this rewriting of a rule *making extractions explicit.*

The rules of Nuprl_0 will be given below. For each primitive step of our Nuprl proof we checked that when extractions were made explicit and when all universe constants were replaced by τ then

1. the step was in a subproof whose last (or first, refinement-style) step is an instance of an axiom of Nuprl_0, or

2. (a) the premises and conclusion of the step are syntactically well-formed sequents of Nuprl_0 and

   (b) the step is an instance of one of the rule schemes given below.

1 and *2(a)* are trivial to check. *2(b)* is straightforward since each rule (except *type-in-type* and *def*) is either a rewritten version of a Nuprl rule or of a special case of a Nuprl rule.

We now give the Nuprl_0 rules. There is redundancy in these rules because many are obtained directly as degenerate cases of Nuprl rules. These rules also reflect Nuprl's library structure, since there are rules that refer to previously proven theorems.

type-in-type

$$\frac{}{\mathcal{A} \vdash \tau \in \tau}$$

Π-formation-1

$$\frac{\mathcal{A} \vdash A \in \tau \quad \mathcal{A},\, x\!:\!A \vdash B \in \tau}{\mathcal{A} \vdash \Pi\, x\!:\!A\,.\,B \in \tau}$$

Π-formation-2

$$\frac{\mathcal{A} \vdash A \in \tau \quad \mathcal{A} \vdash B \in \tau}{\mathcal{A} \vdash \Pi\, x\!:\!A\,.\,B \in \tau}$$

where x does not occur free in B.

function-intro

$$\frac{\mathcal{A} \vdash A \in \tau \quad \mathcal{A},\, x\!:\!A \vdash b \in B}{\mathcal{A} \vdash \lambda x.\, b \in \Pi\, x\!:\!A\,.\,B}$$

function-elim-1

$$\frac{\mathcal{A} \vdash f \in \Pi\, x\!:\!A\,.\,B \quad \mathcal{A} \vdash a \in A}{\mathcal{A} \vdash f(a) \in B[a/x]}$$

function-elim-2

$$\frac{\begin{array}{c}\mathcal{A},\, f\!:\!(\Pi\, x\!:\!A\,.\,B),\, \mathcal{A}' \vdash a \in A \\ \mathcal{A},\, f\!:\!(\Pi\, x\!:\!A\,.\,B),\, \mathcal{A}',\, y\!:\!B[a/x] \vdash t \in T\end{array}}{\mathcal{A},\, f\!:\!(\Pi\, x\!:\!A\,.\,B),\, \mathcal{A}' \vdash t[f(a)/y] \in T}$$

Instances of the Nuprl *function-elim-2* rule can have another hypothesis in the second premise; it was checked that our proof contained no such instances. The following two rules are degenerate because we are replacing universes `Ui` by τ.

universe-intro

$$\frac{}{\mathcal{A} \vdash \tau \in \tau}$$

cumulativity

$$\frac{\mathcal{A} \vdash T \in \tau}{\mathcal{A} \vdash T \in \tau}$$

hypothesis

$$\frac{}{\mathcal{A},\ x\!:\!A,\ \mathcal{A}' \vdash x \in A}$$

sequence

$$\frac{\mathcal{A} \vdash a \in A \quad \mathcal{A},\ x\!:\!A \vdash b \in B}{\mathcal{A} \vdash (\lambda x.\, b)(a) \in B}$$

lemma

$$\frac{\mathcal{A},\ x\!:\!A \vdash t \in T}{\mathcal{A} \vdash t[a/x] \in T}$$

if there is a previously proven lemma with conclusion $a \in A$.

def

$$\frac{}{\mathcal{A} \vdash \texttt{term\_of}(name) \in T}$$

if there is a previously proven lemma $t \in T$ named *name.*

explicit-intro

$$\frac{\mathcal{A} \vdash t \in T}{\mathcal{A} \vdash t \in T}$$

The Nuprl *explicit-intro* rule allows the introduction of a term t which is to be the extraction of the sequent.

For the next two rules reduction is as in $\nu^{\tau\tau}$, except that it can additionally involve the replacement of "`term_of`" constants by their denotations.

direct-computation

$$\frac{\mathcal{A} \vdash t \in T' \quad T \rightarrow T'}{\mathcal{A} \vdash t \in T}$$

direct-computation-hyp

$$\frac{\mathcal{A},\ x\!:\!A',\ \mathcal{A}' \vdash t \in T \quad A \rightarrow A'}{\mathcal{A},\ x\!:\!A,\ \mathcal{A}' \vdash t \in T}$$

equality

$$\frac{}{\mathcal{A},\ x\!:\!A,\ \mathcal{A}' \vdash x \in A}$$

thinning

$$\frac{\mathcal{A}' \vdash t \in T}{\mathcal{A} \vdash t \in T}$$

where $\mathcal{A}'$ is a subsequence of $\mathcal{A}$.

A.2 Translating the Subtheory

For each step of the Nuprl proof that used a rule corresponding to the Nuprl_0 *sequence* or *direct-computation* rules, we constructed an additional proof of $\mathcal{A} \vdash A \in \tau$ and $\mathcal{A} \vdash T \in \tau$ respectively. We then checked, in the way described in the preceding section, that these new proofs could be translated into Nuprl_0. Furthermore, we checked that these new Nuprl_0 proofs did not use the *sequence* or *direct-computation* rules. We may assume, then, that the Nuprl_0 *sequence* and *direct-computation* rules have the additional premises $\mathcal{A} \vdash A \in \tau$ and $\mathcal{A} \vdash T \in \tau$, respectively.

We now successively eliminate rules from Nuprl_0. First, note that if we add hypotheses to the conclusion of any rule instance, we can add hypotheses to the premises to make another rule instance. Hence we can eliminate the *thinning* rule. We can eliminate Π-*formation-2* since it can be reduced to Π-*formation-1*. The *equality* rule is identical to the *hypothesis* rule, *universe-intro* is identical to *type-in-type*, and obviously *cumulativity* and *explicit-intro* can be eliminated. The *def* rule can be eliminated by replacing all "`term_of`" terms by their denotations and using the previous theorem's proof. We can eliminate *sequence* since it is derivable from *function-elim-1* and *function-intro*.

Suppose we have a proof p whose root is an instance r of *function-elim-2* and which does not contain any other instances of that rule or any of the rules eliminated above. From the first premise of r, $f(a) \in B[a/x]$ follows (omitting hypotheses) via *hypothesis* and *function-elim-1*. In the subproof rooted at the second premise of r, remove from all sequents the hypothesis declaring y, replace all free occurrences of y to the right of that declaration by $f(a)$, and replace applications of the hypothesis rule to y by the proof of $f(a) \in B[a/x]$. The result is a proof with the same root sequent as p and which does not use *function-elim-2*. Hence this rule can be eliminated. A similar argument shows that *lemma* can be eliminated.

We are now left with $\nu^{\tau\tau}$.

Appendix B

Additional Rules

The rules discussed in this appendix together with the rules given in Chapter 8 of the Nuprl book [23] form the set of rules of the version of Nuprl used for the bulk of the work described in this thesis. This version of Nuprl was used to build the library described in Chapter 5. The changes that resulted in this version took place over several years, so some of the other libraries that are referred to in this thesis were built with intermediate versions. We will not discuss these other versions.

With the exception of the type `Object` the Nuprl terms referred to in the rules below are all described in Chapter 2. Most of the rules below strictly extend the inference system given in the Nuprl book. There are three exceptions: the extensionality rule has been strengthened, the computation rules for the "decision" terms have been strengthened, and some of the rules for set types have been changed to reflect a change in the definition of equality of set types. All of the changes that have been made, except as noted below, have straightforward semantic justifications which we will not give here. See Allen [3] for the semantics of Nuprl.

Computation

Evaluation

The *evaluation* rules are efficient special cases of the direct computation rules described in the Nuprl book. They are important for the work described in Chapter 5, where they are used, for example, to execute theorem-proving procedures written in Nuprl's type theory.

For the purpose of the description below, we define a particular evaluation function, call it v, which takes a list *thm_names* of names of theorems, a boolean value b and a term t, and returns a term t' that can be obtained from t by performing a particular sequence of computation steps. Specifically, evaluation proceeds as follows. If t is non-canonical, evaluate the principal argument(s) and replace them in t by their values, obtaining a term t'. If t' is a redex, then contract it and return the value of the result, otherwise return t'. If t is canonical then return the result of replacing in t by their values the immediate subterms that are not within the scope of a binding variable of t. If t is `term_of(`*name*`)` then return t if b is *true* and *name* is in the list *thm_names* or if b is *false* and b in not in *thm_names*, otherwise return the value of the term that `term_of(`*name*`)` stands for. If t is a variable, then return t.

The two evaluation rules are as follows.

$$\texttt{>>}\ T\ [\texttt{ext}\ t]\ \texttt{by eval}\ b\ name_1\texttt{,}\ \ldots\texttt{,}name_n$$
$$\texttt{>>}\ T'\ [\texttt{ext}\ t]$$

where b is `true` or `false`, each $name_i$ is an identifier, and

$$T' = v([name_1; \ldots; name_n], b, T).$$

The analogous rule for hypotheses is

$$H\texttt{,}\ x\!:\!T\texttt{,}\ H'\ \texttt{>>}\ G\ [\texttt{ext}\ t]\ \texttt{by eval\_hyp}\ i\ b\ name_1\texttt{,}\ldots\texttt{,}name_n$$
$$H\texttt{,}\ x\!:\!T'\texttt{,}\ H'\ \texttt{>>}\ G\ [\texttt{ext}\ t]$$

where b is `true` or `false`, each $name_i$ is an identifier, the hypothesis $x : T$ is the i^{th}, and T' is as above.

Reverse Direct Computation

These rules are similar to the direct computation rules described in the Nuprl book. The difference is that they allow computation in the reverse direction. The rules in the book allow replacement of some component of a sequent by the result of performing some computation steps on it. The new rules allow replacement of a component T of a sequent by some term T' such that T is the result of performing some computation steps on T'. These rules are proven sound in the same way as the direct computation rules of the book. See [3] for details.

The rules are as follows.

$$H\ \texttt{>>}\ T\ [\texttt{ext}\ t]\ \ \texttt{by reverse\_direct\_computation using}\ S$$
$$\texttt{>>}\ T'\ [\texttt{ext}\ t]$$

where S is a legal tagging (as defined in the book) of T', and T is the result of performing the computations indicated in S.

$$H\texttt{,}\ x\!:\!T\texttt{,}\ H'\ \texttt{>>}\ T''\ \ [\texttt{ext}\ t]\ \texttt{by reverse\_direct\_computation\_hyp}\ i\ S$$
$$H\texttt{,}\ x\!:\!T'\texttt{,}\ H'\ \texttt{>>}\ T''\ [\texttt{ext}\ t]$$

where S is a legal tagging of T', the hypothesis $x : T$ is the i^{th}, and T is the result of performing the computations indicated in S.

Recursive Types

The general form of the recursive type constructor presented in the book can be somewhat unwieldy in practice. A simpler form has been implemented which does not allow mutual recursion or recursive definition of type-valued functions.

The proof rules follow. The formation rule for recursive types is

```
H >> rec(Z.T) in Ui by intro new Y
Y:Ui >> T[Y/Z] in Ui
```

where no instance of Z bound by rec($Z.T$) may occur in the domain type of a function space, in the argument of a function application or in the (a) principal argument of an elimination form that is not an application. The introduction rule is

```
H >> rec(Z.T) [ext t] by intro at Ui
    >> T[rec(Z.T)/Z] [ext t]
    >> rec(Z.T) in Ui.
```

The equality-introduction rule for recursive types is

```
H >> t in rec(Z.T) by intro at Ui
    >> t in T[rec(Z.T)/Z]
    >> rec(Z.T) in Ui
```

There is a special "unrolling rule" since it cannot be derived (as it can with the list types) from the elimination rule:

```
H, x:rec(Z.T), H' >> G [ext g[x/y]]  by unroll x new y
y:T[rec(Z.T)/Z], y=x in T[rec(Z.T)/Z] >> G[y/x] [ext g].
```

The elimination rule is

```
H, x:rec(Z.T), H' >> G [ext rec_ind(x;w,z.g[\x.void/u])]
                          by elim x at Ui new u,v,w,z
                       >> rec(Z.T) in Ui
u: rec(Z.T)->Ui, w: (x:{v:rec(Z.T)|u(v)}->G),
z: T[{v:rec(Z.T)|u(v)}/Z]
                       >> G[z/x] [ext g].
```

Finally, the equality-introduction rule for rec_ind terms is

```
H >> rec_ind(r;h,z.t) in S[r/x]
          by intro over x.S using rec(Z.T) at Ui new u,v,w
     >> r in rec(Z.T)
     >> rec(Z.T) in Ui
u: rec(Z.T)->Ui, v: w:{w:rec(Z.T)|u(w)}->S[w/x],
w: T[{w:rec(Z.T)|u(w)}/Z]
     >> t[v,w/h,z] in S[w/x]
```

The recursive type rules that were proven sound by Mendler in [48] have a slightly different form than the ones here. The main difference is that the recursive type constructor there is parameterized by universe level. Rather than attempt to modify Mendler's soundness proof, we just note that there is only one recursive type used in our work, namely the type `Term0` defined in Chapter 5. It is a simple matter to verify directly from the Nuprl semantics that the above rules are sound for `Term0`.

Set Types

Two of Nuprl's set rules have been changed. First is the set elimination rule (set rule number 9, page 167 in the book). There are two cases to the new version of this rule:

$$\begin{array}{l} H\texttt{,}\ u\texttt{:}\{x\texttt{:}A\,|\,B\}\texttt{,}\ H'\ \texttt{>>}\ T\ [\texttt{ext}\ t]\ \texttt{by elim}\ u \\ H\texttt{,}\ u\texttt{:}A\texttt{,}\ \texttt{[}B[u/x]\texttt{]}\texttt{,}\ H'\ \texttt{>>}\ T\ [\texttt{ext}\ t] \end{array}$$

where the notation $\texttt{[}B[u/x]\texttt{]}$ means that the hypothesis $B[u/x]$ is *hidden* (see the book for a definition of this), and

$$\begin{array}{l} H\texttt{,}\ \{x\texttt{:}A\,|\,B\}\texttt{,}\ H'\ \texttt{>>}\ T\ [\texttt{ext}\ t]\ \texttt{by elim}\ i\ [\texttt{new}\ y] \\ \quad H\texttt{,}\ \{x\texttt{:}A\,|\,B\}\texttt{,}\ H'\texttt{,}\ y\texttt{:}A\texttt{,}\ \texttt{[}B[y/x]\texttt{]}\ \texttt{>>}\ T\ [\texttt{ext}\ t] \end{array}$$

The other changed set rule is the equality rule (number 10, page 168).

$$\begin{array}{l} \texttt{>>}\ \{x\texttt{:}A\,|\,B\}\ \texttt{=}\ \{x\texttt{:}A'\,|\,B'\}\ \texttt{in U}i\ \ \texttt{by intro}\ [\texttt{new}\ z] \\ \quad \texttt{>>}\ A\texttt{=}A'\ \texttt{in U}i \\ z\texttt{:}A\ \texttt{>>}\ B[z/x]\texttt{=}B'[z/x]\ \texttt{in U}i \end{array}$$

The main reason for the change to the set rules was to remove the well-formedness subgoal from the set elimination rule. Eliminating $\{x:A\,|\,B\}$ using the old rule generate a subgoal

$$x\texttt{:}A\ \texttt{>>}\ B\ \texttt{in U}i$$

This turned out to cause unacceptable practical problems. The change requires modifying the semantics to give a new definition of equality of set types. The semantic change is reflected in the last rule above. The old version of the rules only required showing B and B' to be equivalent under `<=>`.

The Type `Object`

The type `Object` is the collection of all canonical closed terms. Its equality is trivial: all members are equal. Following are the three rules associated with the type.

```
>> object in Ui  BY intro

>> t in object  BY intro [using A]
  [>> t in A]

>> t=t' in object BY intro
  >> t in object
  >> t' in object
```

In the second rule, if t is canonical then no subgoal is generated, and if t is not canonical, then the "using" term must be present and the subgoal is generated. In the work described in this thesis, the type `Object` is used only in making definitions in the way described in the first section of Chapter 3.

Other Rules

The rules described in this section are of less interest in connection with this thesis than the ones given above, so many of the details are omitted. A complete description is contained in the documentation set that is part of the Nuprl system [37].

In the Nuprl book the `int_eq`, `atom_eq` and `less` terms have associated computation rules. These rules have been replaced by stronger versions. We give only an example of these rules since the others can be easily deduced.

```
>> int_eq(a;b;t;t') = t'' in T  BY reduce 1 true
  >> a = b in Int
  >> t = t'' in T
```

One supplies an integer to indicate which term of the equality is to be reduced, and then either `true` or `false` to indicate which case holds.

There have been several additions related to reasoning about the integers. First, the *monotonicity* rule of PL/CV2 [20] has been adapted to Nuprl. This is a table-driven rule that concisely encapsulates a collection of monotonicity properties of integer equality and inequality. An example of the use of the rule is

```
a<b, c<d >> T   BY monotonicity 1 + 2
        a+c<b+d >> T.
```

Second, the *division* rule has been added to axiomatize the integer operations `/` (division) and `mod`. Finally, the `arith` decision procedure has been slightly extended to make better use of the hypothesis list.

Nuprl's extensionality rule has been strengthened. The extensionality rule allows one to show functions equal by showing they have equal values on all arguments. The original rule required that the functions be in the same type in which they are to be proven equal. The new rule only requires that they be in *some* function type.

Finally, the *thinning* rule deletes specified hypotheses, subject to the restriction that the sequent remain closed.

Appendix C

The Partial-Reflection Library

C.1 Complete Listing

The first part of this appendix gives a complete listing of the library that was described in Chapter 5. What we show is similar to what would be displayed in Nuprl's library window, with the following exceptions.

- The bodies of DEF, EVAL and ML library objects are also displayed.
- The extracted term of any theorem whose name ends with an underscore is displayed after the statement of the theorem, unless the term is very large, in which case it is elided.
- Whitespace and some parenthesization has been modified.
- We have removed some library objects that served as division markers.
- Section headings and a few comments have been added.
- A small amount of reordering of objects has been done.

The second part of the appendix gives a listing of a subproof of the proof of the theorem `match_`. This proof was described in Section 5.7.

C.1.1 Basics

```
* ML kill_tactic_additions
  reinitialize_lists () ;;

* DEF c
  %<c:comment>% ==

* DEF parens
```

```
  (<a:anything>) == (<a>)

* DEF F
  <T:tactic> == (FastAp (\p. (<T>) p))

* DEF t
  (<T:tactic>...) == (\p.((<T>) THEN Autotactic) p)

* DEF w
  (<T:tactic>...) == (\p.((<T>) THEN WeakAutotactic) p)

* DEF tf
  (<T:tactic>...) == (FastAp (\p.((<T>) THEN Autotactic) p))

* DEF wf
  (<T:tactic>...) ==
  (FastAp (\p. let c = concl p in ((<T>) THEN WeakAutotactic) p))

* DEF ts
  (<T:tactic>...!) ==
  (\p.((<T>) THEN Repeat (Autotactic ORELSE MemberI)) p)

* DEF tc
  (<T:tactic>...*) == (\p. ((<T>) THEN (FastAp o Try) (CompleteAutotactic)) p)


* DEF wc
  (<T:tactic>...*) ==
  (\p. ((<T>) THEN (FastAp o Try) (CompleteWeakAutotactic)) p)

* DEF tp
  (<T:tactic>...when ¬(<P:bool (free p)>)) ==
  (\p.((<T>) THEN IfThen (\p.not(<P>) ? true) Autotactic) p)

* DEF wp
  (<T:tactic>...when ¬(<P:bool (free p)>)) ==
  (\p.((<T>) THEN IfThen (\p.not(<P>) ? true) WeakAutotactic) p)

* DEF tm
  (<T:tactic>...∈) == (\p.((<T>) THEN IfThen is_membership_goal Autotactic) p)


* DEF wm
  (<T:tactic>...∈) ==
  (\p.((<T>) THEN IfThen is_membership_goal WeakAutotactic) p)
```

```
* DEF tw
  (<T:tactic>...wf) == (\p.((<T>) THEN IfThen is_wf_goal Autotactic) p)

* DEF ww
  (<T:tactic>...wf) == (\p.((<T>) THEN IfThen is_wf_goal WeakAutotactic) p)

* DEF tn
  (<T:tactic>...¬) == (\p.((<T>) THENO Autotactic) p)

* DEF wn
  (<T:tactic>...¬) == (\p.((<T>) THENO WeakAutotactic) p)

* DEF nothing
  <a:anything> ==

* DEF e
  <x:def> == term_of(<x>_)

* DEF to
  <x:thm name> == term_of(<x>)

* DEF g
  γ == gamma

* DEF d
  δ == delta

* DEF a
  α == alpha

* DEF squash
  ↓(<T:type>) == {(0 in int)|(<T>)}

* ML add_squash
  add_matching_def_adder `squash`
  '{(0 in int)|T}' ``T`` (\l.true)
  ;;

* THM squash2_
  >> Type->Type
  Extraction:
  λA. ↓(A)

* DEF squash2
  ↓(<T:type>) == squash2(<T>)
```

```
* ML ISquashed
  % Only implemented for & so far %

  let ISquashed p =
  ( let A,B = destruct_and (destruct_squash (concl p)
              ? snd (destruct_apply (concl p))) in
    Assert (make_product_term `NIL`
             (make_ext_ap `squash2` [A]) (make_ext_ap `squash2` [B]))
    THENL [Id; Complete (Unfold `squash2`...)]
  ) p
  ;;

* DEF spread
  let <x:var>,<y:var>=<t:term> in <tt:term>
  == spread(<t>;<x>,<y>.<tt>)

* DEF dblspread
  let <w:var>,<x:var>,<y:var>,<z:var>=
  <t1:term>,<t2:term> in <t3:term>==
  let <w>,<x>=<t1> in let <y>,<z>=<t2> in <t3>
```

Logic

```
* DEF and
  <P:prop> & <Q:prop> == ((<P>)#(<Q>))

* DEF or
  <P:prop> ∨ <Q:prop> == ((<P>)|(<Q>))

* DEF imp
  <P:prop> => <Q:prop> == ((<P>)->(<Q>))

* DEF not
  ¬(<P:prop>) == <P> => void

* ML add_not
  add_matching_def_adder `not`
  'x -> void' ``x`` (\l. true)
  ;;

* DEF neq
  <t:term> ≠ <tt:term> in <T:type> == ¬((<t>)=(<tt>) in (<T>))
```

```
* DEF iff
  <P:prop> \<=> <Q:prop> == <P> => <Q> & <Q> => <P>

* ML add_iff
  add_matching_def_adder `iff` 'P <=> Q' ``P Q`` (\x.true) ;;

* DEF all
  ∀<x:var>:<T:type>. <P:prop> == (<x>:(<T>)->(<P>))

* DEF all2
  ∀<x:var>,<y:var>:<T:type>. <P:prop>
  == ∀<x>:<T>. ∀<y>:<T>. <P>

* DEF all3
  ∀<x:var>,<y:var>,<z:var>:<T:type>. <P:prop>
  == ∀<x>:<T>. ∀<y>,<z>:<T>. <P>

* DEF all4
  ∀<w:var>,<x:var>,<y:var>,<z:var>:<T:type>. <P:prop>
  == ∀<w>:<T>. ∀<x>,<y>,<z>:<T>. <P>

* DEF set
  {<x:var>:<T:type>|<P:prop>} == ({<x>:(<T>)|(<P>)})

* DEF all_where
  ∀<x:var>:<T:type> where <P:prop>. <PP:prop> ==
  ∀<x>:<T>. ↓(<P>) => <PP>

* DEF all2_where
  ∀<x:var>,<y:var>:<T:type> where <P:prop>. <PP:prop>
  == ∀<x>:<T>. ∀<y>:<T> where <P>. <PP>

* DEF all3_where
  ∀<x:var>,<y:var>,<z:var>:<T:type> where <P:prop>.<PP:prop>
  == ∀<x>:<T>. ∀<y>,<z>:<T> where <P>. <PP>

* DEF some
  ∃<x:var>:<T:type>. <P:prop> == (<x>:(<T>)#(<P>))

* DEF some2
  ∃<x:var>,<y:var>:<T:type>. <P:prop>
  == ∃<x>:<T>. ∃<y>:<T>. <P>

* DEF some3
  ∃<x:var>,<y:var>,<z:var>:<T:type>.<P:prop>
  == ∃<x>:<T>. ∃<y>,<z>:<T>. <P>
```

```
* DEF some_where
  ∃<x:var>:<T:type> where <P:prop> == {<x>:(<T>)|(<P>)}

* DEF some2_where
  ∃<x:var>,<y:var>:<T:type> where <P:prop> ==
  ∃<x>:<T>. ∃<y>:<T> where <P>

* DEF true
  True == (0 in int)

* DEF True
  True == (0 in int)

* DEF false
  False == void

* DEF False
  False == void

* THM or_decidability
  >> ∀P,Q:Type. P ∨ ¬(P) => Q ∨ ¬(Q) => (P ∨ Q) ∨ ¬(P ∨ Q)

* THM and_decidability
  >> ∀P,Q:Type. P ∨ ¬(P) => Q ∨ ¬(Q) => P & Q ∨ ¬(P & Q)

* THM imp_decidability
  >> ∀P,Q:Type. P ∨ ¬(P) => Q ∨ ¬(Q) => (P => Q) ∨ ¬(P => Q)

* ML PropDecidability
  let PropDecidability =
  (Progress o RepeatW)
  ( Lemma 'or_decidability' ORELSE
    Lemma 'and_decidability' ORELSE
    Lemma 'imp_decidability' ORELSE
    Lemma 'Atom_eq_decidable'
  )
  ;;

  add_to_Decidable 'PropDecidability'
  PropDecidability
  ;;
```

Some Very Simple Definitions

```
* DEF ax
  ax == axiom

* DEF p1
  <x:tuple>.1 == spread(<x>;u,v.u)

* DEF p2
  <x:tuple>.2 == spread(<x>;u,v.v)

* DEF p3
  <x:tuple>.3 == <x>.2.2

* ML add_projs
  add_matching_def_adder `p1`
  'spread(p;u,v.u)' ``p`` (\l.true)
  ;;

  add_matching_def_adder `p2`
  'spread(p;u,v.v)' ``p`` (\l.true)
  ;;

* DEF tup
  \<<a:term>,<b:term>\> == <(<a>),(<b>)>

* DEF tup3
  \<<a:term>,<b:term>,<c:term>\> == <(<a>),<<b>,<c>>>

* DEF tup4
  \<<a:term>,<b:term>,<c:term>,<d:term>\> == <(<a>),<<b>,<c>,<d>>>

* DEF let
  let <x:var> = <t:term> in <tt:term> == ((\ <x>.(<tt>))(<t>))

* DEF let2
  let <x:var>,<y:var> = <p:pair> in <t:term> ==
  let <x> = <p>.1 in let <y> = <p>.2 in <t>

* DEF let3
  let <x:var>,<y:var>,<z:var> = <p:triple> in <t:term> ==
  let <x> = <p>.1 in let <y> = <p>.2.1 in let <z> = <p>.2.2 in <t>

* DEF let4
  let <x:var>,<y:var>,<z:var>,<a:var> = <p:quad> in <t:term> ==
```

```
  let <x> = <p>.1 in
  let <y> = <p>.2.1 in
  let <z> = <p>.2.2.1 in let <a> = .2.2.2 in
  <t>

* DEF if_eq
  if <x:int>=<y:int> then <s:term> else <t:term> == int_eq(<x>;<y>;<s>;<t>)

* DEF if_aeq
  if <x:Atom>=<y:Atom> then <s:term> else <t:term> == atom_eq(<x>;<y>;<s>;<t>)


* DEF if_l
  if <x:int>\<<y:int> then <s:term> else <t:term> == less(<x>;<y>;<s>;<t>)

* DEF dec
  d(<a:term>;<u:var>.<t:term>;<v:var>.<tt:term>) ==
  decide(<a>;<u>.<t>;<v>.<tt>)

* DEF isl
  isl(<x:A|B>) == d(<x>;u.True;u.False)

* DEF isr
  isr(<x:A|B>) == d(<x>;u.False;u.True)

* DEF outl
  outl(<x:A|B>) == d(<x>;u.u;u."uu")

* DEF l
  λ<x:var>. <t:term> == (\ <x>.(<t>))

* DEF l2
  λ <x:var> <y:var>. <t:term> == λ<x>. λ<y>. <t>

* DEF l3
  λ <x:var> <y:var> <z:var>. <t:term> == λ<x>. λ<y>. λ<z>. <t>

* DEF dl
  λ <x:var>,<y:var>. <t:term> == λp. let <x>,<y> = p in <t>

* DEF bv
  <x:var> -> <t:term> == (\ <x>.(<t>))

* DEF bv2
  <x:var>,<y:var> -> <t:term> == <x> -> <y> -> <t>
```

```
* DEF bv3
  <x:var>,<y:var>,<z:var> -> <t:term> == <x> -> <y>,<z> -> <t>

* DEF prim_rec
  [ 0 → <b:base case> ; <n:int var>, <y:var> → <t:term> ; @ <a:int>] ==
  ind( <a>; _,_.<b>; <b>; <n>,<y>. <t> )

* DEF list
  <A:Type> list == ((<A>) list)

* DEF list_rec
  [ nil → <b:term>; <h:var>.<t:var>,<v:var> → <tt:term>; @ <a:list>] ==
  list_ind( <a>; <b>; <h>,<t>,<v>. <tt> )

* DEF cons
  <h:A>.<t:A list> == ((<h>).(<t>))

* DEF ap
  <f:term>(<x:term>) == ((<f>)(<x>))

* DEF bin_ap
  <r:term>(<a:term>,<b:term>) == <r>(<a>)(<b>)

* DEF tri_ap
  <r:term>(<a:term>,<b:term>,<c:term>) == <r>(<a>)(<b>)(<c>)

* DEF ap4
  <r:term>(<a:term>,<b:term>,<c:term>,<d:term>) == <r>(<a>)(<b>)(<c>)(<d>)

* DEF bin_tap
  <r:fun>(<t:arg1>,<tt:arg2>) == <r>(<(<t>),(<tt>)>)

* DEF type_ap
  <f:term><x:term> == ((<f>)(<x>))
```

The Integers

```
* DEF le
  <x:Int>≤ <y:Int> == ¬( (<y>)<(<x>))

* DEF lele
  <x:Int>≤ <y:Int>≤ <z:Int> == <x>≤ <y> & <y>≤ <z>
```

```
* DEF lle
  <x:Int>\<<y:Int>≤ <z:Int> == (<x>)<(<y>) & <y>≤ <z>

* DEF lel
  <x:Int>≤ <y:Int>\<<z:Int> == <x>≤ <y> & (<y>)<(<z>)

* DEF ll
  <x:Int>\<<y:Int>\<<z:Int> == (<x>)<(<y>) & (<y>)<(<z>)

* THM Int_abs_
  >> Int -> Int
  Extraction:
  λ n.  less(n;0;-n;n)

* DEF Int_abs
  |<n:Int>| == term_of(Int_abs_)(<n>)

* DEF eq
  <m:Int> = <n:Int> == ((<m>)=(<n>) in Int)

* THM N_
  >> U1
  Extraction:
  {n:Int|0≤ n}

* DEF N
  N == term_of(N_)

* THM Int_eq_if_N_eq
  >> ∀x,y:N. x=y in N => x=y in Int

* THM N_max_
  >> N -> N -> N
  Extraction:
  λ m n. if m<n then n else m

* DEF N_max
  max(<m:N>,<n:N>) == N_max(<m>)(<n>)

* THM Int_tail_
  >> Int -> U1
  Extraction:
  λm.  {n:int|m≤ n}

* DEF Int_tail
  {<n:Int>..} == term_of(Int_tail_)(<n>)
```

```
* THM Int_seg_
  >> Int -> Int -> U1
  Extraction:
  λ  m  n .  {i:Int|m≤ i≤ n}

* DEF Int_seg
  {<m:Int>..<n:Int>} == term_of(Int_seg_)( <m>)( <n>)

* THM N_plus_
  >> U1
  Extraction:
  {n:Int|0<n}

* DEF N_plus
  N+ == term_of(N_plus_)

* DEF Type
  Type == U13

* THM isl_char
  >> ∀A:Type. ∀x:A ∨ ¬(A). isl(x) <=> A
```

Universe Polymorphism

```
* DEF infinity
  (∞-1) == 12

* THM omega_
  >> U1
  Extraction:
  {1..(∞-1)}

* DEF omega
  ∞ == term_of(omega_)

* THM some_omega_members
  >> 1 in ∞ & 8 in ∞ & 2 in ∞ & 3 in ∞ & 4 in ∞ & 5 in ∞ & 6 in ∞
     & 7 in ∞ & 9 in ∞ & 10 in ∞ & 11 in ∞ & 12 in ∞

* ML add_some_omega_members
  add_to_member_i `some_omega_members`
  (IfOnConcl (\c. let [t],T = destruct_equal c in is_integer_term t & T='∞')
```

```
    (Refine (lemma `NIL` `some_omega_members`)
     THEN OnLastHyp FastRepeatAndE THEN Trivial)
    Fail
  )
  ;;

* THM U_list_
  >> Object
  Extraction:
  <1,U1>.<2,U2>.<3,U3>.<4,U4>.<5,U5>.<6,U6>.<7,U7>.<8,U8>.<9,U9>
  .<10,U10>.<11,U11>.<12,U12>.nil

* THM U_
  >> ∞->Type
  Extraction:
  λi. if 0<i then [ nil → "uu"; h.t,v → if h.1=i then h.2 else v; @ U_list]

      else "uu"

* DEF U
  U<i:int> == term_of(U_)(<i>)

* DEF all_types
  ∀<x:var>:Type. <P:prop> == ∀i:∞. ∀<x>:Ui. <P>

* THM omega_cases
  >> ∀P:∞->Type.
      P(1) & P(2) & P(3) & P(4) & P(5) & P(6) & P(7) & P(8)
           & P(9) & P(10) & P(11) & P(12)
      => ∀i:∞. P(i)

* THM omega_cases_2
  >> ∀i:∞. Ui=U1 in Type ∨ Ui=U2 in Type ∨ Ui=U3 in Type ∨
        Ui=U4 in Type ∨ Ui=U5 in Type ∨ Ui=U6 in Type ∨
        Ui=U7 in Type ∨ Ui=U8 in Type ∨ Ui=U9 in Type ∨
        Ui=U10 in Type ∨ Ui=U11 in Type ∨ Ui=U12 in Type

* ML OmegaCases
  let OmegaCases t =
    InstantiateLemma `omega_cases_2` [t]
    THEN OnLastHyp RepeatOrE
  ;;

* ML LiftWf
  let LiftWf =
```

```
 (Id...) THEN (OmegaCases 'i'...)
 THEN OnLastHyp
      \i. (let Ucase = (second o equands o type_of_hyp i) p  in
           Assert (make_equal_term Ucase (equands (concl p)))...)
;;

* THM U_members
  >> ∀i:∞. ∀A:Ui. A in Type

* THM U_cumulativity
  >> ∀i:∞. i<(∞-1) => ∀A:Ui. A in U(i+1)

* THM U1_contained_in_Ui
  >> ∀i:∞. ∀A:U1. A in Ui

* THM and_wf
  >> ∀i:∞. ∀A,B:Ui. A & B in Ui

* THM squash_wf
  >> ∀i:∞. ∀A:Ui. ↓(A) in Ui

* THM prod_wf
  >> ∀i:∞. ∀A:Ui. ∀B:A->Ui. x:A#(B(x)) in Ui

* THM set_wf
  >> ∀i:∞. ∀A:Ui. ∀B:A->Ui. {x:A|B(x)} in Ui

* THM imp_wf
  >> ∀i:∞. ∀A,B:Ui. A => B in Ui

* THM fun_wf
  >> ∀i:∞. ∀A:Ui. ∀B:A->Ui. x:A->(B(x)) in Ui

* THM or_wf
  >> ∀i:∞. ∀A,B:Ui. A ∨ B in Ui

* THM list_wf
  >> ∀i:∞. ∀A:Ui. A list in Ui

* THM eq_wf
  >> ∀i:∞. ∀A:Ui. ∀x,y:A. (x=y in A) in Ui

* ML AbstractAndApplyWfLemma
  let AbstractAndApplyWfLemma p =
  ( if not is_gen_universe_term (concl_type p) then fail ;
    ReverseComputeConclUsing
```

```
    (map_on_member abstract_and_tag_top_type)
   THEN (WfLemma 'prod_wf' ORELSE WfLemma 'fun_wf'
         ORELSE WfLemma 'set_wf')
   THEN Try (OnLastHyp SquashE)
  ) p
  ;;

* ML AndWf
  let AndWf p =
   let [t],T = destruct_equal (concl p) in
   if not (is_and_term t & is_gen_universe_term T) then fail
   else
   ( let a,b = destruct_and t in
     Assert (make_equal_term T [make_product_term (undeclared_id p 'x') a b])

     THENL [AbstractAndApplyWfLemma; Trivial]
   ) p
  ;;

* ML ImpWf
  let ImpWf p =
   let [t],T = destruct_equal (concl p) in
   if not (is_implies_term t & is_gen_universe_term T) then fail
   else
   ( let a,b = destruct_implies t in
     Assert (make_equal_term T
              [make_function_term (undeclared_id p 'x') a b])
     THENL [AbstractAndApplyWfLemma; Trivial]
   ) p
  ;;

* ML add_wfers
  map
   (\x. add_to_autotactic x (WfLemma x))
   ''squash_wf or_wf list_wf eq_wf''
  ;;

  add_to_autotactic 'and_wf' AndWf ;;

  add_to_autotactic 'imp_wf' ImpWf ;;

  add_to_autotactic 'AbstractAndApplyWfLemma'
   AbstractAndApplyWfLemma
  ;;
```

```
* THM tight_subst
  >> ∀A:Type. ∀a,b:A. b=a in A => ∀B:{x:A|x=a in A}->Type. B(a) => B(b)

* ML TightSubst
  let TightSubst t p =
  ( let [b;a],A = destruct_equal t in
    let B_on_b = abstract (concl p) b (undeclared_id p `x`) in
    let B = fst (destruct_apply B_on_b) in
    let B_on_a = make_apply_term B a in
    FLemmaUsing `tight_subst` [t; B_on_a]
    THENS FastAp (Reduce THEN Trivial)
    THEN Try (IfThenOnConcl ($= B o first_equand)
                (EqI THENG OnLastHyp E)
             )
    THEN IfThenOnConcl ($= B_on_a) ReduceConcl
  ) p
  ;;


* THM Atom_eq_decidable
  >> ∀a,b:Atom. a=b in Atom ∨ ¬(a=b in Atom)

* DEF elide
  (...)<h:hidden> == <h>

* ML Elide

  let ElideHyp = ComputeHypUsing (\t. instantiate_trivial_def `elide` t) ;;
  let ElideConcl =
    ComputeConclUsing (\t. instantiate_trivial_def `elide` t) ;;
```

C.1.2 Lists

```
* DEF list1
  [<x:*>] == ((<x>).nil)

* DEF list2
  [<x:*>;<y:*>] == ((<x>).(<y>).nil)

* THM null_
  >> object
  Extraction:
```

```
  λl. [ nil → True; h.t,v → False; @ l]

* DEF null
  null == term_of(null_)

* THM null_
  >> ∀i:∞. ∀A:Ui. ∀l:A list. null(l) in Ui

* THM null_lemma
  >> ∀i:∞. ∀A:Ui. ∀l:A list. null(l) => l = nil in A list

* THM null_char
  >> ∀i:∞. ∀A:Ui. ∀l:A list.
       ∀h:A. ∀t:A list. l=h.t in A list => ¬(null(l))
       & null(l) <=> l = nil in A list

* THM null_decidable
  >> ∀i:∞. ∀A:Ui. ∀l:A list. null(l) ∨ ¬(null(l))

* ML add_null_decidable
  add_to_Decidable `null_decidable`
  (Lemma `null_decidable`)
  ;;

* DEF if_null
  if null(<l:A list>) then <b:B> else <c:C> ==
  [ nil → <b>; _._,__ → <c>; @ <l>]

* THM if_null_wf
  >> ∀A,B:Type. ∀l:A list. ∀b:B. ∀f:↓(¬(null(l)))->B.
       if null(l) then b else f(axiom) in B

* ML add_if_null_wf
  add_to_member_i `if_null_wf`
  ( LemmaWithMatchHack `if_null_wf`
    'if null(x) then y else z in T' [`x`;`y`;`z`;`T`]
    [`z`, \t. mtt 1 (fast_ap (substitute '(\x.t)(axiom)') ['t',t])]
  )
  ;;

* THM if_null_char
  >> ∀A,B:Type. ∀l:A list.
       null(l) => ∀a,b:B. if null(l) then a else b = a in B
       & ¬(null(l)) => ∀a,b:B. if null(l) then a else b = b in B

* THM if_null_wf_2
```

```
  >> ∀A,B,C:Type. ∀l:A list. ∀b:B. ∀c:C.
       if null(l) then b else c
       in if null(l) then B else C

* THM if_null_2_
  >> Object
  Extraction:
  λ b f l. if null(l) then b else f(l)

* DEF if_null_2
  if null(<l:A list>) then <b:B> else <f:(A list)+ -\> B> ==
  if_null_2(<b>)(<f>)(<l>)

* THM if_null_2__
  >> ∀j:∞. ∀A,B:Uj. ∀b:B. ∀f: {l:A list|¬(null(l))} -> B .
       ∀l:A list. if null(l) then b else f in B

* THM hd_
  >> object
  Extraction:
  λl. list_ind(l;"uu";h,t,v.h)

* DEF hd
  hd == term_of(hd_)

* THM hd__
  >> ∀A:Type. ∀l:A list. ¬(null(l)) => hd(l) in A

* THM tl_
  >> object
  Extraction:
  λl. list_ind(l;nil;h,t,v.t)

* DEF tl
  tl == term_of(tl_)

* THM tl__
  >> ∀A:Type. ∀l:A list. tl(l) in A list

* THM append_

  >> object
  Extraction:
  λ l1 l2. [ nil → l2; h.t,v → h.v; @ l1]

* DEF append
```

```
<l:list>@<ll:list> == term_of(append_)(<l>)(<ll>)

* THM append__
  >> ∀A:Type. ∀l1,l2:A list. l1@l2 in A list

* THM cons_of_append
  >> ∀A:Type. ∀a:A. ∀x,y:A list.
     a.(x@y) = (a.x)@y in A list

* THM append_assoc
  >> ∀A:Type. ∀x,y,z:A list. x@(y@z) = (x@y)@z in A list

* THM append_to_nil
  >> ∀A:Type. ∀x:A list. x@nil = x in A list

* DEF all_tails
  ∀ tails <h:var>.<t:var> of <l:list>. <P:prop> ==
  list_ind(<l>;True;<h>,<t>,_.<P> & _)

* DEF if_hd
  if <h:var> = hd(<l:list>) then <P:prop> ==
  list_ind(<l>;True;<h>,_,__.<P>)

* THM all_elements_
  >> Object
  Extraction:
  λ A Q l. [ nil → True; h.t,v → Q(h) & v; @ l]

* DEF all_elements
  ∀∈ <l:list> : <A:type> list. <Q:pred> ==
  all_elements(<A>)(<Q>)(<l>)

* THM all_elements__
  >> ∀i:∞. ∀A:Ui. ∀Q:A->Ui. ∀l:A list. (∀∈ l : A list. Q) in Ui

* THM all_elements_mono
  >> ∀i:∞. ∀A:Ui. ∀P,Q:A->Ui. (∀a:A. P(a) => Q(a)) =>
         ∀x:A list. ∀∈ x : A list. P => ∀∈ x : A list. Q

* THM all_elements_of_append
  >> ∀i:∞. ∀A:Ui. ∀P:A->Ui. ∀x,y:A list.
        ∀∈ x : A list. P => ∀∈ y : A list. P => ∀∈ x@y : A list. P

* THM all_elements_of_cons
  >> ∀i:∞. ∀A:Ui. ∀P:A->Ui. ∀a:A. ∀x:A list.
        P(a) => ∀∈ x : A list. P => ∀∈ a.x : A list. P
```

```
* THM append_of_all_elements
  >> ∀i:∞. ∀A:Ui. ∀P:A->Ui. ∀x,y:A list.
        ∀∈ x@y : A list. P => ∀∈ x : A list. P & ∀∈ y : A list. P

* THM cons_of_all_elements
  >> ∀i:∞. ∀A:Ui. ∀P:A->Ui. ∀a:A. ∀x:A list.
        ∀∈ a.x : A list. P => P(a) & ∀∈ x : A list. P

* THM some_element_
  >> Object
  Extraction:
  λ A Q l. [ nil → False; h.t,v → Q(h) ∨ v; @ l]

* DEF some_element
  ∃∈ <l:list> : <A:type> list. <Q:pred> ==
  some_element(<A>)(<Q>)(<l>)

* THM some_element__
  >> ∀i:∞. ∀A:Ui. ∀Q:A->Ui. ∀l:A list. (∃∈ l : A list. Q) in Ui

* THM some_element_mono
  >> ∀i:∞. ∀A:Ui. ∀P,Q:A->Ui. (∀a:A. P(a) => Q(a)) =>
        ∀x:A list. ∃∈ x : A list. P => ∃∈ x : A list. Q

* THM all_elements_decidability
  >> ∀i:∞. ∀A:Ui. ∀Q:A->Ui. ∀x:A. Q(x) ∨ ¬(Q(x))
        => ∀l:A list. (∀∈ l : A list. Q) ∨ ¬(∀∈ l : A list. Q)

* ML add_all_elements_decidablity
  add_to_Decidable `all_elements_decidability`
  (Lemma `all_elements_decidability` THENM ReduceConcl)
  ;;

* THM some_element_decidability
  >> ∀i:∞. ∀A:Ui. ∀Q:A->Ui. ∀x:A. Q(x) ∨ ¬(Q(x))
        => ∀l:A list. (∃∈ l : A list. Q) ∨ ¬(∃∈ l : A list. Q)

* ML add_some_element_decidability
  add_to_Decidable `some_element_decidability`
  (Lemma `some_element_decidability` THENM ReduceConcl)
  ;;

* THM map_
  >> Object
  Extraction:
```

```
  λ B f l. [ nil → nil; h.t,v → f(h).v; @ l]

* DEF map
  (map <f:fn> on <l:list> to <B:Type> list) == map(<B>)(<f>)(<l>)

* THM map__
  >> ∀A,B:Type. ∀f:A->B.
       ∀l:A list. (map f on l to B list) in B list

* THM map_on_empty
  >> ∀i:∞. ∀A,B:Ui. ∀f:A->B. ∀l:A list.
       null(l) => null((map f on l to B list))

* THM map_on_nonempty
  >> ∀i:∞. ∀A,B:Ui. ∀f:A->B. ∀l:A list.
       ¬(null(l)) => ¬(null((map f on l to B list)))

* THM accumulate_
  >> Object
  Extraction:
  λ x f l. [ nil → x; h.t,v → f(h,v); @ l]

* DEF accumulate
  (accumulate <f:fn> over <l:list> from <x:init>) == accumulate(<x>)(<f>)(<l>)


* THM accumulate__
  >> ∀A,B:Type. ∀f:A->B->B. ∀x:B. ∀l:A list.
       (accumulate f over l from x) in B

* THM it_fun_
  >> Object
  Extraction:
  λ f l.
  [ nil → "uu"; h.t,v → if null(t) then h else f(h,v); @ l]

* DEF it_fun
  it_fun(<f:A-\>A-\>A>,<l:A list>) == it_fun(<f>,<l>)

* THM it_fun__
  >> ∀i:∞. ∀A:Ui. ∀f:A->A->A. ∀l:A list. ¬(null(l)) => it_fun(f,l) in A

* THM it_fun_base
  >> ∀i:∞. ∀A:Ui. ∀f:A->A->A. ∀l:A list.
       ¬(null(l)) => null(tl(l))
           => it_fun(f,l) = hd(l) in A
```

```
* THM it_fun_unroll
  >> ∀i:∞. ∀A:Ui. ∀f:A->A->A. ∀l:A list.
       ¬(null(l)) => ¬(null(tl(l)))
            => it_fun(f,l) = f(hd(l),it_fun(f,tl(l))) in A

* THM it_fun_on_cons_base
  >> ∀i:∞. ∀A:Ui. ∀f:A->A->A. ∀h:A. ∀l:A list.
       null(l) => it_fun(f,h.l) = h in A

* THM it_fun_on_cons_unroll
  >> ∀i:∞. ∀A:Ui. ∀f:A->A->A. ∀h:A. ∀l:A list.
       ¬(null(l)) => it_fun(f,h.l) = f(h,it_fun(f,l)) in A

* THM length_
  >> Object
  Extraction:
  λl. [ nil → 0; h.t,v → v+1; @ l]

* DEF length
  |<l:list>| == length(<l>)

* THM length__
  >> ∀A:Type. ∀l:A list. |l| in N

* THM length_of_tl
  >> ∀i:∞. ∀A:Ui. ∀l:A list. ¬(null(l)) => |tl(l)| = |l|-1 in N

* THM length_of_map
  >> ∀A,B:Type. ∀f:A->B. ∀l:A list. |(map f on l to B list)| = |l| in N

* THM not_null_iff_length
  >> ∀i:∞. ∀A:Ui. ∀l:A list. ¬(null(l)) <=> 0<|l|

* THM com_
  >> Object
  Extraction:
  λ l1 l2. [ nil → λl2. nil; h.t,f → λl2. <h,hd(l2)> . f(tl(l2)); @ l1](l2)


* DEF com
  com(<l:A list>,<l1:B list>) == com(<l>)(<l1>)

* THM com__
  >> ∀i:∞. ∀A,B:Ui. ∀l1:A list. ∀l2:B list.
        |l1|=|l2| in N => com(l1,l2) in A#B list
```

```
* THM com_on_empty
  >> ∀i:∞. ∀A,B:Ui. ∀l1:A list. ∀l2:B list.
        |l1|=|l2| in N => ( null(l1) <=> null(com(l1,l2)) )

* THM com_on_nonempty
  >> ∀i:∞. ∀A,B:Ui. ∀l1:A list. ∀l2:B list.
        |l1|=|l2| in N => ( ¬(null(l1)) <=> ¬(null(com(l1,l2))) )

* THM sublist_
  >> Object
  Extraction:
  λ A l1 l2. ∀∈ l1 : A list. λx. ∃∈ l2 : A list. λy. x=y in A

* DEF sublist
  (<l1:A list>⊂<l2:A list> ∈ <A:A:Ui> list) == sublist(<A>)(<l1>)(<l2>)

* THM sublist__
  >> ∀i:∞. ∀A:Ui. ∀l1,l2:A list. (l1⊂l2 ∈ A list) in Ui

* THM some_element_mono_wrt_sublist
  >> ∀i:∞. ∀A:Ui. ∀P:(A->Ui). ∀l1,l2:A list.
      ∃∈ l1 : A list. P => (l1⊂l2 ∈ A list) => ∃∈ l2 : A list. P

* THM sublist_refl
  >> ∀i:∞. ∀A:Ui. ∀l:A list. (l⊂l ∈ A list)

* THM sublist_trans
  >> ∀i:∞. ∀A:Ui. ∀l1,l2,l3:A list.
        (l1⊂l2 ∈ A list) => (l2⊂l3 ∈ A list) => (l1⊂l3 ∈ A list)

* THM N_list_max_
  >> (N list) -> N
  Extraction:
  λl. (accumulate λ i a. max(i,a) over l from 0)

* DEF N_list_max
  max(<l:N list>) == N_list_max(<l>)

* THM List_inclusion
  >> ∀A:Type. ∀Q:A->Type. ∀l:{z:A|Q(z)} list. l in A list

* THM List_inclusion_2
  >> ∀i:∞. ∀A:Ui. ∀Q:A->Ui. ∀l:{z:A|Q(z)} list.
      l in { l1:A list | ∀∈ l1 : A list. Q}
```

```
* THM List_inclusion_3
  >> ∀i:∞. ∀A:Ui. ∀Q:A->Ui. ∀l:{ l:A list | ∀∈ l : A list. Q }.
       l in {z:A|Q(z)} list

* THM apply2_
  >> Object
  Extraction:
  λ B f l. f(hd(l),hd(tl(l)))

* DEF apply2
  apply2{<B:B:Type>}(<f:A-\>A-\>B>,<l:A list>) == apply2(<B>)(<f>)(<l>)

* THM apply2__
  >> ∀A,B:Type. ∀f:A->A->B. ∀l:A list. |l|=2 in N => apply2{B}(f,l) in B

* THM list2_elim
  >> ∀A:Type. ∀l:A list. |l|=2 in N => ∀P:{l:A list||l|=2 in N}->Type.
       (∀a,b:A. P([a;b])) => P(l)

* ML List2E
  let List2E l =
    AbstractConcl l THEN Lemma `list2_elim`
  ;;

* THM parallel_list_elim
  >> ∀A,B:Type. ∀P: l1:A list -> {l2:B list||l1|=|l2| in N} -> Type .
       P(nil,nil) =>
       ( ∀h1:A. ∀h2:B. ∀t1:A list. ∀t2:B list.
           |t1|=|t2| in N => P(t1,t2) => P(h1.t1, h2.t2) ) =>
       ∀l1:A list. ∀l2:B list. |l1|=|l2| in N => P(l1,l2)

* THM list_rec_1_
  >> Object
  Extraction:
  λ b g. λl. [ nil → "uu"; h.t,v → if null(t) then b(h) else g(h,t,v); @ l]


* DEF list_rec_1
  ([a] → <b:A-\>*>(a); <g:A-\>(A list)-\>*-\>*>; @ <a:A list>) ==
  list_rec_1(<b>)(<g>)(<a>)

* THM list_rec_1__
  >> ∀j:∞. ∀A:Uj. ∀B: {l:A list|¬(null(l))} -> Uj . ∀b:(a:A->B([a])).
       ∀g: h:A -> t:{l:A list|¬(null(l))} -> B(t) -> B(h.t) .  ∀l:A list.
       ¬(null(l)) => ([a] → b(a); g; @ l) in B(l)
```

```
* THM list_subset_membership
  >> ∀j:∞. ∀A:Uj. ∀P:A->Uj. ∀x:{x:A|P(x)} list. ∀y:A list.
       x=y in A list => y in {x:A|P(x)} list

* THM eq_lists_if_eq_in_superset
  >> ∀j:∞. ∀A:Uj. ∀P:A->Uj. ∀x:{x:A|P(x)} list. ∀y:A list.
       x=y in A list => x=y in {x:A|P(x)} list

* THM list_eq_decidable
  >> ∀A:Type. ∀a1,a2:A. a1=a2 in A  ∨ ¬(a1=a2 in A)
        => ∀l1,l2:A list. l1=l2 in A list ∨ ¬(l1=l2 in A list)

* THM Atom_list_eq_decidable
  >> ∀l1,l2:Atom list. l1=l2 in Atom list ∨ ¬(l1=l2 in Atom list)

* THM Int_list_eq_decidable
  >> ∀l1,l2:Int list. l1=l2 in Int list ∨ ¬(l1=l2 in Int list)


* ML ListUnroll2
  let ListUnroll2 t p =
    let T = get_using_type p t in
    if not is_list_term T then failwith `ListUnroll2: not list type` ;
  ( Decide (make_apply_term 'null' t)
    THENS Assert (make_equal_term T [t]) % efficiency %
    THENS (OnNthLastHyp 2 (\i. BringHyps [i]) THEN ETermUsing t T
           THENM UnrollDefsInConcl ``null`` THENM I
           THENW Try (OnLastHyp
                        (\i. IfThenOnHyp ($= '¬(True)' o snd) (E i) i))
           THENW Try Trivial)
    THEN Try (ThinToEnd (number_of_hyps p + 2))
  ) p
  ;;

* ML TrivEqIListInd
  % Elim only the term occurrence in the principal arg. %
  let TrivEqIListInd p =
  ( let equands,T = destruct_equal (concl p) in
    let l = hd (subterms (hd equands)) in
    let z = undeclared_id p `z` in
    let new_equands =
      map (\t. let a,b,c = destruct_list_induction t in
               make_list_induction_term (l=a => mvt z | a) b c )
          equands  in
    let Elim t =
      ETerm t THENM UnrollDefsInConcl ``null`` THENM I THENW
```

```
    OnLastHyp (\i. IfThenOnHyp ($= '¬(True)' o snd) (E i) i)
    THENW Try Trivial
    THEN Try (ThinToEnd (number_of_hyps p + 2)) in
  Decide (make_apply_term 'null' l) THENS
  (\p. let A =
         replace_subterm l (mvt z) (type_of_declaration (last (hyps p))) in

       let B = make_implies_term A (make_equal_term T new_equands) in
       (OnLastHyp (\i. BringHyps [i]) THEN ApplyToOcc Elim z B) p
  )
) p
;;
```

C.1.3 Sets

Basic Definitions

```
* THM trans_
  >> i:∞ -> A:Ui -> r:((A#A)->Ui) -> Ui
  Extraction:
  λ i A r. ∀x,y,z:A. r(x,y) => r(y,z) => r(x,z)

* DEF trans
  trans{<A>:U<i:int>}(<r:reln>) == trans(<i>)(<A>)(<r>)

* THM sym_
  >> i:∞ -> A:Ui -> r:((A#A)->Ui) -> Ui
  Extraction:
  λ i A r. ∀x,y:A. r(x,y) => r(y,x)

* DEF sym
  sym{<A>:U<i:int>}(<r:reln>) == sym(<i>)(<A>)(<r>)

* THM refl_
  >> i:∞ -> A:Ui -> r:((A#A)->Ui) -> Ui
  Extraction:
  λ i A r. ∀x:A. r(x,x)

* DEF refl
  refl{<A>:U<i:int>}(<r:reln>) == refl(<i>)(<A>)(<r>)

* THM eq_reln_
  >> i:∞ -> A:Ui -> r:((A#A)->Ui) -> Ui
```

```
  Extraction:
  λ i A r. refl{A:Ui}(r) & sym{A:Ui}(r) & trans{A:Ui}(r)

* DEF eq_reln
  eq_reln{<A>:U<i:int>}(<r:reln>) == eq_reln(<i>)(<A>)(<r>)

* THM Seti_
  >> ∞ -> Type
  Extraction:
  λi. A:Ui # { r:(A#A)->Ui | eq_reln{A:Ui}(r) }

* DEF Seti
  Set(<i:∞>) == Seti(<i>)

* THM tos_
  >> Object
  Extraction:
  λS. S.1

* DEF tos
  |<S:Set(i)>| == tos(<S>)

* THM tos__
  >> ∀i:∞. ∀S:Set(i). |S| in Ui

* THM eos_
  >> Object
  Extraction:
  λS. S.2

* DEF eos
  =_{<S:S:Set(i)>} == eos(<S>)

* THM eos__
  >> ∀i:∞. ∀S:Set(i). =_{S} in (|S|#|S|) -> Ui

* THM eos_eq_reln
  >> ∀i:∞. ∀S:Set(i). ↓(eq_reln{|S|:Ui}(=_{S}))

* DEF seq
  <a:S>=<b:S> in <S:S:Set(i)> == =_{<S>}(<a>,<b>)

* ML seq_adder
  add_matching_def_adder `seq`
  'eos(x)(<y,z>)' ``y z x`` (\l. true)
  ;;
```

Constructors

```
* THM fnl_
  >> Object
  Extraction:
  λ S1 S2 f. ∀x,y:|S1|. x=y in S1 => f(x)=f(y) in S2

* DEF fnl
  fnl{<S1:S1:Set(i)>,<S2:S2:Set(i)>}(<f:|S1|-\>|S2|>) == fnl(<S1>)(<S2>)(<f>)

* THM fnl__
  >> ∀i:∞. ∀S1,S2:Set(i). ∀f:|S1|->|S2|. fnl{S1,S2}(f) in Ui

* THM fun_type_
  >> Object
  Extraction:
  λ S1 S2. { f:|S1|->|S2| | fnl{S1,S2}(f) }

* DEF fun_type
  fun_type(<S:Set(i)>,<SS:Set(i)>) == fun_type(<S>)(<SS>)

* THM fun_type__
  >> ∀i:∞. ∀S1,S2:Set(i). fun_type(S1,S2) in Ui

* THM fun_eq_
  >> Object
  Extraction:
  λ S1 S2. λ f,g. ∀x:|S1|. f(x)=g(x) in S2

* DEF fun_eq
  fun_eq(<S:Set(i)>,<SS:Set(i)>) == fun_eq(<S>)(<SS>)

* THM fun_eq__
  >> ∀i:∞. ∀S1,S2:Set(i).
       fun_eq(S1,S2) in (fun_type(S1,S2)#fun_type(S1,S2)) -> Ui

* THM fun_eq_eq_reln
  >> ∀i:∞. ∀S1,S2:Set(i).
       ↓( eq_reln{fun_type(S1,S2):Ui}(fun_eq(S1,S2)) )

* THM fun_
```

```
  >> Object
  Extraction:
  λ S1 S2. <fun_type(S1,S2), fun_eq(S1,S2)>

* DEF fun
  <S:Set(i)>-\><SS:Set(i)> == fun(<S>)(<SS>)

* THM fun_
  >> ∀i:∞. ∀S1,S2:Set(i). S1->S2 in Set(i)

* THM prod_type_
  >> Object
  Extraction:
  λ S1 S2. |S1|#|S2|

* DEF prod_type
  prod_type(<S:Set(i)>,<SS:Set(i)>) == prod_type(<S>)(<SS>)

* THM prod_type_
  >> ∀i:∞. ∀S1,S2:Set(i). prod_type(S1,S2) in Ui

* THM prod_eq_
  >> Object
  Extraction:
  λ S1 S2. λ x,y. x.1=y.1 in S1 & x.2=y.2 in S2

* DEF prod_eq
  prod_eq(<S:Set(i)>,<SS:Set(i)>) == prod_eq(<S>)(<SS>)

* THM prod_eq_
  >> ∀i:∞. ∀S1,S2:Set(i).
       prod_eq(S1,S2) in (prod_type(S1,S2)#prod_type(S1,S2)) -> Ui

* THM prod_eq_eq_reln
  >> ∀i:∞. ∀S1,S2:Set(i).
       ↓( eq_reln{prod_type(S1,S2):Ui}(prod_eq(S1,S2)) )

* THM prod_
  >> Object
  Extraction:
  λ S1 S2. <prod_type(S1,S2), prod_eq(S1,S2)>

* DEF prod
  <S:Set(i)>#<SS:Set(i)> == prod(<S>)(<SS>)

* THM prod_
```

```
  >> ∀i:∞. ∀S1,S2:Set(i). S1#S2 in Set(i)

* THM tos_on_prod
  >> ∀i:∞. ∀S1,S2:Set(i). |S1#S2| = |S1| # |S2| in Ui
```

Some Predicates

The prefix "full" in "full_injection" and "full_subset" should be ignored. It is only present for historical reasons.

```
* THM injective_
  >> Object
  Extraction:
  λ S1 S2 i. ∀x,y:|S1|. i(x)=i(y) in S2 => x=y in S1

* DEF injective
  injective(<i:*>∈<S1:Set(i)>-\><S2:Set(i)>) == injective(<S1>)(<S2>)(<i>)

* THM injective__
  >> ∀j:∞. ∀S1,S2:Set(j). ∀i:|S1->S2|. injective(i∈S1->S2) in Uj

* THM subset_
  >> Object
  Extraction:
  λ S1 S2. {i:|S1->S2||injective(i∈S1->S2)}

* DEF subset
  <S:Set(i)>⊂<SS:Set(i)> == subset(<S>)(<SS>)

* THM subset__
  >> ∀i:∞. ∀S1,S2:Set(i). S1⊂S2 in Ui

* THM subtype_
  >> Object
  Extraction:
  λ S1 S2.
  ∃P:|S2->Prop| where
    |S1| = {x:|S2||P(x)} in SET_U
    & ∀x,y:|S1|. x=y in S1 <=> x=y in S2

* DEF subtype
  <S:SET>⊂⊂<SS:SET> == subtype(<S>)(<SS>)

* THM subtype__
```

```
  >> ∀S1,S2:SET. S1⊂⊂S2 in U

* THM member_
  >> Object
  Extraction:
  λ S1 S2 i. λy. ∃x:|S1| where y=i(x) in S2

* DEF member
  <y:S2> ∈ <S1:S1:Set(i)> ⊂{<i:S1-\>S2>} <S2:S2:Set(i)> ==
  member(<S1>)(<S2>)(<i>)(<y>)

* THM member__
  >> ∀j:∞. ∀S1,S2:Set(j). ∀i:S1⊂S2. ∀y:|S2|. (y ∈ S1 ⊂{i} S2) in Uj

* THM full_subset_
  >> Object
  Extraction:
  λ S1 S2 i. ∀y:|S2|. y ∈ S1 ⊂{i} S2 => ∃x:|S1| where y=i(x) in |S2|

* DEF full_subset
  full(<S1:S1:Set(i)> ⊂{<i:S1-\>S2>} <S2:S2:Set(i)>) ==
  full_subset(<S1>)(<S2>)(<i>)

* THM full_subset__
  >> ∀j:∞. ∀S1,S2:Set(j). ∀i:S1⊂S2. (full(S1 ⊂{i} S2)) in Uj

* THM full_injection_
  >> Object
  Extraction:
  λ S1 S2 i. ↓(injective(i∈S1->S2))

* DEF full_injection
  full_injection(<i:*>∈<S1:Set(i)>-\><S2:Set(i)>) ==
  full_injection(<S1>)(<S2>)(<i>)

* THM full_injection__
  >> ∀j:∞. ∀S1,S2:Set(j). ∀i:|S1->S2|. full_injection(i∈S1->S2) in Uj
```

Special Sets

```
* DEF Top
  U == U8
```

```
* DEF Top_level
  U_level == 8

* DEF Topp
  U' == U9

* THM SET_
  >> U
  Extraction:
  Set(7)

* DEF SET
  SET == SET

* DEF SET_U
  SET_U == U7

* DEF SET_level
  i_SET == 7

* THM Set_
  >> U
  Extraction:
  Set(6)

* DEF Set
  Set == Set

* THM Set1_contained_in_Set
  >> ∀S:Set(1). S in Set

* THM Set_contained_in_SET
  >> ∀S:Set. S in SET

* ML add_Set_inclusions
  add_to_inclusion `Set1_contained_in_Set`
  (\i. Lemma `Set1_contained_in_Set` THEN Trivial) ;;

  add_to_inclusion `Set_contained_in_SET`
  (\i. Lemma `Set_contained_in_SET` THEN Trivial) ;;

* THM SET_eq_if_Set_eq
  >> ∀S1,S2:Set. S1=S2 in Set => S1=S2 in SET

* THM eos_of_eq_sets
  >> ∀S1,S2:SET. ∀x,y:|S1|. S1=S2 in SET => x=y in S1 => x=y in S2
```

```
* THM Prop_
  >> SET
  Extraction:
  <U6,λ P,Q. P <=> Q>

* DEF Prop
  Prop == Prop

* THM SmallEqSET_
  >> U
  Extraction:
  A:U7 # { r: (A#A)->U6 | eq_reln{A:U7}(r) }

* DEF SmallEqSET
  SmallEqSET == SmallEqSET

* THM eos_on_SmallEqSET
  >> ∀S:SmallEqSET. =_{S} in (|S|#|S|) -> |Prop|

* THM SmallEqSET_contained_in_SET
  >> ∀S:SmallEqSET. S in SET

* THM Set_contained_in_SmallEqSET
  >> ∀S:Set. S in SmallEqSET

* THM Prop_small_type
  >> Prop in SmallEqSET
```

Further Constructors

```
* THM sot_
  >> Object
  Extraction:
  λT. <T, λ x,y. (x=y in T)>

* DEF sot
  ↑<T:Type> == sot(<T>)

* THM sot__
  >> ∀i:∞. ∀T:Ui. ↑T in Set(i)

* THM prod_of_list_
```

```
  >> Object
  Extraction:
  λl. it_fun(λ S1 S2. S1#S2, l)

* DEF prod_of_list
  #(<l:SET list>) == prod_of_list(<l>)

* THM prod_of_list_
  >> ∀Sl:SET list. ¬(null(Sl)) => (#(Sl) in SET)

* THM prod_of_list_base
  >> ∀Sl:SET list. ∀S:SET. null(Sl) => (#(S.Sl)=S in SET)

* THM prod_of_list_base_mem
  >> ∀Sl:SET list. ∀S:SET. null(Sl) => ∀x:|S|. x in |#(S.Sl)|

* THM prod_of_list_unroll
  >> ∀Sl:SET list. ∀S:SET. ¬(null(Sl)) => (#(S.Sl)=S#(#(Sl)) in SET)

* THM unroll_tos_on_prod_of_list
  >> ∀Sl:SET list. ∀S:SET. ¬(null(Sl)) => ∀x:|S|#|#(Sl)|.  x in |#(S.Sl)|

* THM roll_tos_on_prod_of_list
  >> ∀Sl:SET list. ∀S:SET. ¬(null(Sl)) => ∀x:|#(S.Sl)|.  x in |S|#|#(Sl)|
```

C.1.4 Representing Terms

The Type of Meta-Terms

```
* DEF letrec
  letrec <h:var>(<z:var>) = <b:body> in <a:term> ==
  let <h> = λ<z>. rec_ind(<z>;<h>,<z>.<b>) in <a>

* DEF Atom
  Atom == Atom

* THM Term0_
  >> U1
  Extraction:
  rec(T. (Atom#T list) | (T#T#Atom) | (Atom#T#T) | Int )

* DEF Term0
  Term0 == Term0
```

```
* THM inf_
  >> Atom -> Term0 list -> Term0
  Extraction:
  λ f args. inl(<f,args>)

* DEF inf
  <f:Atom>(<args:Term0 list>) == inf(<f>)(<args>)

* THM ine_
  >> Term0 -> Term0 -> Atom -> Term0
  Extraction:
  λ a b A. inr(inl(<a,b,A>))

* DEF ine
  <a:Term0>=<b:Term0> in <A:Atom> == ine(<a>)(<b>)(<A>)

* THM ini_
  >> Atom -> Term0 -> Term0 -> Term0
  Extraction:
  λ i a b. inr(inr(inl(<i,a,b>)))

* DEF ini
  <b:b:Term0>{<i:Atom> <a:a:Term0>} == ini(<i>)(<a>)(<b>)

* THM inn_
  >> Int -> Term0
  Extraction:
  λn. inr(inr(inr(n)))

* DEF inn
  <n:Int> == inn(<n>)

* ML add_Term0_list_subset_hack
  add_to_inclusion `Term0_list_subset_hack`
  (\i p.
    if eq_type (concl p) = 'Term0 list'
       & is_set_term (fake_compute_ap (destruct_list (h i p)))
    then (E i THEN (Id ...*)) p
    else fail
  )
  ;;
```

Analyzing Meta-Terms

```
* THM Term0_hereditary_
  >> Object
  Extraction:
  λP.
  ∀f:Atom. ∀l:Term0 list. P(f(l)) => ∀∈ l : Term0 list. P   &
  ∀t,u:Term0. ∀A:Atom. P(t=u in A) => P(t) & P(u) &
  ∀i:Atom. ∀t,u:Term0. P(u{i t}) => P(t) & P(u)

* DEF Term0_hereditary
  (<P:Term0-\>Ui> hereditary over Term0) == Term0_hereditary(<P>)

* THM Term0_hereditary__
  >> ∀j:∞. ∀P:Term0->Uj. (P hereditary over Term0) in Uj

* THM Term0_induction
  >> ∀j:∞. ∀P:Term0->Uj.
          ∀f:Atom. ∀l:Term0 list. ∀∈ l : Term0 list. P => P(f(l))
       => ∀t,u:Term0. ∀A:Atom. P(t) => P(u) => P(t=u in A)
       => ∀i:Atom. ∀t,u:Term0. P(t) => P(u) => P(u{i t})
       => ∀n:Int. P(n)
       => ∀t:Term0. P(t)

* THM Term0_induction_2
  >> ∀P:Term0->U.
          ∀f:Atom. ∀l:Term0 list. ∀∈ l : Term0 list. P => P(f(l))
       => ∀t,u:Term0. ∀A:Atom. P(t) => P(u) => P(t=u in A)
       => ∀i:Atom. ∀t,u:Term0. P(t) => P(u) => P(u{i t})
       => ∀n:Int. P(n)
       => ∀t:Term0. P(t)

* ML Term0Induction
  set_d_tactic_args 1 [] [] ``P x h z f l u v A i u v n`` ;;

  let Term0Induction i p =
    Pattern `Term0_induction_pattern` [] []
     (new_ids_from_ids ``P x h z f l u v A i u v n`` p) i p
  ;;

# THM Term0_induction_pattern
  >> ∀s:Term0. "G"

* ML Term0Unroll
  set_d_tactic_args 1 [] [] ``z f l u v A i u v n`` ;;
```

```
  let Term0Unroll i p =
    Pattern `Term0_unroll_pattern` [] []
     (new_ids_from_ids ``z f l u v A i u v n`` p) i p
  ;;

# THM Term0_unroll_pattern
  >> ∀s:Term0. "G"

* THM Term0_heredity_under_squash
  >> ∀j:∞. ∀P:Term0->Uj.
       (P hereditary over Term0) => (λx. ↓(P(x)) hereditary over Term0)

* THM Term0_cases_
  >> ∀A:Type. (Atom->(Term0 list)->A) -> (Term0->Term0->Atom->A)
       -> (Atom->Term0->Term0->A) -> (Int->A) -> Term0 -> A
  Extraction:
  λA. λe. λ f g h.
  λt. d(t; u. (let a,l = u in e(a,l));
           u. d(u; v.let x,y,A = v in f(x,y,A);
                   v.d(v; w.let i,x,y = w in g(i,x,y); w.h(w))))

* DEF Term0_cases
  case <t:Term0> to <A:Type>
    <e:Atom-\>(Term0 list)-\>A>
    <f:Term0-\>Term0-\>Atom-\>A>
    <g:Atom-\>Term0-\>Term0-\>A>
    <h:Int-\>A> ==
  Term0_cases(<A>)(<e>)(<f>)(<g>)(<h>)(<t>)

* THM Term0_subset_induction
  >> ∀j:∞. ∀P:Term0->Uj. ∀Q:{x:Term0|P(x)}->Uj.
         (P hereditary over Term0)
       => ∀f:Atom. ∀l:{x:Term0|P(x)} list.
             ∀∈ l : {x:Term0|P(x)} list. Q => ↓(P(f(l))) => Q(f(l))
       => ∀t,u:{x:Term0|P(x)}. ∀A:Atom. Q(t) => Q(u) => ↓(P(t=u in A))
                                                  => Q(t=u in A)
       => ∀i:Atom. ∀t,u:{x:Term0|P(x)}. Q(t) => Q(u) => ↓(P(u{i t}))
                                                  => Q(u{i t})
       => ∀n:Int. ↓(P(n)) => Q(n)
       => ∀t:{x:Term0|P(x)}. Q(t)

* ML Term0SubsetUnroll
  set_d_tactic_args 1 [] [] ``f l u v A i u v n`` ;;

  let Term0SubsetUnroll i p =
```

```
    Pattern 'Term0_subset_unroll' [] []
     (new_ids_from_ids ``f l u v A i u v n`` p) i p
  ;;

# THM Term0_subset_unroll
  >> ∀s:{s:Term0|"P"(s)}. "G"

* ML Term0SubsetInduction
  set_d_tactic_args 1 [] [] ``P x h z f l u v A i u v n`` ;;

  let Term0SubsetInduction i p =
    Pattern 'Term0_subset_induction_pattern' [] []
     (new_ids_from_ids ``P x h z f l u v A i u v n`` p) i p
  ;;

# THM Term0_subset_induction_pattern
  >> ∀s:{s:Term0|"P"(s)}. "G"

* ML Term0EqI
  set_d_tactic_args 0 [] [] ``P h z f l u v A i u v n`` ;;

  let Term0EqI p =
    Pattern 'Term0_EqI_pattern' [] []
     (new_ids_from_ids ``P h z f l u v A i u v n`` p) 0 p
  ;;

# THM Term0_EqI_pattern
  >> rec_ind("s";h,z."d"(h,z)) in "T"("s")

* ML Term0SubsetEqI
  set_d_tactic_args 1 ['λt. "Q"(t)'] [] ``P h z f l u v A i u v n`` ;;

  let Term0SubsetEqI i p =
    Pattern 'Term0_subset_EqI_pattern' [] []
     (new_ids_from_ids ``P h z f l u v A i u v n`` p) i p
  ;;

# THM Term0_subset_EqI_pattern
  >> rec_ind("s";h,z."d"(h,z)) in "T"("s")

* THM size_
  >> Term0 -> N
  Extraction:
  λt. rec_ind(t; s,x.
  case x to N
    f,l -> 1 + max((map s on l to N list))
```

```
    x,y,A -> 1 + max(s(x),s(y))
    i,x,y -> 1 + max(s(x),s(y))
    n -> 1)

* DEF size
  |<t:Term0>| == size(<t>)

* THM N_induction
  >> ∀A:Type. ∀Q:A->Type. ∀s:A->N.
       ∀u:A. ( ∀t:A. s(t)<s(u) => Q(t) ) => Q(u) =>
       ∀u:A. Q(u)

* THM size_induction
  >> ∀Q:Term0->Type.
       ∀u:Term0. ( ∀t:Term0. |t|<|u| => Q(t) ) => Q(u) =>
       ∀u:Term0. Q(u)

* THM sub_map_
  >> (Term0->Term0) -> Term0 -> Term0
  Extraction:
  λg. λt. case t to Term0
    f,args -> f((map g on args to Term0 list))
    x,y,A  -> g(x)=g(y) in A
    i,x,y  -> g(y){i g(x)}
    n      -> n

* DEF sub_map
  sub_map(<f:Term0-\>Term0>,<t:Term0>) == sub_map(<f>)(<t>)

* ML UnrollTermFun
  let UnrollTermFunInConcl name =
    UnrollRecInConcl name THEN UnfoldsInConcl ``Term0_cases inf ine ini inn``

    THEN EvalConclOnly [] THEN RedefConcl
  ;;

  let UnrollTermFunInHyp name i =
    UnrollRecInHyp name i THEN UnfoldsInHyp ``Term0_cases inf ine ini inn`` i

    THEN EvalHypOnly [] i THEN RedefHyp i
  ;;

  let UnrollTermFun name =
    TryEverywhere (UnrollTermFunInHyp name) (UnrollTermFunInConcl name)
  ;;
```

C.1.5 Failure and Association Lists

Failure, Success, and Failure Trapping

```
* DEF Fail
  ?<T:Type> == <T> ∨ True

* DEF fail
  fail == inr(axiom)

* DEF succeed
  s(<x:A>) == inl(<x>)

* DEF catch
  <t:?A>?<tt:?A> == d(<t>;u.inl(u);v.<tt>)

* DEF fails
  fails(<x:?A>) == isr(<x>)

* ML add_fails
  add_matching_def_adder `fails`
  'isr(x)' ``x`` (\l.true)
  ;;

* DEF succeeds
  succeeds(<x:?A>) == isl(<x>)

* ML add_succeeds
  add_matching_def_adder `fails`
  'isr(x)' ``x`` (\l.true)
  ;;

* DEF flet
  let (s(<x:var:A>)=<a:?A>) ? <c:B> in <d:B> ==
  let _ = <a> in d(_;<x>.<d>;__.<c>)

* DEF with
  with s(<x:var:A>)=<a:?A> . <d:Prop> == let (s(<x>)=<a>) ? False in <d>

* DEF ifs
  if s(<x:var:A>)=<a:?A> . <d:Prop> == let (s(<x>)=<a>) ? True in <d>
```

```
* THM fails_or_succeeds
  >> ∀A:Type. ∀x:?A. fails(x) ∨ succeeds(x)

* THM succeeds_iff_not_fails
  >> ∀i:∞. ∀A:Ui. ∀x:?A. succeeds(x) <=> ¬(fails(x))

* THM catch_fails
  >> ∀A:Type. ∀x,y:?A. fails(x?y) <=> fails(x) & fails(y)

* THM catch_succeeds
  >> ∀A:Type. ∀x,y:?A. succeeds(x?y) <=> succeeds(x) ∨ succeeds(y)

* THM fails_char
  >> ∀A:Type. ∀x:?A. fails(x) <=> ∃u:True where x=inr(u) in ?A

* THM catch_char
  >> ∀A:Type. ∀a,b:?A.
           succeeds(a) & outl(a?b) = outl(a) in A
         ∨ fails(a) & succeeds(b) & outl(a?b) = outl(b) in A
         ∨ fails(a) & fails(b) & fails(a?b)
```

Association Lists

```
* THM alist_ap_
  >> ∀A:Type. ∀l:(Atom#A) list. ∀x:Atom. ?A
  Extraction:
  λ A l x. [ nil → inr(axiom); h.t,v → if h.1=x then inl(h.2) else v; @ l]


* DEF alist_ap
  <l:(Atom#A) list>{<A:A:Type>}(<x:Atom>) == alist_ap(<A>,<l>,<x>)

* THM alist_membership_decidable
  >> ∀i:∞. ∀A:Ui. ∀a:Atom. ∀l:(Atom # A) list.
         ∃∈ l : Atom#A list. λx. a=x.1 in Atom
         ∨ ¬(∃∈ l : Atom#A list. λx. a=x.1 in Atom)

* THM fails_iff_unbound
  >> ∀i:∞. ∀A:Ui. ∀x:Atom. ∀l:Atom#A list.
         fails(l{A}(x)) <=> ¬(∃∈ l : Atom#A list. λu. x=u.1 in Atom)

* THM succeeds_iff_bound
  >> ∀i:∞. ∀A:Ui. ∀x:Atom. ∀l:Atom#A list.
```

```
      succeeds(l{A}(x)) <=> ∃∈ l : Atom#A list. λu. x=u.1 in Atom

* THM with_char
  >> ∀A:Type. ∀P:A->Type. ∀a:?A. with s(x)=a . P(x) =>
        succeeds(a) => P(outl(a))
      & ¬(fails(a))

* THM ifs_char
  >> ∀A:Type. ∀P:A->Type. ∀a:?A. if s(x)=a . P(x) => succeeds(a) => P(outl(a))


* THM ifs_with_lemma
  >> ∀A:Type. ∀P:A->A->Type. ∀a,b:?A. if s(x)=a . with s(y)=b . P(x,y)
        => succeeds(a) => (succeeds(b) & P(outl(a),outl(b)))

* THM sub_alist_
  >> Object
  Extraction:
  λ A l1 l2.
  ∀a:Atom.
    if s(x)=l1{A}(a) .
    with s(y)=l2{A}(a) .
    x=y in A

* DEF sub_alist
  (<l1>⊂<l2> ∈ (Atom#<A:A:Ui> list)) == sub_alist(<A>)(<l1>)(<l2>)

* THM sub_alist__
  >> ∀i:∞. ∀A:Ui. ∀l1,l2:Atom#A list. (l1⊂l2 ∈ (Atom#A list)) in Ui

* THM sub_alist_char
  >> ∀i:∞. ∀A:Ui. ∀l1,l2:Atom#A list. (l1⊂l2 ∈ (Atom#A list)) => ∀a:Atom.

        succeeds(l1{A}(a))
        => succeeds(l2{A}(a)) & outl(l1{A}(a)) = outl(l2{A}(a)) in A

* THM alist_ap_on_cons
  >> ∀i:∞. ∀A:Ui. ∀l:Atom#A list. ∀x:Atom#A. ∀a:Atom.
      x.1=a in Atom => x.l{A}(a) = s(x.2) in ?A
      & ¬(x.1=a in Atom) => x.l{A}(a) = l{A}(a) in ?A

* THM sub_alist_of_cons
  >> ∀i:∞. ∀A:Ui. ∀l1,l2:Atom#A list. ∀x:Atom#A.
      ( if s(a)=l1{A}(x.1) . x.2=a in A )
      => (l1⊂l2 ∈ (Atom#A list)) => (l1⊂x.l2 ∈ (Atom#A list))
```

```
* THM cons_sub_alist
  >> ∀i:∞. ∀A:Ui. ∀l1,l2:Atom#A list. ∀x:Atom#A.
       ( with s(a)=l2{A}(x.1) . x.2=a in A )
       => (l1⊂l2 ∈ (Atom#A list)) => (x.l1⊂l2 ∈ (Atom#A list))

* THM sub_alist_if_all_elements
  >> ∀i:∞. ∀A:Ui. ∀l1,l2:Atom#A list.
       ∀∈ l1 : Atom#A list. λu. (with s(v)=l2{A}(u.1) . u.2=v in A)
       => (l1⊂l2 ∈ (Atom#A list))

* THM sub_alist_refl
  >> ∀i:∞. ∀A:Ui. ∀l:Atom#A list. (l⊂l ∈ (Atom#A list))

* THM sub_alist_trans
  >> ∀i:∞. ∀A:Ui. ∀l1,l2,l3:Atom#A list.
        (l1⊂l2 ∈ (Atom#A list)) => (l2⊂l3 ∈ (Atom#A list))
                                  => (l1⊂l3 ∈ (Atom#A list))

* THM lemma_for_alist_ap_mono_lemma
  >> ∀A:Type. ∀a,b,c:?A. (if s(x)=b . with s(y)=c . x=y in A) =>
       succeeds(a) ∨ succeeds(b)  => outl(a?b) = outl(a?c) in A

* THM alist_ap_mono_lemma
  >> ∀j:∞. ∀A:Uj. ∀l0,l1,l2:Atom#A list. (l1⊂l2 ∈ (Atom#A list)) =>
       ∀a:Atom. ∃∈ l0 : Atom#A list. λu. a=u.1 in Atom
                ∨ ∃∈ l1 : Atom#A list. λu. a=u.1 in Atom
       => outl(l0{A}(a)?l1{A}(a)) = outl(l0{A}(a)?l2{A}(a)) in A
```

Building Functions That May Fail

```
* THM slet_
  >> Object
  Extraction:
  λB. λ f a. d(a;x.f(x);x.fail)

* DEF fap
  <f:A-\>?B>(<a:A>): <B:B:Type> == slet(<B>)(<f>)(<a>)

* DEF slet
  let s(<x:var>) = <a:?A> in <t:*>: ?<B:Type> == slet(<B>)(λ<x>. <t>)(<a>)

* THM slet__
  >> ∀A,B:Type. ∀a:?A. ∀f:A->?B. slet(B,f,a) in ?B
```

```
* THM slet2_
  >> Object
  Extraction:
  λB. λ f a b. let s(x) = a in let s(y) = b in f(x,y): ?B: ?B

* DEF slet2
  let s(<x:var>), s(<y:var>) = <a:?A>, <b:?B> in <t:*>: ?<C:Type> ==
  slet2(<C>)(λ <x> <y>. <t>)(<a>)(<b>)

* THM slet2__
  >> ∀A,B,C:Type. ∀a:?A. ∀b:?B. ∀f:A->B->?C. slet2(C,f,a,b) in ?C

* THM fmap_
  >> Object
  Extraction:
  λ B f l. [ nil → s(nil);
             h.t,v → let s(x), s(y) = v, f(h) in s(y.x): ?(B list); @ l]

* DEF fmap
  (map <f:A-\>?B> on <l:A list> to ?(<B:B:Type> list)) == fmap(<B>)(<f>)(<l>)


* THM fmap__
  >> ∀A,B:Type. ∀f:A->?B.
       ∀l:A list. (map f on l to ?(B list)) in ?B list

* THM ds_
  >> Object
  Extraction:
  λa. d(a;x.x;x."uu")

* DEF ds
  ds(<a:?A>) == ds(<a>)

* THM ds__
  >> ∀A:Type. ∀a:?A. succeeds(a) => ds(a) in A

* THM ifs_i
  >> ∀A:Type. ∀P:A->Type. ∀a:?A. (succeeds(a) => P(ds(a))) => if s(x)=a . P(x)


* THM fmap_success
  >> ∀A,B:U. ∀f:A->?B. ∀l:A list.
       succeeds((map f on l to ?(B list)))
       <=> ∀∈ l : A list. λx. succeeds(f(x))
```

```
* THM fmap_char
  >> ∀A,B:U. ∀f:A->?B.
       ∀l:A list. succeeds((map f on l to ?(B list)))
       => ds((map f on l to ?(B list))) = (map λx. ds(f(x)) on l to B list)
          in B list

* THM ds_on_slet
  >> ∀A,B:Type. ∀a:?A. ∀f:A->?B.
       succeeds(slet(B,f,a)) => slet(B,f,a) = f(ds(a)) in ?B

* THM slet_success
  >> ∀A,B:Type. ∀a:?A. ∀f:A->?B.
       succeeds(slet(B,f,a)) <=> succeeds(a) & succeeds(f(ds(a)))

* THM df_
  >> t:Term0 -> ?( f:Atom # {l:Term0 list| t=f(l) in Term0} )
  Extraction:
  λt. case t to ?(f:Atom#{l:Term0 list|t=f(l) in Term0})
    f,l -> s(<f,l>)
    x,y,A -> fail
    i,x,y -> fail
    n -> fail

* DEF df
  df(<t:Term0>) == df(<t>)

* THM de_
  >> t:Term0 -> ?( u:Term0 # v:Term0 # {A:Atom|t=(u=v in A) in Term0} )
  Extraction:
  λt. case t to ?( u:Term0 # v:Term0 # {A:Atom|t=(u=v in A) in Atom} )
    f,l -> fail
    x,y,A -> s(<x,y,A>)
    i,x,y -> fail
    n -> fail

* DEF de
  de(<t:Term0>) == de(<t>)

* THM di_
  >> t:Term0 -> ?( i:Atom # u:Term0 # {v:Term0|t=(v{i u}) in Term0} )
  Extraction:
  λt. case t to ?( i:Atom # u:Term0 # {v:Term0|t=(v{i u}) in Term0} )
    f,l -> fail
    x,y,A -> fail
    i,x,y -> s(<i,x,y>)
```

```
  n -> fail

* DEF di
  di(<t:Term0>) == di(<t>)

* THM dn_
  >> t:Term0 -> ?( {n:Int|t=(n) in Term0} )
  Extraction:
  λt. case t to ?( i:Atom # u:Term0 # {v:Term0|t=(v{i u}) in Term0} )
    f,l -> fail
    x,y,A -> fail
    i,x,y -> fail
    n -> s(n)

* DEF dn
  dn(<t:Term0>) == dn(<t>)

* THM f_sub_map_
  >> (Term0->?Term0) -> Term0 -> ?Term0
  Extraction:
  λg. λt. case t to ?Term0
    f,args -> let s(args2) = (map g on args to ?(Term0 list)) in
              s(f(args2)): ?Term0
    x,y,A  -> let s(x2), s(y2) = g(x), g(y) in s(x2=y2 in A): ?Term0
    i,x,y  -> let s(x2), s(y2) = g(x), g(y) in s(y2{i x2}): ?Term0
    n      -> s(n)

* DEF f_sub_map
  f_sub_map(<f:Term0-\>?Term0>,<t:Term0>) == f_sub_map(<f>)(<t>)
```

C.1.6 Type Environments

The Type of Type Environments

```
* THM triv_eq_
  >> SET -> U
  Extraction:
  λS. ∀x,y:|S|. ↓(x=y in S) => x=y in S

* DEF triv_eq
  triv_eq(<S:Set>) == triv_eq(<S>)

* THM Int_eq_triv
```

```
  >> triv_eq(↑Int)

* THM False_eq_triv
  >> triv_eq(↑False)

* THM True_eq_triv
  >> triv_eq(↑True)

* THM TEnvVal_
  >> U
  Extraction:
  S:Set # ?triv_eq(S)

* DEF TEnvVal
  TEnvVal == TEnvVal

* THM TEnvUnit_
  >> U
  Extraction:
  Atom # TEnvVal

* DEF TEnvUnit
  TEnvUnit == TEnvUnit

* THM TEnv_
  >> U
  Extraction:
  TEnvUnit list

* DEF TEnv
  TEnv == TEnv

* THM TEnv_containment_lemma
  >> ∀γ:TEnv. γ in Atom#TEnvVal list

* ML add_TEnv_containment_lemma
  add_to_inclusion `TEnv_containment_lemma`
  (\i.
   IfOnConcl ($= 'Atom#TEnvVal list' o eq_type)
    (Lemma `TEnv_containment_lemma` THEN Trivial)
    Fail
  )
  ;;

* THM g0_
  >> TEnv
```

```
  Extraction:
  <"Int", <↑Int, s(Int_eq_triv)>>
  . <"False", <↑False, s(False_eq_triv)>>
  . <"True", <↑True, s(True_eq_triv)>>
  . nil

* DEF g0
  γ0 == g0
```

Meta-Types

```
* THM type_atom@_
  >> TEnv -> Atom -> U
  Extraction:
  λ γ A.
   A="Prop" in Atom
   ∨ ∃∈ γ0 : TEnvUnit list. λu. A=u.1 in Atom
   ∨ ∃∈ γ : TEnvUnit list. λu. A=u.1 in Atom

* DEF type_atom@
  type_atom@(<g:TEnv>,<A:Atom>) == type_atom@(<g>,<A>)

* THM type_atom@_decidable
  >> ∀γ:TEnv. ∀a:Atom. type_atom@(γ,a) ∨ ¬(type_atom@(γ,a))

* ML add_type_atom@_decidable
  add_to_Decidable `type_atom@_decidable`
  (Lemma `type_atom@_decidable`)
  ;;

* THM type_atom_
  >> TEnv -> Atom -> U
  Extraction:
  λ γ a. isl(type_atom@_decidable(γ,a))

* DEF type_atom
  type_atom(<g:TEnv>,<A:Atom>) == type_atom(<g>,<A>)

* THM type_atom_char
  >> ∀γ:TEnv. ∀a:Atom. type_atom(γ,a) <=> type_atom@(γ,a)

* THM Prop_type_atom
  >> ∀γ:TEnv. type_atom(γ,"Prop")
```

```
* THM Int_type_atom
  >> ∀γ:TEnv. type_atom(γ,"Int")

* THM all_type_atoms@_
  >> TEnv -> (Atom list # Atom) -> U
  Extraction:
  λ γ t. let a,b = t in
  ∀∈ a : Atom list. λx. type_atom@(γ,x) &
  type_atom@(γ,b)

* DEF all_type_atoms@
  all_type_atoms@(<g:TEnv>,<x:Atom list # Atom>) == all_type_atoms@(<g>,<x>)

* THM all_type_atoms@_decidable
  >> ∀γ:TEnv. ∀t:Atom list # Atom.
       all_type_atoms@(γ,t) ∨ ¬(all_type_atoms@(γ,t))

* THM all_type_atoms_
  >> TEnv -> (Atom list # Atom) -> U
  Extraction:
  λ γ t. isl(all_type_atoms@_decidable(γ,t))

* DEF all_type_atoms
  all_type_atoms(<g:TEnv>,<x:Atom list # Atom>) == all_type_atoms(<g>,<x>)

* THM all_type_atoms_char
  >> ∀γ:TEnv. ∀t:Atom list # Atom.
  all_type_atoms(γ,t) <=> all_type_atoms@(γ,t)

* THM AtomicMType_i_
  >> TEnv -> U
  Extraction:
  λγ. { a: Atom | type_atom(γ,a) }

* DEF AtomicMType_i
  AtomicMType(<g:TEnv>) == AtomicMType_i(<g>)

* THM MType_i_
  >> TEnv -> U
  Extraction:
  λγ. { t: Atom list # Atom | all_type_atoms(γ,t) }

* DEF MType_i
  MType(<g:TEnv>) == MType_i(<g>)
```

```
* THM MType_i_char
  >> ∀γ:TEnv.
       ∀mt:(Atom list)#Atom. ↓(all_type_atoms@(γ,mt)) => mt in MType(γ)
       & ∀mt:(Atom list)#Atom. ↓(all_type_atoms(γ,mt)) => mt in MType(γ)
       & ∀mt:MType(γ). ↓(all_type_atoms(γ,mt)) & ↓(type_atom(γ,mt.2))
                    & ↓(all_type_atoms@(γ,mt)) & ↓(type_atom@(γ,mt.2))
                    & mt.2 in AtomicMType(γ)

* THM Prop_in_AtomicMType_i
  >> ∀γ:TEnv. "Prop" in AtomicMType(γ)

* ML add_Prop_in_AtomicMType_i
  add_to_member_i `add_Prop_in_AtomicMType_i`
  (IfOnConcl ($= '"Prop"' o first_equand)
    (Lemma `add_Prop_in_AtomicMType_i`)
    Fail
  )
  ;;

* THM mtype_dom_i_atoms
  >> ∀γ:TEnv. ∀mt:MType(γ). mt.1 in AtomicMType(γ) list

* THM mtype_range_i
  >> ∀γ:TEnv. ∀mt:MType(γ). mt.2 in AtomicMType(γ)

* ML add_MType_containment_lemmas
  add_to_member_i `mtype_dom_i_atoms`
  (IfOnConcl (is_list_term o eq_type)
    (Lemma `mtype_dom_i_atoms`)
    Fail
  );;
  add_to_member_i `mtype_range_i`
  (IfOnConcl ($= `AtomicMType_i` o ext_name o eq_type)
    (Lemma `mtype_range_i`)
    Fail
  );;
```

Type-Environment Containment

```
* THM TEnvAp_i_
  >> U
  Extraction:
  S:SET # ?triv_eq(S)
```

```
* DEF TEnvAp_i
  TEnvAp == TEnvAp_i

* DEF TEnvAp
  TEnvAp == TEnvAp_i

* THM TEnvVal_contained_in_TEnvAp_i
  >> ∀v:TEnvVal. v in TEnvAp

* ML add_TEnvVal_contained_in_TEnvAp_i
  add_to_inclusion `TEnvVal_contained_in_TEnvAp_i`
  (\i. Lemma `TEnvVal_contained_in_TEnvAp_i` THEN Trivial) ;;

* THM TEnvAp_i_eq_if_TEnvVal_eq
  >> ∀u,v:TEnvVal. u=v in TEnvVal => u=v in TEnvAp

* THM tenv_ap_i_
  >> γ:TEnv -> AtomicMType(γ) -> TEnvAp
  Extraction:
  λ γ a. if a="Prop" then <Prop,fail>
   else outl( γ0{TEnvVal}(a) ? γ{TEnvVal}(a) )

* DEF tenv_ap_i
  <g:TEnv>(<a:AtomicMType>) == tenv_ap_i(<g>)(<a>)

* THM sub_tenv_
  >> TEnv -> TEnv -> U
  Extraction:
  λ γ1 γ2. (γ1⊂γ2 ∈ (Atom#TEnvVal list))

* DEF sub_tenv
  <g1:TEnv>⊂<g2:TEnv> == sub_tenv(<g1>)(<g2>)

* THM trivial_sub_tenv
  >> ∀γ:TEnv. nil ⊂ γ

* THM tenv_ap_i_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀a:AtomicMType(γ1). γ1(a)=γ2(a) in TEnvAp

* THM type_atom@_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀a:Atom.
    type_atom@(γ1,a) => type_atom@(γ2,a)

* THM type_atom_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀a:Atom.
```

```
  type_atom(γ1,a) => type_atom(γ2,a)

* THM AtomicMType_i_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀a:AtomicMType(γ1). a in AtomicMType(γ2)

* ML add_AtomicMType_i_mono_to_Inclusion
  add_to_inclusion `AtomicMType_i_mono`
  (\i.
   IfOnConcl
    ($= `AtomicMType_i` o ext_name o eq_type)
    (ContainmentLemma `AtomicMType_i_mono` i THEN Trivial)
    Fail
  )
  ;;

* THM all_type_atoms@_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀x:Atom list # Atom.
    all_type_atoms@(γ1,x) => all_type_atoms@(γ2,x)

* THM all_type_atoms_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀x:Atom list # Atom.
    all_type_atoms(γ1,x) => all_type_atoms(γ2,x)

* THM MType_i_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀x:MType(γ1). x in MType(γ2)

* ML add_MType_i_mono_to_Inclusion
  add_to_inclusion `MType_i_mono`
  (\i.
   IfOnConcl
    ($= `MType_i` o ext_name o eq_type)
    (ContainmentLemma `MType_i_mono` i THEN Trivial)
    Fail
  )
  ;;
```

Meta-Type Values

```
* THM type_atom_val_i_
  >> γ:TEnv -> AtomicMType(γ) -> SET
  Extraction:
  λ γ a. γ(a).1
```

```
* DEF type_atom_val_i
  val(<g:TEnv>,<a:AtomicMType>) == type_atom_val_i(<g>,<a>)

* THM type_atom_val_i_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀a:AtomicMType(γ1). val(γ1,a)=val(γ2,a) in SET

* THM type_atom_val_i_on_Prop
  >> ∀γ:TEnv. ∀x:Atom. x="Prop" in Atom => val(γ,x)=Prop in SET

* THM mtype_dom_val_i_
  >> γ:TEnv -> { mt:MType(γ) | ¬(null(mt.1)) } -> SET
  Extraction:
  λ γ mt.
   let l,b = mt in
   #((map λa. val(γ,a) on l to SET list))

* DEF mtype_dom_val_i
  dom_val(<g:TEnv>,<mt:MType>) == mtype_dom_val_i(<g>,<mt>)

* THM mtype_dom_val_i_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 =>
       ∀mt:{ mt:MType(γ1) | ¬(null(mt.1)) }.
         dom_val(γ1,mt)=dom_val(γ2,mt) in SET

* THM mtype_val_i_
  >> γ:TEnv -> MType(γ) -> SET
  Extraction:
  λ γ mt.
   let l,b = mt in
   if null(l) then val(γ,b) else dom_val(γ,mt) -> val(γ,b)

* DEF mtype_val_i
  val(<g:TEnv>,<mt:MType>) == mtype_val_i(<g>,<mt>)

* THM mtype_val_i_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 =>
       ∀mt:MType(γ1). val(γ1,mt)=val(γ2,mt) in SET

* THM mtype_val_i_type_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀mt:MType(γ1).
       ∀f:|val(γ1,mt)|. f in |val(γ2,mt)|

* ML add_mtype_val_i_type_mono
  add_to_inclusion `add_mtype_val_i_type_mono`
  (\i.
    IfOnConcl ($= `tos` o ext_name o eq_type)
```

```
    (BringHyps [i] THEN Lemma `mtype_val_i_type_mono`)
    Fail
)
;;
```

```
* THM mtype_val_i_char
  >> ∀γ:TEnv. ∀mt:MType(γ).
       null(mt.1) => val(γ,mt)=val(γ,mt.2) in SET
       & ¬(null(mt.1)) => val(γ,mt)=dom_val(γ,mt) -> val(γ,mt.2) in SET

* THM type_atom_val_smallish
  >> ∀γ:TEnv. ∀a:AtomicMType(γ).
       val(γ,a) in A:U7 # { r: (A#A)->U6 | eq_reln{A:U7}(r) }

* THM type_atom_val_eq_small
  >> ∀γ:TEnv. ∀a:AtomicMType(γ). ∀x,y:|val(γ,a)|. (x=y in val(γ,a)) in |Prop|

* THM type_atom_val_i_small_type
  >> ∀γ:TEnv. ∀a:AtomicMType(γ). val(γ,a) in SmallEqSET

* THM SmallTEnvAp_i_
  >> U
  Extraction:
  S:SmallEqSET # ?triv_eq(S)

* DEF SmallTEnvAp_i
  SmallTEnvAp == SmallTEnvAp_i

* ML AdjustType
  % When type subterms can be computed out. %
  let AdjustTypeInConcl old new p =
  ( let new_concl = replace_subterm old new (concl p) in
    Assert new_concl
    THENS (OnLastHyp EvalHyp THEN EvalConcl THEN Trivial)
  ) p
  ? failwith `AdjustTypeInConcl`
  ;;

  let AdjustTypeInHyp old new i p =
  ( let new_hyp = replace_subterm old new (h i p) in
    Assert new_hyp
    THENO (EvalHyp i THEN EvalConcl THEN Trivial)
  ) p
  ? failwith `AdjustTypeInHyp`
  ;;
```

```
* THM tenv_alist_ap_small
  >> ∀γ:TEnv. ∀a:Atom. γ{TEnvVal}(a) in ?SmallTEnvAp

* THM tenv_ap_i_small_type
  >> ∀γ:TEnv. ∀a:AtomicMType(γ). γ(a) in SmallTEnvAp

* THM TEnvVal_contained_in_SmallTEnvAp_i
  >> ∀v:TEnvVal. v in SmallTEnvAp

* THM tenv_ap_i_small_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀a:AtomicMType(γ1). γ1(a)=γ2(a) in SmallTEnvAp
```

C.1.7 Function Environments

```
* THM triv_pred_
  >> S:SET -> |S -> Prop| -> U
  Extraction:
  λ S P. ∀x:|S|. ↓(P(x)) => P(x)

* DEF triv_pred
  triv(<S:S:SET>,<P:S-\>Prop>) == triv_pred(<S>,<P>)

* THM semi_triv_pred_
  >> S:SET -> |S -> Prop| -> U
  Extraction:
  λ S P. ∀x,y:|S| where x=y in S. P(x) => P(y)

* DEF semi_triv_pred
  semi_triv(<S:S:SET>,<P:S-\>Prop>) == semi_triv_pred(<S>,<P>)

* THM val_triv_pred_i_
  >> γ:TEnv -> mt:MType(γ) -> |val(γ,mt)| -> U
  Extraction:
  λ γ mt f. ↓(mt.2="Prop" in Atom & ¬(null(mt.1))) & triv(dom_val(γ,mt),f)

* DEF val_triv_pred_i
  triv_pred(<g:TEnv>,<mt:MType>,<v:val>) == val_triv_pred_i(<g>)(<mt>)(<v>)

* THM val_triv_pred_i_mono
  >> ∀γ1,γ2:TEnv. ∀mt:MType(γ1). ∀f:|val(γ1,mt)|.
    γ1⊂γ2 => triv_pred(γ1,mt,f) => triv_pred(γ2,mt,f)
```

```
* THM val_triv_pred_i_mono_2
  >> ∀γ1,γ2:TEnv. ∀mt:MType(γ1). ∀f:|val(γ1,mt)|.
    γ1⊂γ2 => ∀x:triv_pred(γ1,mt,f). x in triv_pred(γ2,mt,f)

* ML add_val_triv_pred_i_mono_2
  add_membership_mono_lemma_to_inclusion
  'val_triv_pred_i_mono_2' 'val_triv_pred_i'
  ;;

* THM val_semi_triv_pred_i_
  >> γ:TEnv -> mt:MType(γ) -> |val(γ,mt)| -> U
  Extraction:
  λ γ mt f. ↓(mt.2="Prop" in Atom & ¬(null(mt.1)))
            & semi_triv(dom_val(γ,mt),f)

* DEF val_semi_triv_pred_i
  semi_triv_pred(<g:TEnv>,<mt:MType>,<v:val>) ==
  val_semi_triv_pred_i(<g>)(<mt>)(<v>)

* THM val_semi_triv_pred_i_mono
  >> ∀γ1,γ2:TEnv. ∀mt:MType(γ1). ∀f:|val(γ1,mt)|.
       γ1⊂γ2 => semi_triv_pred(γ1,mt,f) => semi_triv_pred(γ2,mt,f)

* THM val_semi_triv_pred_i_mono_2
  >> ∀γ1,γ2:TEnv. ∀mt:MType(γ1). ∀f:|val(γ1,mt)|.
       γ1⊂γ2 => ∀x:semi_triv_pred(γ1,mt,f). x in semi_triv_pred(γ2,mt,f)

* ML add_val_semi_triv_pred_i_mono_2
  add_membership_mono_lemma_to_inclusion
  'val_semi_triv_pred_i_mono_2' 'val_semi_triv_pred_i'
  ;;

* THM val_full_injection_i_
  >> γ:TEnv -> mt:MType(γ) -> |val(γ,mt)| -> U
  Extraction:
  λ γ mt f. ¬(null(mt.1)) & full_injection(f∈dom_val(γ,mt)->val(γ,mt.2))

* DEF val_full_injection_i
  full_injection(<g:TEnv>,<mt:MType>,<v:val>) ==
  val_full_injection_i(<g>)(<mt>)(<v>)

* THM val_full_injection_i_mono
  >> ∀γ1,γ2:TEnv. ∀mt:MType(γ1). ∀f:|val(γ1,mt)|.
       γ1⊂γ2 => full_injection(γ1,mt,f) => full_injection(γ2,mt,f)
```

```
* THM val_full_injection_i_mono_2
  >> ∀γ1,γ2:TEnv. ∀mt:MType(γ1). ∀f:|val(γ1,mt)|.
       γ1⊂γ2 => ∀x:full_injection(γ1,mt,f). x in full_injection(γ2,mt,f)

* ML add_val_full_injection_i_mono_2
  add_membership_mono_lemma_to_inclusion
  `val_full_injection_i_mono_2` `val_full_injection_i`
  ;;

* THM val_kind_i_
  >> γ:TEnv -> mt:MType(γ) -> |val(γ,mt)| -> U
  Extraction:
  λ γ mt f. True ∨ triv_pred(γ,mt,f) ∨ semi_triv_pred(γ,mt,f)
             ∨ full_injection(γ,mt,f)

* DEF val_kind_i
  val_kind(<g:TEnv>,<mt:MType>,<v:val>) == val_kind_i(<g>)(<mt>)(<v>)

* THM val_kind_i_mono
  >> ∀γ1,γ2:TEnv. ∀mt:MType(γ1). ∀f:|val(γ1,mt)|.
       γ1⊂γ2 => val_kind(γ1,mt,f) => val_kind(γ2,mt,f)

* THM val_kind_i_mono_2
  >> ∀γ1,γ2:TEnv. ∀mt:MType(γ1). ∀f:|val(γ1,mt)|.
       γ1⊂γ2 => ∀x:val_kind(γ1,mt,f). x in val_kind(γ2,mt,f)

* ML add_val_kind_i_mono_2
  add_membership_mono_lemma_to_inclusion
  `val_kind_i_mono_2` `val_kind_i`
  ;;

* DEF no_kind
  no_kind == inl(axiom)

* DEF triv_pred_kind
  triv_pred_kind(<x:triv_pred>) == inl(inr(<x>))

* DEF semi_triv_pred_kind
  semi_triv_pred_kind(<x:semi_triv_pred>) == inl(inr(inr(<x>)))

* DEF full_injection_kind
  full_injection_kind(<x:full_injection>) == inr(inr(inr(<x>)))

* THM FEnvVal_i_
  >> TEnv -> U
  Extraction:
```

```
  λγ. mt:MType(γ) # f:|val(γ,mt)| # val_kind(γ,mt,f)

* DEF FEnvVal_i
  FEnvVal(<g:TEnv>) == FEnvVal_i(<g>)

* THM FEnvUnit_
  >> TEnv -> U
  Extraction:
  λγ. Atom # FEnvVal(γ)

* DEF FEnvUnit
  FEnvUnit(<g:TEnv>) == FEnvUnit(<g>)

* THM FEnv_
  >> TEnv -> U
  Extraction:
  λγ. FEnvUnit(γ) list

* DEF FEnv
  FEnv(<g:TEnv>) == FEnv(<g>)

* THM FEnvVal_i_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀δ:FEnvVal(γ1). δ in FEnvVal(γ2)

* THM FEnvVal_i_mono_2
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀δ:?FEnvVal(γ1). δ in ?FEnvVal(γ2)

* ML add_FEnvVal_i_mono
  add_to_inclusion ‘FEnvVal_i_mono_2‘
  (\i.
   IfOnConcl (is_union_term o eq_type)
    (BringTyping i THEN Lemma ‘FEnvVal_i_mono_2‘)
    Fail
  )
  ;;

  add_membership_mono_lemma_to_inclusion
  ‘FEnvVal_i_mono‘ ‘FEnvVal_i‘
  ;;

* THM FEnv_mono
  >> ∀γ1,γ2:TEnv. γ1⊂γ2 => ∀δ:FEnv(γ1). δ in FEnv(γ2)

* ML add_FEnv_mono
  add_membership_mono_lemma_to_inclusion
  ‘FEnv_mono‘ ‘FEnv‘
```

```
  ;;

* THM FEnv_containment_lemma_2
  >> ∀γ:TEnv. ∀δ:FEnv(γ). δ in (Atom # FEnvVal(γ)) list

* ML add_FEnv_containment_lemma
  add_to_inclusion `FEnv_containment_lemma_2`
  (\i.
   IfOnConcl (is_list_term o eq_type)
    (Lemma `FEnv_containment_lemma_2`)
    Fail) ;;

  add_to_inclusion `FEnv_containment_lemma`
  (\i.
   IfOnConcl (is_list_term o eq_type)
    (Lemma `FEnv_containment_lemma`)
    Fail) ;;

* THM a0_
  >> FEnv(γ0)
  Extraction:
  <"True",<nil,"Prop">,True,no_kind>
  . <"False",<nil,"Prop">,False,no_kind>
  . nil

* DEF a0
  δ0 == d0

* THM d0_
  >> FEnv(nil)
  Extraction:
  <"True",<nil,"Prop">,True,no_kind>
  . <"False",<nil,"Prop">,False,no_kind>
  . nil

* DEF d0
  δ0 == d0
```

C.1.8 Environments

The Type of Environments

```
* THM Env_
```

```
  >> U
  Extraction:
  γ:TEnv # FEnv(γ)

* DEF Env
  Env == Env

* THM FEnv_containment_lemma
  >> ∀α:Env. ∀δ:FEnv(α.1). δ in (Atom # FEnvVal(α.1)) list

* THM FEnvVal_
  >> Env -> U
  Extraction:
  λα. FEnvVal(α.1)

* DEF FEnvVal
  FEnvVal(<g:Env>) == FEnvVal(<g>)

* THM MType_
  >> Env -> U
  Extraction:
  λα. MType(α.1)

* DEF MType
  MType(<a:Env>) == MType(<a>)

* THM AtomicMType_
  >> Env -> U
  Extraction:
  λα. AtomicMType(α.1)

* DEF AtomicMType
  AtomicMType(<a:Env>) == AtomicMType(<a>)

* THM AtomicMType_eq_char
  >> ∀α:Env. ∀A,B:AtomicMType(α). A=B in AtomicMType(α) <=> A=B in Atom

* THM mtype_dom_atoms
  >> ∀α:Env. ∀mt:MType(α). mt.1 in AtomicMType(α) list

* THM mtype_range
  >> ∀α:Env. ∀mt:MType(α). mt.2 in AtomicMType(α)

* ML add_more_MType_containment_lemmas
  add_to_member_i `mtype_dom_atoms`
  (IfOnConcl (is_list_term o eq_type)
```

```
  (IfOnConcl (\c. can (match 'mtype(α,f).1' (first_equand c)) ``α f``)
     (Lemma `mtype_dom_atoms`)
     (Lemma `mtype_dom_atoms` THEN Complete (Id...)))
  Fail
)
;;

add_to_member_i `mtype_range_i`
(IfOnConcl ($= `AtomicMType` o ext_name o eq_type)
  (IfOnConcl (\c. can (match 'mtype(α,f).2' (first_equand c)) ``α f``)
     (Lemma `mtype_range`)
     (Lemma `mtype_range` THEN Complete (Id...)))
  Fail
)
;;

* THM tenv_ap_
>> α:Env -> AtomicMType(α) -> TEnvAp
Extraction:
λ α a. (α.1)(a)

* DEF tenv_ap
<g:Env>(<a:AtomicMType>) == tenv_ap(<g>)(<a>)

* THM type_atom_val_
>> α:Env -> AtomicMType(α) -> SET
Extraction:
λ α a. val(α.1,a)

* DEF type_atom_val
val(<g:Env>,<a:AtomicMType>) == type_atom_val(<g>,<a>)

* THM type_atom_val_small_type
>> ∀α:Env. ∀a:AtomicMType(α). val(α,a) in SmallEqSET

* THM mtype_dom_val_
>> α:Env -> { mt:MType(α) | ¬(null(mt.1)) } -> SET
Extraction:
λ α mt. dom_val(α.1,mt)

* DEF mtype_dom_val
dom_val(<g:Env>,<mt:MType>) == mtype_dom_val(<g>,<mt>)

* THM mtype_val_
>> α:Env -> MType(α) -> SET
Extraction:
```

```
  λ α mt. val(α.1,mt)

* DEF mtype_val
  val(<g:Env>,<mt:MType>) == mtype_val(<g>,<mt>)

* THM mtype_val_char
  >> ∀α:Env. ∀mt:MType(α).
       null(mt.1) => val(α,mt)=val(α,mt.2) in SET
       & ¬(null(mt.1)) => val(α,mt)=dom_val(α,mt) -> val(α,mt.2) in SET
```

The Type of Meta-Functions

```
* THM fun_atom@_
  >> Env -> Atom -> U
  Extraction:
  λ α a.  ∃∈ δ0 : FEnvUnit(nil) list. λu. a=u.1 in Atom
   ∨ ∃∈ α.2 : FEnvUnit(α.1) list. λu. a=u.1 in Atom

* DEF fun_atom@
  fun_atom@(<g:Env>,<A:Atom>) == fun_atom@(<g>,<A>)

* THM fun_atom@_decidable
  >> ∀α:Env. ∀a:Atom. fun_atom@(α,a) ∨ ¬(fun_atom@(α,a))

* THM fun_atom_
  >> Env -> Atom -> U
  Extraction:
  λ α a. isl(fun_atom@_decidable(α,a))

* DEF fun_atom
  fun_atom(<g:Env>,<A:Atom>) == fun_atom(<g>,<A>)

* THM fun_atom_char
  >> ∀α:Env. ∀a:Atom. fun_atom(α,a) <=> fun_atom@(α,a)

* THM MFun_
  >> Env -> U
  Extraction:
  λα. {a:Atom|fun_atom(α,a)}

* DEF MFun
  MFun(<a:Env>) == MFun(<a>)
```

```
* THM fenv_ap_
  >> α:Env -> f:MFun(α) -> FEnvVal(α)
  Extraction:
  λ α f. outl( δ0{FEnvVal(nil)}(f) ? (α.2){FEnvVal(α.1)}(f) )

* DEF fenv_ap
  <g:Env>(<a:AtomicMFun>) == fenv_ap(<g>)(<a>)

* THM mtype_
  >> α:Env -> MFun(α) -> MType(α)
  Extraction:
  λ α f. (α(f)).1

* DEF mtype
  mtype(<a:Env>,<f:MFun>) == mtype(<a>)(<f>)

* THM mfun_val_
  >> α:Env -> f:MFun(α) -> |val(α,mtype(α,f))|
  Extraction:
  λ α f. (α(f)).2.1

* DEF mfun_val
  val(<a:Env>,<f:MFun>) == mfun_val(<a>)(<f>)

* THM constant_mfun_val
  >> ∀α:Env. ∀f:MFun(α). null(mtype(α,f).1) =>
       val(α,f) in |val(α,mtype(α,f).2)|

* ML add_constant_mfun_val
  add_to_member_i `constant_mfun_val`
  (IfOnConcl
    (\c. `tos` = ext_name (eq_type c)
         & let l = decompose_ap (first_equand c) in
           length l = 3 & `mfun_val` = ext_name (hd l)
    )
    (Lemma `constant_mfun_val`)
    Fail
  )
  ;;

* THM mfun_val_fun
  >> ∀α:Env. ∀f:MFun(α). ¬(null(mtype(α,f).1)) =>
       val(α,f) in |dom_val(α,mtype(α,f))| -> |val(α,mtype(α,f).2)|

* THM mfun_val_ap
  >> ∀α:Env. ∀f:MFun(α). ¬(null(mtype(α,f).1)) =>
```

```
     ∀x:|dom_val(α,mtype(α,f))|.  val(α,f)(x) in |val(α,mtype(α,f).2)|

* ML add_mfun_val_ap
  add_to_member_i `mfun_val_ap`
  (IfOnConcl
    (\c. `tos` = ext_name (eq_type c)
         & let l = decompose_ap (first_equand c) in
           length l = 4 & `mfun_val` = ext_name (hd l)
    )
    (Lemma `mfun_val_ap`)
    Fail
  )
  ;;

* THM val_kind_
  >> α:Env -> MFun(α) -> U
  Extraction:
  λ α f. val_kind(α.1, mtype(α,f), val(α,f))

* DEF val_kind
  val_kind(<g:Env>,<f:MFun>) == val_kind(<g>)(<f>)

* THM val_kind_pf_
  >> α:Env -> f:MFun(α) -> val_kind(α,f)
  Extraction:
  λ α f. (α(f)).2.2

* DEF val_kind_pf
  val_kind_pf(<g:Env>,<mt:MFun>) == val_kind_pf(<g>)(<mt>)

* THM is_injection_
  >> α:Env -> MFun(α) -> U
  Extraction:
  λ α f. d(val_kind_pf(α,f);u.False;v.d(v;u.False;v.isr(v)))

* DEF is_injection
  is_injection(<a:Env>,<mt:MFun>) == is_injection(<a>)(<mt>)

* THM is_injection_char
  >> ∀α:Env. ∀f:MFun(α). is_injection(α,f) =>
       ¬( null ( mtype(α,f).1 ) ) &
       ↓(injective( val(α,f) ∈ dom_val(α,mtype(α,f)) -> val(α,mtype(α,f).2)
))

* THM is_triv_pred_
  >> α:Env -> MFun(α) -> U
```

```
Extraction:
λ α f. d(val_kind_pf(α,f);u.False;v.isl(v))

* DEF is_triv_pred
is_triv_pred(<a:Env>,<mt:MFun>) == is_triv_pred(<a>)(<mt>)

* THM is_triv_pred_char
>> ∀α:Env. ∀f:MFun(α). is_triv_pred(α,f) =>
     ↓(mtype(α,f).2 = "Prop" in Atom &  ¬(null(mtype(α,f).1)))
     & triv(dom_val(α,mtype(α,f)), val(α,f))

* THM is_semi_triv_pred_
>> α:Env -> MFun(α) -> U
Extraction:
λ α f. d(val_kind_pf(α,f);u.False;v.d(v;u.False;v.isl(v)))

* DEF is_semi_triv_pred
is_semi_triv_pred(<a:Env>,<mt:MFun>) == is_semi_triv_pred(<a>)(<mt>)

* THM is_semi_triv_pred_char
>> ∀α:Env. ∀f:MFun(α). is_semi_triv_pred(α,f) =>
     ↓(mtype(α,f).2 = "Prop" in Atom &  ¬(null(mtype(α,f).1)))
     & semi_triv(dom_val(α,mtype(α,f)),val(α,f))

* THM Int_in_AtomicMType
>> ∀α:Env. "Int" in AtomicMType(α)

* ML add_Int_in_AtomicMType
add_to_member_i `Int_in_AtomicMType`
(IfOnConcl ($= '"Int"' o first_equand)
  (Lemma `Int_in_AtomicMType`)
  Fail
)
;;
```

C.1.9 Booleans

```
* THM Bool_
>> U1
Extraction:
True ∨ True

* DEF Bool
```

```
  Bool == Bool

* THM btrue_
  >> Bool
  Extraction:
  inl(ax)

* DEF btrue
  true == btrue

* THM is_btrue_
  >> Bool -> U1
  Extraction:
  λb. isl(b)

* DEF is_btrue
  <b:Bool> == is_btrue(<b>)

* THM bfalse_
  >> Bool
  Extraction:
  inr(ax)

* DEF bfalse
  false == bfalse

* THM is_bfalse_
  >> Bool -> U1
  Extraction:
  λb. isr(b)

* DEF is_bfalse
  ¬(<b:Bool>) == is_bfalse(<b>)

* THM bif_
  >> Object
  Extraction:
  λ b f g. d(b;u.f(u);v.g(v))

* DEF bif
  if <b:Bool> then <f:True-\>*> else <g:True-\>*> == bif(<b>)(<f>)(<g>)

* THM bif__
  >> ∀A:Type. ∀b:Bool. ∀f:↓(b)->A. ∀g:↓(¬(b))->A.
       (if b then f else g) in A
```

The "Partial" Booleans

```
* THM PBool_
  >> U'
  Extraction:
  Bool ∨ U

* DEF PBool
  PBool == PBool

* ML PBoolE
  set_d_tactic_args 1 [] [] [] ;;

  let PBoolE i =
   Pattern `PBoolE_pattern` [] [] [] i ;;

# THM PBoolE_pattern
  >> x:PBool -> "P"(x)

* THM prop_
  >> Object
  Extraction:
  λP. inr(P)

* DEF prop
  prop(<P:U>) == prop(<P>)

* THM prop__
  >> ∀Q:U. prop(Q) in PBool

* THM bool_
  >> Object
  Extraction:
  λb. inl(b)

* DEF bool
  bool(<P:Bool>) == bool(<P>)

* THM bool__
  >> ∀b:Bool. bool(b) in PBool

* THM PBool_cases_
```

```
  >> Object
  Extraction:
  λ tt ff g. λt. d(t;u.d(u;v.tt;v.ff);u.g(u))

* DEF PBool_cases
  case <pb:PBool(P)>   true: <tt:A> false: <ff:A> else: <g:U-\>A> ==
  PBool_cases(<tt>)(<ff>)(<g>)(<pb>)

* THM PBool_cases_
  >> ∀A:Type. ∀tt,ff:A. ∀g:U->A. ∀pb:PBool.
       case pb   true: tt false: ff else: g in A

* THM is_pbtrue_
  >> PBool -> U
  Extraction:
  λpb. case pb   true: True false: False else: λQ. Q

* DEF is_pbtrue
  <pb:PBool> == is_pbtrue(<pb>)

* THM prop_char
  >> ∀Q:U. prop(Q) <=> Q

* THM pband_
  >> PBool -> PBool -> PBool
  Extraction:
  λ pb1 pb2.
  case pb1
     true: pb2
     false: bool(false)
     else: λQ1. case pb2   true: prop(Q1) false: bool(false)
                           else: λQ2. prop(Q1 & Q2)

* DEF pband
  <P:PBool> & <Q:PBool> == pband(<P>)(<Q>)

* THM pband_char
  >> ∀pb1,pb2:PBool. pb1 & pb2 <=> pb1 & pb2

* THM pbcand_
  >> pb:PBool -> (↓(pb)->PBool) -> PBool
  Extraction:
  λ pb1 pb2.
  case pb1
     true: pb2(ax)
     false: bool(false)
```

```
    else: λQ1. prop(Q1 & pb2(ax))

* DEF pbcand
  <P:PBool> & <Q:↓(.)-\>PBool> == pbcand(<P>)(<Q>)

* THM pbcand_char
  >> ∀pb1:PBool. ∀pb2:↓(pb1)->PBool.
       pb1 & pb2 <=> pb1 & pb2(ax)

* THM pbnot_
  >> PBool -> PBool
  Extraction:
  λpb.
  case pb   true: bool(false) false: bool(true) else: λQ. prop(¬(Q))

* DEF pbnot
  ¬(<pb:PBool>) == pbnot(<pb>)

* THM pbnot_char
  >> ∀pb:PBool. ¬(pb) <=> ¬(pb)

* THM pbor_
  >> PBool -> PBool -> PBool
  Extraction:
  λ pb1 pb2.
  case pb1
    true: bool(true)
    false: pb2
    else: λQ1. case pb2   true: bool(true) false: prop(Q1)
                          else: λQ2. prop(Q1 ∨ Q2)

* DEF pbor
  <P:PBool> ∨ <Q:PBool> == pbor(<P>)(<Q>)

* THM pbor_char
  >> ∀pb1,pb2:PBool.
       pb1 ∨ pb2 <=> pb1 ∨ pb2

* THM pb_all_elements_
  >> Object
  Extraction:
  λ A Q l. [ nil → bool(true); h.t,v → Q(h) & v; @ l]

* DEF pb_all_elements
  ∀∈ <l:list> : <A:type> list. <Q:pb-pred> ==
  pb_all_elements(<A>)(<Q>)(<l>)
```

```
* THM pb_all_elements__
  >> ∀A:Type. ∀P:A->PBool. ∀l:A list. (∀∈ l : A list. P) in PBool

* THM pb_all_elements_char
  >> ∀A:U. ∀P:A->PBool. ∀l:A list.
       ∀∈ l : A list. P
       <=> ∀∈ l : A list. λa. P(a)

* THM pb_Int_eq_
  >> Int -> Int -> PBool
  Extraction:
  λ m n. if m=n then bool(true) else bool(false)

* DEF pb_Int_eq
  <x:int>=<y:int> == pb_Int_eq(<x>)(<y>)

* THM pb_Int_eq_char
  >> ∀m,n:Int. m=n <=> m = n

* THM pb_Atom_eq_
  >> Atom -> Atom -> PBool
  Extraction:
  λ a b. if a=b then bool(true) else bool(false)

* DEF pb_Atom_eq
  <x:Atom>=<y:Atom> == pb_Atom_eq(<x>)(<y>)

* THM pb_Atom_eq_char
  >> ∀a,b:Atom. a=b <=> (a=b in Atom)

* THM pb_of_decide_
  >> Object
  Extraction:
  λd. d(d;u.bool(true);u.bool(false))

* DEF pb_of_decide
  pb_of_decide(<d:P∨¬P>) == pb_of_decide(<d>)

* THM pb_of_decide__
  >> ∀P:Type. ∀d:(P ∨ ¬(P)). pb_of_decide(d) in PBool

* THM pb_of_decide_char
  >> ∀P:Type. ∀d:(P ∨ ¬(P)). pb_of_decide(d) <=> P

* ML PBoolChar
```

```
let PBoolChar_lemmas =
``pband_char pbcand_char pbor_char pb_all_elements_char
  pb_Int_eq_char pb_Atom_eq_char pb_of_decide_char  pbnot_char`` ;;

let PBoolChar = First (map Lemma PBoolChar_lemmas) ;;

let FPBoolChar i = First (map (\x. FLemma x [i]) PBoolChar_lemmas) ;;
```

C.1.10 Well-Formedness and Evaluation

"Values"

```
* THM Prop_in_AtomicMType
  >> ∀α:Env. "Prop" in AtomicMType(α)

* ML add_Prop_in_AtomicMType
  add_to_member_i `Prop_in_AtomicMType`
  (IfOnConcl ($= '"Prop"' o first_equand)
    (Lemma `Prop_in_AtomicMType`)
    Fail
  )
  ;;

* THM Val_
  >> Env -> U
  Extraction:
  λα. a:AtomicMType(α) # |val(α,a)|

* DEF Val
  Val(<a:Env>) == Val(<a>)

* THM val_member_
  >> α:Env -> v:Val(α) -> B:AtomicMType(α) -> U
  Extraction:
  λ α v B.
  let A,a = v in
  A=B in Atom ∨ val(α,A)⊂⊂val(α,B) ∨ ∃Q:val(α,B)⊂⊂val(α,A) where Q(a)

* DEF val_member
  <v:Val> ∈{<a:Env>} <B:AtomicMType> == val_member(<a>)(<v>)(<B>)

* THM val_member_char
  >> ∀α:Env. ∀v:Val(α). ∀B:AtomicMType(α). v ∈{α} B => v.2 in |val(α,B)|
```

```
* THM vals_in_mtypes@_
  >> α:Env -> Val(α) list -> AtomicMType(α) list -> U
  Extraction:
  λ α vl aml.
  |aml|=|vl| in N &
  ∀∈ com(vl,aml) : Val(α)#AtomicMType(α) list. (λ v,B. v ∈{α} B)

* DEF vals_in_mtypes@
  <vl:Val list> ∈{<a:Env>} <mt:AtomicMType list> ==
  vals_in_mtypes@(<a>)(<vl>)(<mt>)

* ML NilApMember
  let NilApMember =
    (If ($= 'nil' o last o decompose_ap o first_equand o concl)
        (Progress ComputeEquands)
        Fail
    ...*)
  ;;

  add_to_autotactic `NilApMember` NilApMember ;;

* DEF context_of_main_induction
  ∀α,P,wf,mtype,val. <P:Prop> ==
  ∀α:Env. ∀P:Term0->U. ∀wf: {t:Term0|P(t)} -> U .
  ∀mtype: {t:Term0|P(t) & wf(t)} -> AtomicMType(α) .
  ∀val: t:{t:Term0|P(t) & wf(t)} -> |val(α,mtype(t))| . <P>

* THM arg_tuple_typing
  >> ∀α,P,wf,mtype,val. ∀mt:MType(α). ∀l:({t:Term0|P(t) & wf(t)} list).
       ¬(null(mt.1))
       => (map λt. <mtype(t),val(t)> on l to Val(α) list) ∈{α} mt.1
       => ([a] → val(a); λ h t v. <val(h),v>; @ l) in |dom_val(α,mt)|
```

The Basic Functions

```
* THM term_mtype_
  >> Object
  Extraction:
  λ α t. rec_ind(t; h,t.
  case t to AtomicMType(α)
    f,args -> mtype(α,f).2
    x,y,A -> "Prop"
```

```
  i,x,y -> h(x)
  n -> "Int")

* DEF term_mtype
  mtype(<a:Env>,<t:Term@>) == term_mtype(<a>)(<t>)

* THM term_type_
  >> Object
  Extraction:
  λ α t. val(α,mtype(α,t))

* DEF term_type
  type(<a:Env>,<t:Term@>) == term_type(<a>)(<t>)

* THM term_list_val_
  >> Object
  Extraction:
  λ α g l. ([a] → g(a); λ h t v. <g(h),v>; @ l)

* DEF term_list_val
  <g:val fun>{<a:Env>}(<l:Term@ list>) == term_list_val(<a>)(<g>)(<l>)

* THM term_val_
  >> Object
  Extraction:
  λ α t. rec_ind(t; g,t.
  case t to |type(α,t)|
    f,args -> if null(mtype(α,f).1) then val(α,f)
              else val(α,f) (g{α}(args))
    x,y,A  -> g(x)=g(y) in val(α,A)
    i,x,y -> g(x)
    n -> n)

* DEF term_val
  val(<a:Env>,<t:Term@>) == term_val(<a>)(<t>)

* THM wf_fun_ap@_
  >> Object
  Extraction:
  λ α f args.
  fun_atom@(α,f) &
  (map λt. <mtype(α,t), val(α,t)> on args to Val(α) list)  ∈{α} mtype(α,f).1


* DEF wf_fun_ap@
  wf_fun_ap@(<a:Env>,<f:Atom>,<args:Term list>) == wf_fun_ap@(<a>)(<f>)(<args>)
```

```
* THM wf_eq_ap@_
  >> Object
  Extraction:
  λα. λ x y A.
  type_atom(α.1,A) &
  <mtype(α,x),val(α,x)> ∈{α} A  &  <mtype(α,y),val(α,y)> ∈{α} A

* DEF wf_eq_ap@
  wf_eq_ap@(<a:Env>,<x:Term>,<y:Term>,<A:Atom>) == wf_eq_ap@(<a>)(<x>)(<y>)(<A>)


* THM wf_i_pair@_
  >> Object
  Extraction:
  λα. λ i x y.
  fun_atom@(α,i) & is_injection(α,i) &
  [mtype(α,x)] = mtype(α,i).1 in Atom list &
  mtype(α,y) = mtype(α,i).2 in Atom &
  val(α,i)(val(α,x)) = val(α,y) in val(α,mtype(α,y))

* DEF wf_i_pair@
  wf_i_pair@(<a:Env>,<i:Atom>,<x:Term>,<y:Term>) ==
  wf_i_pair@(<a>)(<i>)(<x>)(<y>)

* THM wf_term@_
  >> Object
  Extraction:
  \ α.\ t.rec_ind(t; F,t.
  case t to U
    f,args -> ∀∈ args : Term0 list. F & wf_fun_ap@(α,f,args)
    x,y,A  -> F(x) & F(y) & wf_eq_ap@(α,x,y,A)
    i,x,y  -> F(x) & F(y) & wf_i_pair@(α,i,x,y)
    n      -> True)

* DEF wf_term@
  wf@(<a:Env>,<t:Term0>) == wf_term@(<a>)(<t>)

* THM wf_fun_ap@_body
  >> ∀α,P,wf,mtype,val. ∀f:Atom. ∀args:{x:Term0|P(x)} list.
        ∀∈ args : {x:Term0|P(x)} list. wf =>
        fun_atom@(α,f) &
          (map λt. <mtype(t), val(t)> on args to Val(α) list)
          ∈{α} mtype(α,f).1
        in U
```

```
* THM wf_i_pair@_body
  >> ∀α,P,wf,mtype,val. ∀i:Atom. ∀x,y:{t:Term0|P(t)}.
        wf(x) =>
        wf(y) =>
        fun_atom@(α,i) & is_injection(α,i)
           & [mtype(x)] = mtype(α,i).1 in Atom list
           & mtype(y) = mtype(α,i).2 in Atom
           & val(α,i)(val(x)) = val(y) in val(α,mtype(y))
        in U

* THM main_typing_lemma
  >> ∀α:Env. ∀t:Term0. wf@(α,t) in U &
       ↓(wf@(α,t)) => mtype(α,t) in AtomicMType(α) & val(α,t) in |type(α,t)|

* THM wf_term@_
  >> ∀α:Env. ∀t:Term0. wf@(α,t) in U

* THM Term_
  >> Env -> U
  Extraction:
  λα. {t:Term0|wf@(α,t)}

* DEF Term
  Term(<a:Env>) == Term(<a>)

* THM term_mtype_
  >> ∀α:Env. ∀t:Term(α). mtype(α,t) in AtomicMType(α)

* THM term_type_
  >> ∀α:Env. ∀t:Term(α). type(α,t) in SET

* THM term_val_
  >> ∀α:Env. ∀t:Term(α). val(α,t) in |type(α,t)|
```

Miscellaneous

```
* THM term_list_type_
  >> Object
  Extraction:
  λ α l. #((map term_type(α) on l to SET list))
```

```
* DEF term_list_type
  type(<a:Env>,<l:Term@ list>) == term_list_type(<a>)(<l>)

* THM term_list_type_
  >> ∀α:Env. ∀l:Term(α) list. ¬(null(l)) => type(α,l) in SET

* THM term_list_val_
  >> ∀α:Env. ∀l:Term(α) list.
       ¬(null(l)) => (term_val(α)){α}(l) in |type(α,l)|

* THM term_list_val_in_mtype_dom
  >> ∀α:Env. ∀f:Atom. ∀args:Term(α) list.
        wf_fun_ap@(α,f,args) => ¬(null(mtype(α,f).1))
        => term_val(α){α}(args) in |dom_val(α, mtype(α,f))|

* THM terms_val_
  >> Object
  Extraction:
  λ α l. term_val(α){α}(l)

* DEF terms_val
  val(<a:Env>,<l:Term list>) == terms_val(<a>)(<l>)

* THM terms_val_
  >> ∀α:Env. ∀l:Term(α) list. ¬(null(l)) => val(α,l) in |type(α,l)|

* THM term_list_type_base
  >> ∀α:Env. ∀h:Term(α). ∀l:Term(α) list.
       null(l) => type(α,h.l) = type(α,h) in SET

* THM term_list_type_base_mem
  >> ∀α:Env. ∀h:Term(α). ∀l:Term(α) list. null(l)
       => ∀x:|type(α,h)|. x in |type(α,h.l)|

* THM term_list_type_unroll
  >> ∀α:Env. ∀h:Term(α). ∀l:Term(α) list. ¬(null(l))
        => type(α,h.l) = type(α,h) # type(α,l) in SET

* THM term_list_type_unroll_mem
  >> ∀α:Env. ∀h:Term(α). ∀l:Term(α) list. ¬(null(l))
        => ∀x:|type(α,h)|#|type(α,l)|. x in |type(α,h.l)|

* THM mfun_if_wf_term@_1
  >> ∀α:Env. ∀f:Atom. ∀l:Term0 list. wf@(α,f(l)) => f in MFun(α)

* THM mfun_if_wf_term@_2
```

```
  >> ∀α:Env. ∀i:Atom. ∀u,v:Term0. wf@(α,v{i u}) => i in MFun(α)

* THM wf_fun_ap@_
  >> ∀α:Env. ∀f:Atom. ∀args:Term0 list.
       ∀∈ args : Term0 list. wf_term@(α) => wf_fun_ap@(α,f,args) in U

* THM wf_i_pair@_aux_
  >> Object
  Extraction:
  λα. λ i x y.
  fun_atom@(α,i) & is_injection(α,i) &
  [mtype(α,x)] = mtype(α,i).1 in Atom list &
  mtype(α,y) = mtype(α,i).2 in Atom

* DEF wf_i_pair@_aux
  wf_i_pair@_aux(<a:Env>,<i:Atom>,<x:Term>,<y:Term>) ==
  wf_i_pair@_aux(<a>)(<i>)(<x>)(<y>)

* THM wf_i_pair@_aux_
  >> ∀α:Env. ∀i:Atom. ∀x,y:Term0.
       wf@(α,x) => wf@(α,y) => wf_i_pair@_aux(α,i,x,y) in U

* THM wf_i_pair@_aux_lemma
  >> ∀α:Env. ∀i:Atom. ∀x,y:Term0.
       wf@(α,x) => wf@(α,y) => wf_i_pair@_aux(α,i,x,y) =>
       val(α,i) in |type(α,x)|->|type(α,y)|

* THM wf_i_pair@_
  >> ∀α:Env. ∀i:Atom. ∀x,y:Term0.
       wf@(α,x) => wf@(α,y) => wf_i_pair@(α,i,x,y) in U

* THM wf_eq_ap@_
  >> ∀α:Env. ∀x,y:Term0. ∀A:Atom.
       wf@(α,x) => wf@(α,y) => wf_eq_ap@(α,x,y,A) in U

* THM mtype_pair
  >> ∀α:Env. ∀mt:MType(α). mt = <mt.1, mt.2> in MType(α)

* THM wf_term@_hereditary
  >> ∀α:Env. (λt. wf@(α,t) hereditary over Term0)

* ML TermUnroll
  let TermUnroll i p =
    if can (match 'Term(a)' (h i p)) ``a``
    then CHThen Term0SubsetUnroll i p
    else failwith `TermUnroll`
```

```
;;

* ML TermInduction
  let TermInduction i p =
    if can (match 'Term(a)' (h i p)) ''a''
    then CHThen Term0SubsetInduction i p
    else failwith `TermInduction`
  ;;

* THM Term_induction
  >> ∀α:Env. ∀Q:Term(α)->U.
          ∀f:Atom. ∀l:Term(α) list.
             ∀∈ l : Term(α) list. Q => ↓(wf@(α,f(l))) => Q(f(l))
       => ∀t,u:Term(α). ∀A:Atom. Q(t) => Q(u) => ↓(wf@(α,t=u in A))
                                              => Q(t=u in A)
       => ∀i:Atom. ∀t,u:Term(α). Q(t) => Q(u) => ↓(wf@(α,u{i t}))
                                              => Q(u{i t})
       => ∀n:Int. Q(n)
       => ∀t:Term(α). Q(t)

* THM Term_containment_lemma
  >> ∀α:Env. ∀l:Term0 list. ∀∈ l : Term0 list.
       wf_term@(α) => l in Term(α) list

* ML add_Term_containment_lemma
  add_to_inclusion `add_Term_containment_lemma`
  (\i. IfOnConcl ($= `Term` o ext_name o destruct_list o eq_type)
         ((Lemma `Term_containment_lemma` ...*))
         Fail
  )
  ;;
```

C.1.11 Typechecking

```
* THM pb_val_member_
  >> α:Env -> v:Val(α) -> B:AtomicMType(α) -> PBool
  Extraction:
  λ α v B.
  let A,a = v in
  A=B ∨ prop(val(α,A)⊂⊂val(α,B) ∨ ∃Q:val(α,B)⊂⊂val(α,A) where Q(a))

* DEF pb_val_member
  <v:Val> ∈{<a:Env>} <B:AtomicMType> == pb_val_member(<a>)(<v>)(<B>)
```

```
* THM pb_val_member_char
  >> ∀α:Env. ∀v:Val(α). ∀B:AtomicMType(α).
       v ∈{α} B <=> v ∈{α} B

* THM vals_in_mtypes_
  >> α:Env -> Val(α) list -> AtomicMType(α) list -> PBool
  Extraction:
  λ α vl aml.
  |aml|=|vl| &
  λax. ∀∈ com(vl,aml) : Val(α)#AtomicMType(α) list. (λ v,B. v ∈{α} B)

* DEF vals_in_mtypes
  <vl:Val list> ∈{<a:Env>} <mt:AtomicMType list> ==
  vals_in_mtypes(<a>)(<vl>)(<mt>)

* THM vals_in_mtypes_char
  >> ∀α:Env. ∀vl:Val(α) list. ∀amt:AtomicMType(α) list.
       vl ∈{α} amt <=> vl ∈{α} amt

* THM wf_fun_ap_aux_
  >> Object
  Extraction:
  λ α f args.
  pb_of_decide(fun_atom@_decidable(α,f)) &
  λax. (map λt. <mtype(α,t), val(α,t)> on args to Val(α) list)
       ∈{α} mtype(α,f).1

* DEF wf_fun_ap_aux
  wf_fun_ap_aux(<a:Env>,<f:Atom>,<args:Term list>) ==
  wf_fun_ap_aux(<a>)(<f>)(<args>)

* THM wf_fun_ap_aux__
  >> ∀α:Env. ∀f:Atom. ∀args:Term0 list.
       ∀∈ args : Term0 list. wf_term@(α) => wf_fun_ap_aux(α,f,args) in PBool


* THM wf_fun_ap_aux_char
  >> ∀α:Env. ∀f:Atom. ∀args:Term0 list.
       ∀∈ args : Term0 list. wf_term@(α) =>
       wf_fun_ap_aux(α,f,args) <=> wf_fun_ap@(α,f,args)

* THM wf_fun_ap_
  >> Object
  Extraction:
  λ α f args.
```

```
  pb_of_decide(fun_atom@_decidable(α,f)) & λax.
  let aml  = mtype(α,f).1 in
  let vl = (map λt. <mtype(α,t), val(α,t)> on args to Val(α) list) in
  |aml|=|args| & λax.
  ∀∈ com(vl,aml) : Val(α)#AtomicMType(α) list.
    λ v,B. let A,a = v in A=B
          ∨ prop( val(α,A)⊂⊂val(α,B) ∨ ∃Q:val(α,B)⊂⊂val(α,A) where Q(a)
)

* DEF wf_fun_ap
  wf_fun_ap(<a:Env>,<f:Atom>,<args:Term list>) == wf_fun_ap(<a>)(<f>)(<args>)


* THM wf_fun_ap__
  >> ∀α:Env. ∀f:Atom. ∀args:Term0 list.
       ∀∈ args : Term0 list. wf_term@(α) => wf_fun_ap(α,f,args) in PBool

* THM wf_eq_ap_
  >> Object
  Extraction:
  λα. λ x y A.
  pb_of_decide(type_atom@_decidable(α.1,A)) &
  λax. <mtype(α,x),val(α,x)> ∈{α} A  &  <mtype(α,y),val(α,y)> ∈{α} A

* DEF wf_eq_ap
  wf_eq_ap(<a:Env>,<x:Term>,<y:Term>,<A:Atom>) == wf_eq_ap(<a>)(<x>)(<y>)(<A>)


* THM wf_eq_ap__
  >> ∀α:Env. ∀x,y:Term0. ∀A:Atom.
       wf@(α,x) => wf@(α,y) => wf_eq_ap(α,x,y,A) in PBool

* THM wf_eq_ap_char
  >> ∀α:Env. ∀x,y:Term0. ∀A:Atom.
       wf@(α,x) => wf@(α,y) =>
       wf_eq_ap(α,x,y,A) <=> wf_eq_ap@(α,x,y,A)

* THM b_is_injection_
  >> α:Env -> MFun(α) -> Bool
  Extraction:
  λ α f. d(val_kind_pf(α,f);u.false;v.d(v;u.false;v.d(v;w.false;w.true)))

* THM b_is_injection_char
  >> ∀α:Env. ∀f:MFun(α). bool(b_is_injection(α,f)) <=> is_injection(α,f)

* THM wf_i_pair_aux_
```

```
  >> Object
  Extraction:
  λα. λ i x y.
  pb_of_decide(fun_atom@_decidable(α,i)) &
  λax. bool(b_is_injection(α,i))
       & pb_of_decide(Atom_list_eq_decidable([mtype(α,x)], mtype(α,i).1))
       & mtype(α,y)=mtype(α,i).2

* DEF wf_i_pair_aux
  wf_i_pair_aux(<a:Env>,<i:Atom>,<x:Term>,<y:Term>) ==
  wf_i_pair_aux(<a>)(<i>)(<x>)(<y>)

* THM wf_i_pair_aux__
  >> ∀α:Env. ∀i:Atom. ∀x,y:Term0.
       wf@(α,x) => wf@(α,y) => wf_i_pair_aux(α,i,x,y) in PBool

* THM wf_i_pair_aux_char
  >> ∀α:Env. ∀i:Atom. ∀x,y:Term0. wf@(α,x) => wf@(α,y) =>
       wf_i_pair_aux(α,i,x,y) <=> wf_i_pair@_aux(α,i,x,y)

* THM wf_i_pair_
  >> Object
  Extraction:
  λα. λ i x y.
  wf_i_pair_aux(α,i,x,y) &
  λax. prop( val(α,i)(val(α,x)) = val(α,y) in val(α,mtype(α,y)) )

* DEF wf_i_pair
  wf_i_pair(<a:Env>,<i:Atom>,<x:Term>,<y:Term>) == wf_i_pair(<a>)(<i>)(<x>)(<y>)


* THM wf_i_pair__
  >> ∀α:Env. ∀i:Atom. ∀x,y:Term0. wf@(α,x) => wf@(α,y) =>
       wf_i_pair(α,i,x,y) in PBool

* THM wf_i_pair_char
  >> ∀α:Env. ∀i:Atom. ∀x,y:Term0. wf@(α,x) => wf@(α,y) =>
       wf_i_pair(α,i,x,y) <=> wf_i_pair@(α,i,x,y)

* THM wf_term_
  >> Object
  Extraction:
  \ α.\ t.rec_ind(t; F,t.
  case t to U
    f,args -> ∀∈ args : Term0 list. F  &  λax. wf_fun_ap_aux(α,f,args)
    x,y,A  -> F(x) & F(y) & λax. wf_eq_ap(α,x,y,A)
```

```
  i,x,y  -> F(x) & F(y) & λax. wf_i_pair(α,i,x,y)
  n      -> bool(true))

* DEF wf_term
  wf(<a:Env>,<t:Term0>) == wf_term(<a>)(<t>)

* THM all_elements_mono_lemma
  >> ∀A:U. ∀P,Q:A->U. (∀a:A. P(a) => Q(a)) =>
         ∀x:A list. ∀∈ x : A list. P => ∀∈ x : A list. Q

* THM wf_term_lemma
  >> ∀α:Env. ∀t:Term0.
       wf(α,t) in PBool   &   wf(α,t) <=> wf@(α,t)

* THM wf_term_char
  >> ∀α:Env. ∀t:Term0. wf(α,t) <=> wf@(α,t)

* THM wf_term_
  >> ∀α:Env. ∀t:Term0. wf(α,t) in PBool
```

C.1.12 Monotonicity

```
* THM sub_fenv_
  >> γ:TEnv -> FEnv(γ) -> FEnv(γ) -> U
  Extraction:
  λ γ δ1 δ2. (δ1⊂δ2 ∈ (Atom#FEnvVal(γ) list))

* DEF sub_fenv
  <d1:FEnv>⊂{<g:TEnv>}<d2:FEnv> == sub_fenv(<g>)(<d1>)(<d2>)

* ML add_sub_tenv_membership_hack
  add_to_member_i `sub_tenv_membership_hack`
  (Progress
    (\p.(let [t],T = destruct_equal (concl p) in
         if is_spread_term t & ext_name T = `FEnv`
         then Assert (make_equal_term (get_type p t) [t])
              THENL [Id; OnLastHyp Inclusion]
         else fail
        ) p
    )
  )
  ;;
```

```
* THM subenv_
  >> Env -> Env -> U
  Extraction:
  λ α1 α2. α1.1 ⊂ α2.1  &  α1.2 ⊂{α2.1} α2.2

* DEF subenv
  <a1:Env>⊂<a2:Env> == subenv(<a1>)(<a2>)

* DEF subenv_ctxt
  ∀α1⊂α2. <P:*> == ∀α1,α2:Env. α1⊂α2 => <P>

* THM FEnvVal_mono
  >> ∀α1⊂α2. ∀v:FEnvVal(α1). v in FEnvVal(α2)

* ML add_FEnvVal_mono
  add_membership_mono_lemma_to_inclusion
  'FEnvVal_mono' 'FEnvVal'
  ;;

* THM MType_mono
  >> ∀α1⊂α2. ∀mt:MType(α1). mt in MType(α2)

* ML add_MType_mono
  add_membership_mono_lemma_to_inclusion
  'MType_mono' 'MType'
  ;;

* THM AtomicMType_mono
  >> ∀α1⊂α2. ∀a:AtomicMType(α1). a in AtomicMType(α2)

* ML add_AtomicMType_mono
  add_membership_mono_lemma_to_inclusion
  'AtomicMType_mono' 'AtomicMType'
  ;;

* THM tenv_ap_mono
  >> ∀α1⊂α2. ∀a:AtomicMType(α1). α1(a) = α2(a) in TEnvAp

* THM type_atom_val_mono
  >> ∀α1⊂α2. ∀a:AtomicMType(α1). val(α1,a) = val(α2,a) in SET

* THM type_atom_val_small_mono
  >> ∀α1⊂α2. ∀a:AtomicMType(α1). val(α1,a) = val(α2,a) in SmallEqSET

* THM type_atom_val_mem_mono
  >> ∀α1⊂α2. ∀a:AtomicMType(α1). ∀x:|val(α1,a)|. x in |val(α2,a)|
```

```
* THM mtype_dom_val_mono
  >> ∀α1⊂α2. ∀mt: { mt:MType(α1) | ¬(null(mt.1)) }.
       dom_val(α1,mt) = dom_val(α2,mt) in SET

* THM mtype_dom_val_mem_mono
  >> ∀α1⊂α2. ∀mt: { mt:MType(α1) | ¬(null(mt.1)) }.
       ∀x:|dom_val(α1,mt)|. x in |dom_val(α2,mt)|

* THM mtype_val_mono
  >> ∀α1⊂α2. ∀mt:MType(α1). val(α1,mt) = val(α2,mt) in SET

* THM mtype_val_mem_mono
  >> ∀α1⊂α2. ∀mt:MType(α1). ∀x:|val(α1,mt)|. x in |val(α2,mt)|

* THM bound_mono
  >> ∀A:U. ∀l1,l2:Atom#A list. (l1⊂l2 ∈ (Atom#A list)) => ∀a:Atom.
          ∃∈ l1 : Atom#A list. λu. a=u.1 in Atom
       => ∃∈ l2 : Atom#A list. λu. a=u.1 in Atom

* THM fun_atom@_mono
  >> ∀α1⊂α2. ∀a:Atom. fun_atom@(α1,a) => fun_atom@(α2,a)

* THM fun_atom_mono
  >> ∀α1⊂α2. ∀a:Atom. fun_atom(α1,a) => fun_atom(α2,a)

* THM MFun_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). f in MFun(α2)

* ML add_MFun_mono
  add_membership_mono_lemma_to_inclusion
  `MFun_mono` `MFun`
  ;;

* THM fenv_ap_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). α1(f) = α2(f) in FEnvVal(α2)

* THM mtype_mono
  >> ∀α1,α2:Env. α1⊂α2 => ∀f:MFun(α1). mtype(α1,f) = mtype(α2,f) in MType(α1)

* THM mtype_val_on_mtype_mem_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). ∀x:|val(α1,mtype(α1,f))|. x in |val(α2,mtype(α2,f))|

* ML add_some_mem_mono_lemmas
```

```
  add_membership_mono_lemma_to_inclusion `type_atom_val_mem_mono` `tos` ;;
  add_membership_mono_lemma_to_inclusion `mtype_dom_val_mem_mono` `tos` ;;
  add_membership_mono_lemma_to_inclusion `mtype_val_mem_mono` `tos` ;;
  add_membership_mono_lemma_to_inclusion
    `mtype_val_on_mtype_mem_mono` `tos` ;;

* THM mfun_val_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). val(α1,f) = val(α2,f) in |val(α1,mtype(α1,f))|

* THM val_kind_mem_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). ∀x:val_kind(α1,f). x in val_kind(α2,f)

* ML add_val_kind_mem_mono
  add_membership_mono_lemma_to_inclusion
  `val_kind_mem_mono` `val_kind`
  ;;

* THM val_kind_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). val_kind(α1,f) => val_kind(α2,f)

* THM val_kind_pf_mono
  >> ∀α1⊂α2. ∀f:MFun(α1).
        val_kind_pf(α1,f) = val_kind_pf(α2,f) in val_kind(α2,f)

* THM is_injection_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). is_injection(α1,f) => is_injection(α2,f)

* THM is_triv_pred_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). is_triv_pred(α1,f) => is_triv_pred(α2,f)

* THM is_semi_triv_pred_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). is_semi_triv_pred(α1,f) => is_semi_triv_pred(α2,f)


* THM Val_mono
  >> ∀α1⊂α2. ∀x:Val(α1). x in Val(α2)

* ML add_Val_mono
  add_membership_mono_lemma_to_inclusion
  `Val_mono` `Val`
  ;;

* THM val_member_eq_mono
  >> ∀α1⊂α2. ∀v:Val(α1). ∀a:AtomicMType(α1). v ∈{α1} a = v ∈{α2} a in U

* THM val_member_mono
```

```
  >> ∀α1⊂α2. ∀v:Val(α1). ∀a:AtomicMType(α1). v ∈{α1} a => v ∈{α2} a

* THM vals_in_mtypes@_mono
  >> ∀α1⊂α2. ∀vl:Val(α1) list. ∀aml:AtomicMType(α1) list.
        vl ∈{α1} aml => vl ∈{α2} aml

* THM term_mtype_mono
  >> ∀α1,α2:Env. α1⊂α2 => ∀t:Term(α1).
        mtype(α1,t) = mtype(α2,t) in AtomicMType(α1)

* THM term_mtype_mono_2
  >> ∀α1,α2:Env. α1⊂α2 => ∀t:Term(α1).
        mtype(α1,t) = mtype(α2,t) in Atom

* THM term_type_mono_lemma
  >> ∀α1⊂α2. ∀t:Term(α1). wf@(α2,t) => type(α1,t) = type(α2,t) in SET

* THM val_eq_when_val_member
  >> ∀α:Env. ∀A,B:AtomicMType(α). ∀a1,a2:|val(α,A)|.
        a1=a2 in |val(α,A)| => <A,a1> ∈{α} B => a1=a2 in |val(α,B)|

* THM List_inclusion_2_lemma
  >> ∀A:U. ∀Q:A->U. ∀l:{z:A|Q(z)} list.
        l in { l1:A list | ∀∈ l1 : A list. Q}

* THM List_inclusion_3_lemma
  >> ∀A:U. ∀Q:A->U. ∀l:{ l:A list | ∀∈ l : A list. Q }.
        l in {z:A|Q(z)} list

* THM arg_tuple_eq
  >> ∀α:Env. ∀P:Term0->U.
     ∀val1,val2: t:{t:Term0|wf@(α,t) & P(t)} -> |type(α,t)|.
     ∀t:{t:Term0|wf@(α,t) & P(t)}. val1(t)=val2(t) in |type(α,t)| =>
     ∀aml:AtomicMType(α) list. ¬(null(aml)) =>
         ∀l:{t:Term0|P(t)} list. |aml| = |l|
       => ∀∈ l : {t:Term0|P(t)} list. wf_term@(α)
       => (map λt. <mtype(α,t),val1(t)> on l to Val(α) list) ∈{α} aml
       => val1{α}(l) = val2{α}(l)
          in |#((map λa. val(α,a) on aml to SET list))|

* THM constant_mfun_val_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). null(mtype(α1,f).1) =>
        val(α1,f) = val(α2,f) in |val(α1,mtype(α1,f).2)|

* THM mfun_val_fun_mono
  >> ∀α1⊂α2. ∀f:MFun(α1). ¬(null(mtype(α1,f).1)) =>
```

```
      val(α1,f) = val(α2,f)
      in |dom_val(α1,mtype(α1,f))| -> |val(α1,mtype(α1,f).2)|

* ML add_some_omega_members_2
  add_to_member_i `some_omega_members`
  (IfOnConcl (\c. let [t],T = destruct_equal c in is_integer_term t & T='∞')

    (Refine (lemma `some_omega_members` `NIL`)
     THEN OnLastHyp FastRepeatAndE THEN Trivial)
    Fail
  )
  ;;

* THM term_val_mono_lemma
  >> ∀α1⊂α2. ∀t:Term(α1). wf@(α2,t) => val(α1,t) = val(α2,t) in |type(α1,t)|


* THM and_all_elements
  >> ∀A:U. ∀P,Q:A->U. ∀l:A list.
      ∀∈ l : A list. P & ∀∈ l : A list. Q <=> ∀∈ l : A list. λa. P(a) & Q(a)


* THM subset_if_all_elements
  >> ∀A:U. ∀P:A->U. ∀l:A list. ↓(∀∈ l : A list. P) => l in {a:A|P(a)} list

* THM wf_i_pair@_aux_mono_lemma
  >> ∀α1⊂α2. ∀i:Atom. ∀x,y:Term0.
      wf@(α1,x) => wf@(α1,y) => wf_i_pair@_aux(α1,i,x,y) =>
      wf@(α2,x) => wf@(α2,y) => wf_i_pair@_aux(α2,i,x,y) =>
      (val(α1,i)(val(α1,x)) = val(α1,y) in val(α1,mtype(α1,y)))
      = val(α2,i)(val(α2,x)) = val(α2,y) in val(α2,mtype(α2,y))
      in U

* THM val_member_mono_2
  >> ∀α1⊂α2. ∀t:Term0. wf@(α1,t) => wf@(α2,t) => ∀a:AtomicMType(α1).
      <mtype(α1,t),val(α1,t)> ∈{α1} a => <mtype(α2,t),val(α2,t)> ∈{α2} a

* THM wf_term@_mono
  >> ∀α1⊂α2. ∀t:Term0. wf@(α1,t) => wf@(α2,t)

* THM term_type_mono
  >> ∀α1⊂α2. ∀t:Term(α1). type(α1,t) = type(α2,t) in SET

* THM term_val_mono
  >> ∀α1⊂α2. ∀t:Term(α1). val(α1,t) = val(α2,t) in |type(α1,t)|
```

```
* THM Term_mono
  >> ∀α1⊂α2. ∀t:Term(α1). t in Term(α2)

* ML add_Term_mono
  add_membership_mono_lemma_to_inclusion
  `Term_mono` `Term`
  ;;

* THM term_list_type_mono
  >> ∀α1⊂α2. ∀l:Term(α1) list. ¬(null(l)) => type(α1,l) = type(α2,l) in SET


* THM term_list_type_mem_mono
  >> ∀α1⊂α2. ∀l:Term(α1) list. ¬(null(l)) => ∀x:|type(α1,l)|.
        x in |type(α2,l)|

* ML add_term_list_type_mem_mono
  add_membership_mono_lemma_to_inclusion `term_list_type_mem_mono` `tos` ;;

* THM terms_val_mono
  >> ∀α1⊂α2. ∀l:Term(α1) list. ¬(null(l)) =>
        val(α1,l) = val(α2,l) in |type(α1,l)|
```

C.1.13 Operations on Environments

```
* THM appended_alist_ap
  >> ∀A:U. ∀l1,l2:Atom#A list. ∀a:Atom.
        l1@l2{A}(a) = l1{A}(a) ? l2{A}(a) in ?A

* THM alist_values_char
  >> ∀A:U. ∀P:A->U. ∀l:Atom#A list.
      ∀a:Atom. if s(x)=l{A}(a) . P(x)
      <=> ∀a:Atom where ∃∈ l : Atom#A list. λx. a=x.1 in Atom.
            P(outl(l{A}(a)))

* THM cst_alists_
  >> A:U -> Atom#A list -> Atom#A list -> U
  Extraction:
  λ A l1 l2. ∀a:Atom. succeeds(l1{A}(a)) => succeeds(l2{A}(a)) =>
              l1{A}(a)=l2{A}(a) in ?A

* DEF cst_alists
  cst{<A:U>}(<l1:Atom#A list>,<l2:Atom#A list>) == cst_alists(<A>)(<l1>)(<l2>)
```

```
* THM cst_alists_suff_cond_
  >> A:U -> Atom#A list -> Atom#A list -> U
  Extraction:
  λ A l1 l2. ∀∈ l2 : Atom#A list. λy. if s(x)=l1{A}(y.1) . x=y.2 in A

* DEF cst_alists_suff_cond
  cst{<A:U>}(<l1:Atom#A list>,<l2:Atom#A list>) ==
  cst_alists_suff_cond(<A>)(<l1>)(<l2>)

* THM sub_alist_intro_lemma
  >> ∀A:Type. ∀x,y:?A. (succeeds(x) => succeeds(y) & x=y in ?A)
       => if s(a)=x . with s(b)=y . a=b in A

* THM sub_alist_intro
  >> ∀A:Type. ∀l1,l2:Atom#A list.
       (∀a:Atom. succeeds(l1{A}(a))
       => succeeds(l2{A}(a)) & l1{A}(a)=l2{A}(a) in ?A)
       => (l1⊂l2 ∈ (Atom#A list))

* THM catch_char_2
  >> ∀A:Type. ∀a,b:?A.
       (succeeds(a) & a?b = a in ?A) ∨ (fails(a) & a?b=b in ?A)

* THM catch_char_3
  >> ∀A:Type. ∀a,b:?A.
       (succeeds(a) => a?b = a in ?A) & (fails(a) => a?b=b in ?A)

* THM cst_alists_suff_cond_suffices
  >> ∀A:U. ∀l1,l2:Atom#A list. cst{A}(l1,l2)
       => cst{A}(l1,l2)

* THM sub_alist_of_append
  >> ∀A:U. ∀l1,l2:Atom#A list. (l1⊂l1@l2 ∈ (Atom#A list))
       & (cst{A}(l1,l2) => (l2⊂l1@l2 ∈ (Atom#A list)))

* THM sub_alist_char_2
  >> ∀A:U. ∀l1,l2:Atom#A list. (l1⊂l2 ∈ (Atom#A list)) => ∀a:Atom.
        succeeds(l1{A}(a))
        => succeeds(l2{A}(a)) & l1{A}(a) = l2{A}(a) in ?A

* THM sub_fenv_mono
  >> ∀γ1,γ2:TEnv. ∀δ1,δ2:FEnv(γ1). γ1⊂γ2 => δ1⊂{γ1}δ2 => δ1⊂{γ2}δ2

* THM cst_if_sub_alist
```

```
  >> ∀A:U. ∀l1,l2:Atom#A list. (l1⊂l2 ∈ (Atom#A list)) => cst{A}(l1,l2)

* THM append_to_sub_alist
  >> ∀A:U. ∀l1,l2:Atom#A list. (l1⊂l2 ∈ (Atom#A list))
        => (l1@l2⊂l2 ∈ (Atom#A list))
           & (l2⊂l1@l2 ∈ (Atom#A list))

* THM append_lub_wrt_sub_alist
  >> ∀A:U. ∀l1,l2,l3:Atom#A list.
      (l1⊂l3 ∈ (Atom#A list)) => (l2⊂l3 ∈ (Atom#A list))
      => (l1@l2⊂l3 ∈ (Atom#A list))

* THM cst_alists_sym
  >> ∀A:U. ∀l1,l2:Atom#A list. cst{A}(l1,l2) => cst{A}(l2,l1)

* THM append_when_cst_lemma
  >> ∀A:U. ∀l1,l2:Atom#A list. cst{A}(l1,l2) => (l1@l2⊂l2@l1 ∈ (Atom#A list))


* THM catch_succeeds_2
  >> ∀A:Type. ∀x,y:?A.
       succeeds(x?y) <=> succeeds(x) ∨ (fails(x) & succeeds(y))

* THM alist_cst_with_append
  >> ∀A:U. ∀l1,l2,l3:Atom#A list.
       cst{A}(l1,l2) => cst{A}(l1,l3) => cst{A}(l1,l2@l3)

* THM cst_tenvs_
  >> TEnv -> TEnv -> U
  Extraction:
  λ γ1 γ2. cst{TEnvVal}(γ1,γ2)

* DEF cst_tenvs
  cst(<l1:TEnv>,<l2:TEnv>) == cst_tenvs(<l1>)(<l2>)

* THM cst_fenvs_
  >> γ:TEnv -> FEnv(γ) -> FEnv(γ) -> U
  Extraction:
  λ γ δ1 δ2. cst{FEnvVal(γ)}(δ1,δ2)

* DEF cst_fenvs
  cst{<g:TEnv>}(<l1:FEnv>,<l2:FEnv>) == cst_fenvs(<g>)(<l1>)(<l2>)

* THM membership_of_FEnv_of_append
  >> ∀α1,α2:Env. cst(α1.1,α2.1) =>
       α1.2 in FEnv(α1.1@α2.1) & α2.2 in FEnv(α1.1@α2.1)
```

```
* ML add_membership_of_FEnv_of_append
  add_to_inclusion 'membership_of_FEnv_of_append'
  (\i.
   IfOnConcl (\c. let a,b = destruct_apply (eq_type c) in
                  ext_name a = 'FEnv' & ext_name b = 'append')
    (Progress (Lemma 'membership_of_FEnv_of_append' ...*))
    Fail
  )
  ;;

* THM cst_envs_
  >> Env -> Env -> U
  Extraction:
  λ α1 α2. cst(α1.1,α2.1) & cst{α1.1@α2.1}(α1.2,α2.2)

* DEF cst_envs
  cst(<l1:Env>,<l2:Env>) == cst_envs(<l1>)(<l2>)

* THM env_append_
  >> Object
  Extraction:
  λ α1 α2. < α1.1 @ α2.1, α1.2 @ α2.2 >

* DEF env_append
  <l:Env>@<l1:Env> == env_append(<l>)(<l1>)

* THM env_append__
  >> ∀α1,α2:Env. cst(α1,α2) => α1@α2 in Env

* THM subenv_of_append
  >> ∀α1,α2:Env.  cst(α1,α2)  =>  α1 ⊂ α1@α2 & α2 ⊂ α1@α2

* THM cst_if_subenv
  >> ∀α1,α2:Env. α1⊂α2 => cst(α1,α2)

* THM append_to_subenv
  >> ∀α1,α2:Env. α1⊂α2 => α1@α2 ⊂ α2 & α2 ⊂ α1@α2

* THM subenv_refl
  >> ∀α:Env. α⊂α

* THM subenv_trans
  >> ∀α1,α2,α3:Env. α1⊂α2 => α2⊂α3 => α1⊂α3

* THM cst_envs_refl
```

```
  >> ∀α:Env. cst(α,α)

* THM cst_fenvs_mono
  >> ∀γ1,γ2:TEnv. ∀δ1,δ2:FEnv(γ1). γ1⊂γ2 => cst{γ1}(δ1,δ2) => cst{γ2}(δ1,δ2)


* THM cst_envs_sym
  >> ∀α1,α2:Env. cst(α1,α2) => cst(α2,α1)

* THM env_cst_with_append
  >> ∀α1,α2,α3:Env. cst(α1,α2) => cst(α2,α3) => cst(α1,α3) => cst(α1,α2@α3)


* THM all_pairs_
  >> A:U -> (A->A->U) -> A list -> U
  Extraction:
  λ A P l. [ nil → True; h.t,Q → (∀∈ t : A list. P(h)) & Q; @ l]

* DEF all_pairs
  ∀(x,y)∈<l:A list>: <A:A:U>. <P:A-\>A-\>U>(x,y) == all_pairs(<A>)(<P>)(<l>)


* THM cst_env_list_
  >> Env list -> U
  Extraction:
  λl. ∀(x,y)∈l: Env. cst_envs(x,y)

* DEF cst_env_list
  cst(<l:Env list>) == cst_env_list(<l>)

* THM append_env_list_
  >> Object
  Extraction:
  λl. [ nil → <nil,nil>; h.t,v → h@v; @ l]

* DEF append_env_list
  @(<l:Env list>) == append_env_list(<l>)

* THM append_env_list_wf_lemma
  >> ∀l:Env list. cst(l) =>
       @(l) in Env  &  ∀α:Env. (∀∈ l : Env list. cst_envs(α)) => cst(α,@(l))


* THM append_env_list__
  >> ∀l:Env list. cst(l) => @(l) in Env
```

```
* THM env_cst_with_list_append
  >> ∀l:Env list. cst(l) => ∀α:Env.
       (∀∈ l : Env list. cst_envs(α)) => cst(α,@(l))

* THM member_subenv
  >> ∀l:Env list. ∀α:Env. cst(l) =>
       ∃∈ l : Env list. λx. x=α in Env => α ⊂ @(l)

* THM disjoint_alists_
  >> A:U -> Atom#A list -> Atom#A list -> U
  Extraction:
  λ A l1 l2. ∀∈ l1 : Atom#A list. λa. fails(l2{A}(a.1))

* DEF disjoint_alists
  disjoint{<A:U>}(<l1:Atom#A list>,<l2:Atom#A list>) ==
  disjoint_alists(<A>)(<l1>)(<l2>)

* THM cst_alists_if_disjoint
  >> ∀A:U. ∀l1,l2:Atom#A list. disjoint{A}(l1,l2) => cst{A}(l1,l2)

* THM all_elements_anti_mono
  >> ∀A:U. ∀P:A->U. ∀l1,l2:A list. (l1⊂l2 ∈ A list) =>
       ∀∈ l2 : A list. P => ∀∈ l1 : A list. P

* THM member_cst
  >> ∀l:Env list. ∀α:Env. cst(l) =>
       ∃∈ l : Env list. λx. x=α in Env => ∀∈ l : Env list. cst_envs(α)

* THM append_lub_wrt_subenv
  >> ∀α1,α2,α3:Env. α1⊂α3 => α2⊂α3 => α1@α2⊂α3

* THM sublist_of_cons
  >> ∀A:U. ∀h:A. ∀l:A list. (l⊂h.l ∈ A list)

* THM sublist_cst
  >> ∀l1,l2:Env list. (l1⊂l2 ∈ Env list) => cst(l2) => cst(l1) & @(l1)⊂@(l2)


* THM subseq_
  >> A:U -> A list -> A list -> U
  Extraction:
  λ A l1 l2.
  [ nil → null
  ; h2.uu,P → λl1. [ nil → True
                   ; h1.t1,uu → P(l1) ∨ (h1=h2 in A & P(t1))
                   ; @ l1]
```

```
; @ l2]
(l1)

* DEF subseq
  (<l1:A list>⊂⊂<l2:A list> ∈ <A:A:Ui> list) == subseq(<A>)(<l1>)(<l2>)

* THM sublist_if_subseq
  >> ∀A:U. ∀l2,l1:A list. (l1⊂⊂l2 ∈ A list) => (l1⊂l2 ∈ A list)
```

C.1.14 Lifting

```
* DEF tok
  <x:string> == "<x>"

* DEF constant
  <c:lifted constant> == <c>(nil)

* THM mark_
  >> Object
  Extraction:
  λ mark t. <mark,t>.2

* DEF val_kind_cases
  with <u:var> = <x:val_kind>.  none: <a:*>  triv: <b:*>
                                semi_triv: <c:*>  injection: <d:*> ==
  d(<x>;<u>.<a>;<u>.d(<u>;<u>.<b>;<u>.d(<u>;<u>.<c>;<u>.<d>)))

* THM make_fenv_unit_
  >> Object
  Extraction:
  λγ. λa. λ mt f vk. <a,<mt,f,vk>>

* DEF make_fenv_unit
  make_fenv_unit(<g:TEnv>,<a:atom>,<mt:mt:MType>,<f:val(mt)>,<vk:kind>) ==
  make_fenv_unit(<g>)(<a>)(<mt>)(<f>)(<vk>)

* THM make_fenv_unit__
  >> ∀γ:TEnv. ∀a:Atom. ∀mt:Atom list # Atom.
     all_type_atoms(γ,mt) => ∀f:|val(γ,mt)|. ∀vk:val_kind(γ,mt,f).
       make_fenv_unit(γ,a,mt,f,vk) in FEnvUnit(γ)

* THM make_simple_fenv_unit_
  >> Object
```

```
Extraction:
λγ. λ a mt f. make_fenv_unit(γ,a,mt,f,no_kind)

* DEF make_simple_fenv_unit
make_simple_fenv_unit(<g:TEnv>,<a:atom>,<mt:mt:MType>,<f:val(mt)>) ==
make_simple_fenv_unit(<g>)(<a>)(<mt>)(<f>)

* THM make_simple_fenv_unit_
>> ∀γ:TEnv. ∀a:Atom. ∀mt:Atom list # Atom. all_type_atoms(γ,mt) =>
   ∀f:|val(γ,mt)|.  make_simple_fenv_unit(γ,a,mt,f) in FEnvUnit(γ)

* ML EqI_make_simple_fenv_unit
let EqI_make_simple_fenv_unit =
  MemberI THENM EvalConcl THEN
  IfOnConcl (\c. `tos` = ext_name (eq_type c) ? false)
    (EvalConclExcept (defs `fun prod sot tos`) THEN OnLastHyp Thin)
    (Id...)
;;

* THM Constant_
>> Env -> U
Extraction:
λα. Atom # a:AtomicMType(α) # |val(α,a)|

* DEF Constant
Constant(<a:Env>) == Constant(<a>)

* THM make_constant_
>> Object
Extraction:
λα. λ a A v. <a,A,v>

* DEF make_constant
make_constant(<alpha:Env>,<a:Atom>,<A:A:AtomicMType>,<v:val(A)>) ==
make_constant(<alpha>)(<a>)(<A>)(<v>)

* THM make_constant_
>> ∀α:Env. ∀a,A:Atom. type_atom(α.1,A) => ∀v:|val(α,A)|.
     make_constant(α,a,A,v) in Constant(α)

* THM fenv_unit_of_constant_
>> α:Env -> Constant(α) -> FEnvUnit(α.1)
Extraction:
λ α c. let a,A,v = c in make_simple_fenv_unit(α.1, a, <nil,A>, v)

* DEF fenv_unit_of_constant
```

```
  fenv_unit(<a:Env>,<c:Constant>) == fenv_unit_of_constant(<a>)(<c>)

* THM add_constants_
  >> α:Env -> Constant(α) list -> Env
  Extraction:
  λ α l. <α.1, α.2 @ ((map fenv_unit_of_constant(α) on l
                            to FEnvUnit(α.1) list))
        >

* DEF add_constants
  add_constants(<a:Env>,<cl:Constant list>) == add_constants(<a>)(<cl>)

* THM subenv_of_add_constants
  >> ∀α1,α2:Env. ∀cl:Constant(α2) list. α1⊂α2 => α1 ⊂ add_constants(α2,cl)


* ML Lift_templates
  main_env_template := 'add_constants(@(l), cl)' ;;
  env_hyp_template := 'l in Env list & cst(l) & cl in Constant(@(l)) list' ;;

  subenv_hyp_template := 'a1⊂a2' ;;
  term_hyp_template :=
    't in Term0 & wf(α,t) & mtype(α,t) = "Prop" in Atom' ;;
  lift_template := 'val(α,t)' ;;
  term_val_type := 'α:Env->t:Term(α)->|type(α,t)|' ;;

* THM term_mtype_eval_lemma
  >> ∀α1,α2:Env. ∀t:Term0. α1=α2 in Env => wf(α1,t) =>
      mtype(α2,t) = "Prop" in Atom => mtype(α1,t) = "Prop" in Atom

* THM wf_term_eval_lemma
  >> ∀α1,α2:Env. ∀t:Term0. α1=α2 in Env => wf(α2,t) => wf(α1,t)

* THM term_val_eval_lemma_1
  >> ∀α1,α2:Env. ∀t:Term0. α1=α2 in Env => wf(α1,t) =>
      mtype(α1,t) = "Prop" in Atom => val(α2,t) => val(α1,t)

* THM term_val_eval_lemma_2
  >> ∀α1,α2:Env. ∀t:Term0. α1=α2 in Env => wf(α1,t) =>
      mtype(α1,t) = "Prop" in Atom => val(α1,t) => val(α2,t)

* THM all_elements_if_all_elements
  >> ∀A:U. ∀P:A->U. ∀l:A list. ∀a:A.
       ∃∈ l : A list. λx. x=a in A => P(a) => ∀∈ l : A list. P

* THM subenv_of_append_list
```

```
  >> ∀l:Env list. cst(l) => ∀∈ l : Env list. λα. α⊂@(l)

* THM subenv_of_append_list_lemma
  >> ∀α:Env. ∀l:Env list. cst(l) => ∀cl:Constant(@(l)) list.
      α=add_constants(@(l),cl) in Env => ∀∈ l : Env list. λx. x⊂α
```

C.1.15 Term Rewriting

The Type of Rewriting Functions

```
* ML restrict_inclusion
  Inclusion_restricted_p := true ;;

* THM eq_term_val_
  >> α:Env -> Term(α) -> Term(α) -> U
  Extraction:
  λ α t1 t2. mtype(α,t1) = mtype(α,t2) in Atom
             & val(α,t1)=val(α,t2) in type(α,t1)

* DEF eq_term_val
  val(<t1:Term>) ={<a:Env>} val(<t2:Term>) == eq_term_val(<a>)(<t1>)(<t2>)

* THM simple_val_member
  >> ∀α:Env. ∀v:Val(α). ∀B:AtomicMType(α). v.1=B in Atom => v ∈{α} B

* THM eq_term_val_char
  >> ∀α:Env. ∀t1,t2:Term(α).
      val(t1) ={α} val(t2)
      <=> mtype(α,t1)=mtype(α,t2) in Atom & val(α,t1=t2 in mtype(α,t1))

* THM eq_term_val_mono
  >> ∀α1,α2:Env. α1⊂α2 => ∀t1,t2:Term(α1).
       val(t1) ={α1} val(t2) => val(t1) ={α2} val(t2)

* THM eq_types_when_eq_mtypes
  >> ∀α:Env. ∀t1,t2:Term(α).
       mtype(α,t1)=mtype(α,t2) in Atom =>  type(α,t1)=type(α,t2) in SET

* THM mem_when_eq_mtype
  >> ∀α:Env. ∀t1,t2:Term(α).
       mtype(α,t1)=mtype(α,t2) in Atom => val(α,t1) in |type(α,t2)|

* THM eq_term_val_refl
```

```
  >> ∀α:Env. ∀t:Term(α). ↓(val(t) ={α} val(t))

* THM eq_term_val_sym
  >> ∀α:Env. ∀t1,t2:Term(α). ↓(val(t1) ={α} val(t2) => val(t2) ={α} val(t1))


* THM eq_term_val_trans
  >> ∀α:Env. ∀t1,t2,t3:Term(α).
       ↓( val(t1) ={α} val(t2) => val(t2) ={α} val(t3)
          => val(t1) ={α} val(t3) )

* THM val_inv_
  >> Env -> ( Term0 -> ?Term0 ) -> U
  Extraction:
  λ α f. ∀t1:Term0 where wf@(α,t1).
           if s(t2)=f(t1) . ↓(wf@(α,t2)) & ↓(val(t1) ={α} val(t2))

* DEF val_inv
  val_inv(<a:Env>,<f:Term0-\>?Term0>) == val_inv(<a>)(<f>)

* THM val_inv_triv
  >> ∀α:Env. ∀f:Term0->?Term0. ↓(val_inv(α,f)) => val_inv(α,f)

* THM Rewrite_
  >> Env -> U
  Extraction:
  λα.
  { f: Term0->?Term0 | ∀α2:Env where cst(α,α2). val_inv(α@α2,f) }

* DEF Rewrite
  Rewrite(<a:Env>) == Rewrite(<a>)

* THM Rewrite_eq_i
  >> ∀f:Term0->?Term0. ∀α1:Env.
       ↓(∀α2:Env where cst(α1,α2). val_inv(α1@α2,f))
       => f in Rewrite(α1)

* THM Rewrite_char
  >> ∀f:Term0->?Term0. ∀α1:Env. ∀α2:Env where cst(α1,α2).
       val_inv(α1@α2,f) <=> ∀α2:Env where α1⊂α2. val_inv(α2,f)

* THM Rewrite_mono
  >> ∀α2,α3:Env. α2⊂α3 => ∀f:Rewrite(α2). f in Rewrite(α3)
```

Some Simple Combinators

```
* THM rewrite_THEN_
  >> (Term0->?Term0) -> (Term0->?Term0) -> Term0 -> ?Term0
  Extraction:
  λ f g. λt. d(f(t);u.g(u);u.fail)

* DEF rewrite_THEN
  <f:Rewrite> THEN <g:Rewrite> == rewrite_THEN(<f>)(<g>)

* THM rewrite_THEN_wf_lemma
  >> ∀α:Env. ∀f,g:Term0->?Term0.
       ↓(val_inv(α,f)) => ↓(val_inv(α,g)) => ↓(val_inv(α,f THEN g))

* THM rewrite_THEN__
  >> ∀α:Env. ∀f,g:Rewrite(α). f THEN g in Rewrite(α)

* THM rewrite_ORELSE_
  >> (Term0->?Term0) -> (Term0->?Term0) -> Term0 -> ?Term0
  Extraction:
  λ f g. λt. d(f(t);u.s(u);u.g(t))

* DEF rewrite_ORELSE
  <f:Rewrite> ORELSE <g:Rewrite> == rewrite_ORELSE(<f>)(<g>)

* THM rewrite_ORELSE_wf_lemma
  >> ∀α:Env. ∀f,g:Term0->?Term0.
       ↓(val_inv(α,f)) => ↓(val_inv(α,g)) => ↓(val_inv(α,f ORELSE g))

* THM rewrite_ORELSE__
  >> ∀α:Env. ∀f,g:Rewrite(α). f ORELSE g in Rewrite(α)

* THM rewrite_Id_
  >> Term0 -> ?Term0
  Extraction:
  λt. s(t)

* DEF rewrite_Id
  Id == rewrite_Id

* THM rewrite_Id_wf_lemma
  >> ∀α:Env. val_inv(α,Id)

* THM rewrite_Id__
  >> ∀α:Env. Id in Rewrite(α)
```

```
* THM simple_ind_
  >> A:Type -> n:N -> A -> (A->A) -> A
  Extraction:
  λA. λ n a f. [ 0 → a ; n, y → f(y) ; @ n]

* DEF simple_ind
  simple_ind{<A:A:Type>}(<a:A>,<f:A-\>A>,<n:N>) ==
  simple_ind(<A>)(<n>)(<a>)(<f>)

* THM a_big_number_
  >> Int
  Extraction:
  9999999999999

* DEF a_big_number
  a_big_number == a_big_number

* THM fake_fixed_point_
  >> A:Type -> A -> (A->A) ->A
  Extraction:
  λ A a f. simple_ind{A}(a,f,a_big_number)

* DEF fake_fixed_point
  fix{<A:A:Type>}(<a:A>,<f:A-\>A>) == fake_fixed_point(<A>)(<a>)(<f>)

* THM fake_fixed_point_induction
  >> ∀A:Type. ∀a:A. ∀f:A->A. ∀P:A->Type.
        P(a) => (∀x:A. P(x) => P(f(x))) => P(fix{A}(a,f))

* THM rewrite_letrec_
  >> ( (Term0->?Term0) -> (Term0->?Term0) ) -> (Term0->?Term0)
  Extraction:
  λF. fix{(Term0->?Term0)}(Id, F)

* DEF rewrite_letrec
  letrec <f:var> = <t:body> == rewrite_letrec(λ<f>. <t>)

* THM rewrite_letrec_wf_lemma
  >> ∀α:Env. ∀F: (Term0->?Term0) -> (Term0->?Term0) .
        (∀f:Term0->?Term0. val_inv(α,f) => val_inv(α,F(f)))
        => val_inv(α,rewrite_letrec(F))

* THM rewrite_letrec__
  >> ∀α:Env. ∀F:Rewrite(α)->Rewrite(α). rewrite_letrec(F) in Rewrite(α)
```

Rewriting Subterms

```
* THM parallel_list_elim_2
  >> ∀A,B:Type. ∀P: l1:A list -> {l2:B list||l1|=|l2| in Int} -> Type .
       P(nil,nil) =>
       ( ∀h1:A. ∀h2:B. ∀t1:A list. ∀t2:B list.
           |t1|=|t2| in Int => P(t1,t2) => P(h1.t1, h2.t2) ) =>
       ∀l1:A list. ∀l2:B list. |l1|=|l2| in Int => P(l1,l2)

* THM parallel_list_elim_3
  >> ∀A,B:U. ∀P: l1:A list -> {l2:B list||l1|=|l2| in Int} -> U .
       P(nil,nil) =>
       ( ∀h1:A. ∀h2:B. ∀t1:A list. ∀t2:B list. |t1|=|t2| in Int => P(t1,t2)

            => (null(t1) & null(t2) => P(h1.t1, h2.t2))
               & ¬(null(t1)) & ¬(null(t2)) => P(h1.t1, h2.t2) ) =>
       ∀l1:A list. ∀l2:B list. |l1|=|l2| in Int => P(l1,l2)

* THM parallel_null_cases
  >> ∀A:U. ∀l1,l2:A list. |l1|=|l2| in Int =>
       null(l1) & null(l2) ∨ ¬(null(l1)) & ¬(null(l2))

* THM parallel3_null_cases
  >> ∀A,B,C:U. ∀l1:A list. ∀l2:B list. ∀l3:C list.
       |l1|=|l2| in Int => |l1|=|l3| in Int =>
       null(l1) & null(l2) & null(l3)
       ∨ ¬(null(l1)) & ¬(null(l2)) & ¬(null(l3))

* THM eq_term_val_when_val_member
  >> ∀α:Env. ∀t1,t2:Term(α). ∀A:AtomicMType(α).
       <mtype(α,t1),val(α,t1)> ∈{α} A => <mtype(α,t2),val(α,t2)> ∈{α} A
       => val(t1) ={α} val(t2) => ↓(val(α,t1)=val(α,t2) in val(α,A))

* THM length_when_vals_in_mtypes
  >> ∀α:Env. ∀l:Term(α) list. ∀aml:AtomicMType(α) list.
       (map λt. <mtype(α,t),val(α,t)> on l to Val(α) list) ∈{α} aml
       => |l|=|aml| in Int

* THM vals_in_mtypes_unroll
  >> ∀α:Env. ∀vl:Val(α) list. ∀aml:AtomicMType(α) list.
     ∀v:Val(α). ∀a:AtomicMType(α).
       v.vl ∈{α} a.aml => vl ∈{α} aml & v ∈{α} a
```

```
* THM terms_val_mem
  >> ∀α:Env. ∀l:Term(α) list. ∀aml:AtomicMType(α) list. ¬(null(l)) =>
       (map λt. <mtype(α,t),val(α,t)> on l to Val(α) list) ∈{α} aml
       => val(α,l) in |#((map type_atom_val(α) on aml to SET list))|


* THM terms_val_unroll
  >> ∀α:Env. ∀l:Term(α) list. ∀h:Term(α). ∀aml:AtomicMType(α) list.
       ¬(null(l))
       => (map λt. <mtype(α,t),val(α,t)> on h.l to Val(α) list) ∈{α} aml
       => val(α,h.l) = <val(α,h),val(α,l)>
          in |#((map type_atom_val(α) on aml to SET list))|

* THM fun_ap_arg_length
  >> ∀α:Env. ∀f:Atom. ∀args:Term0 list.
       wf@(α,f(args)) => |args|=|mtype(α,f).1| in Int

* THM fun_aps_arg_length
  >> ∀α:Env. ∀f:Atom. ∀args1,args2:Term0 list.
       wf@(α,f(args1)) => wf@(α,f(args2)) => |args1|=|args2| in Int

* THM fun_ap_functionality_lemma
  >> ∀α:Env. ∀aml:AtomicMType(α) list. ∀l1,l2:Term(α) list. ¬(null(l1)) =>

          (map λt. <mtype(α,t),val(α,t)> on l1 to Val(α) list) ∈{α} aml
       => (map λt. <mtype(α,t),val(α,t)> on l2 to Val(α) list) ∈{α} aml
       => ∀∈ com(l1,l2) : Term0 list. (λ t1,t2. val(t1) ={α} val(t2))
       => ↓(val(α,l1) = val(α,l2)
          in #((map type_atom_val(α) on aml to SET list)))

* THM fun_ap_functionality_lemma_2
  >> ∀α:Env. ∀f:Atom. ∀args1,args2:Term0 list. wf@(α,f(args1))
       => wf@(α,f(args2))
       => ¬(null(mtype(α,f).1))
       => ∀∈ com(args1,args2) : Term(α)#Term(α) list.
            (λ t1,t2. val(t1) ={α} val(t2))
       => ↓(val(α,args1) = val(α,args2) in dom_val(α,mtype(α,f)))

* THM mfun_val_fnl
  >> ∀α:Env. ∀f:MFun(α). ¬(null(mtype(α,f).1))
       => ↓(fnl{dom_val(α,mtype(α,f)), val(α,mtype(α,f).2)}(val(α,f)))

* THM fun_ap_functional
  >> ∀α:Env. ∀f:Atom. ∀args1,args2:Term0 list. wf@(α,f(args1))
       => wf@(α,f(args2))
```

```
      => ∀∈ com(args1,args2) : Term(α)#Term(α) list.
             (λ t1,t2. val(t1) ={α} val(t2))
      => ↓(val(f(args1)) ={α} val(f(args2)))

* THM rewrite_Sub_
  >> (Term0->?Term0) -> Term0 -> ?Term0
  Extraction:
  λg. λt. case t to ?Term0
    f,args -> let s(args2) = (map g on args to ?(Term0 list)) in
              s(f(args2)): ?Term0
    x,y,A  -> let s(x2), s(y2) = g(x), g(y) in s(x2=y2 in A): ?Term0
    i,x,y  -> let s(x2), s(y2) = g(x), g(y) in s(y2{i x2}): ?Term0
    n      -> s(n)

* DEF rewrite_Sub
  Sub(<f:Rewrite>) == rewrite_Sub(<f>)

* ML add_Term_subset_hack
  add_to_inclusion `Term_subset_hack`
  (\i p.
    if ht i p = 'Term0' & ext_name (eq_type (concl p)) = `Term`
    then (SetElementI THENM Trivial) p
    else fail
  )
  ;;

* THM vals_in_mtypes_roll
  >> ∀α:Env. ∀vl:Val(α) list. ∀aml:AtomicMType(α) list.
     ∀v:Val(α). ∀a:AtomicMType(α).
       vl ∈{α} aml & v ∈{α} a => v.vl ∈{α} a.aml

* THM val_member_when_eq_term_vals
  >> ∀α:Env. ∀t1,t2:Term(α). ∀A:AtomicMType(α).
       ↓(val(t1) ={α} val(t2)) => <mtype(α,t1),val(α,t1)> ∈{α} A
       => <mtype(α,t2),val(α,t2)> ∈{α} A

* THM vals_in_mtypes_on_eq_term_vals
  >> ∀α:Env. ∀aml:AtomicMType(α) list. ∀P:Term(α)->U.
     ∀g:{t:Term(α)|P(t)}->Term(α).
       (∀t:{t:Term(α)|P(t)}. ↓(val(t) ={α} val(g(t))))
       => ∀l:{t:Term(α)|P(t)} list.
             (map λt. <mtype(α,t),val(α,t)> on l to Val(α) list) ∈{α} aml
             => (map λt. <mtype(α,t),val(α,t)>
                     on (map g on l to Term(α) list)
                     to Val(α) list)
                ∈{α} aml
```

```
* THM members_when_wf_eq_ap
  >> ∀α:Env. ∀A:Atom. ∀u,v:Term0.
       wf@(α,u=v in A) => val(α,u) in |val(α,A)| & val(α,v) in |val(α,A)|


* THM eq_ap_fnl
  >> ∀α:Env. ∀t,u,v,w:Term0. ∀A:Atom.
       ↓(wf@(α,t=u in A)) => ↓(wf@(α,v=w in A))
       => ↓(val(t) ={α} val(v)) => ↓(val(u) ={α} val(w))
       => ↓(val(t=u in A) ={α} val(v=w in A))

* THM inj_ap_eq_lemma
  >> ∀S1,S2:SET. ∀i:|S1|->|S2|. ∀a1,b1:|S1|. ∀a2,b2:|S2|.
       eq_reln{|S2|:Ui_SET}(=_{S2}) => fnl{S1,S2}(i)
       => a1=b1 in S1 => a2=b2 in S2 => i(a1)=a2 in S2 => i(b1)=b2 in S2

* THM types_when_wf_i_pair
  >> ∀α:Env. ∀u,v:Term0. ∀i:Atom. wf@(α,v{i u})
       => type(α,u) = dom_val(α,mtype(α,i)) in SET
          & type(α,v) = val(α,mtype(α,i).2) in SET
          & val(α,i) in |type(α,u)|->|type(α,v)|

* THM wf_eq_ap_when_eq_term_val
  >> ∀α:Env. ∀i:Atom. ∀t1,t2,u1,u2:Term0.
       ↓(wf@(α,t2{i t1})) => ↓(wf@(α,u1)) => ↓(wf@(α,u2))
       => ↓(val(t1) ={α} val(u1)) => ↓(val(t2) ={α} val(u2))
       => ↓(wf@(α,u2{i u1}))

* THM rewrite_Sub_wf_lemma
  >> ∀α:Env. ∀g:Term0->?Term0. ↓(val_inv(α,g)) => ↓(val_inv(α, Sub(g)))

* THM rewrite_Sub_
  >> ∀α:Env. ∀f:Rewrite(α). Sub(f) in Rewrite(α)

* ML add_Rewrite_clauses_to_member_i
  add_to_member_i `Rewrite membership`
  (\p. if not 'Rewrite' = fst (destruct_apply (eq_type (concl p))) then fail
       if (ext_name (first_equand (concl p)) = `rewrite_from_eqn` ? false)
       then Id p
       else (ReduceConcl THEN TryMembershipThm append_underscore) p
  )
  ;;
```

More Simple Combinators

```
* THM rewrite_Progress_
  >> (Term0->?Term0) -> (Term0->?Term0)
  Extraction:
  λf. λx. let s(y) = f(x) in d(eq(x,y);u.fail;u.s(y)): ?Term0

* DEF rewrite_Progress
  Progress(<f:Rewrite>) == rewrite_Progress(<f>)

* THM rewrite_Progress_wf_lemma
  >> ∀α:Env. ∀f:Term0->?Term0. val_inv(α,f) => val_inv(α,Progress(f))

* THM rewrite_Progress__
  >> ∀α:Env. ∀f:Rewrite(α). Progress(f) in Rewrite(α)

* THM rewrite_Repeat_
  >> (Term0->?Term0) -> Term0 -> ?Term0
  Extraction:
  λf. fix{Term0->?Term0}(Id, λg. (Progress(f) THEN g) ORELSE Id)

* DEF rewrite_Repeat
  Repeat(<f:Rewrite>) == rewrite_Repeat(<f>)

* THM rewrite_Repeat_wf_lemma
  >> ∀α:Env. ∀f:Term0->?Term0. ↓(val_inv(α,f)) => ↓(val_inv(α, Repeat(f)))

* THM rewrite_Repeat__
  >> ∀α:Env. ∀f:Rewrite(α). Repeat(f) in Rewrite(α)

* THM rewrite_Try_
  >> (Term0->?Term0) -> (Term0->?Term0)
  Extraction:
  λf. λx. let res = f(x) in d(res;u.s(u);v.s(x))

* DEF rewrite_Try
  Try(<f:Rewrite>) == rewrite_Try(<f>)

* THM rewrite_Try_wf_lemma
  >> ∀α:Env. ∀f:Term0->?Term0. val_inv(α,f) => val_inv(α,Try(f))

* THM rewrite_Try__
  >> ∀α:Env. ∀f:Rewrite(α). Try(f) in Rewrite(α)

* THM TopDown_
```

```
  >> Object
  Extraction:
  λf. letrec g = Try(f) THEN Sub(g)

* DEF TopDown
  TopDown(<f:Rewrite>) == TopDown(<f>)

* THM TopDown__
  >> ∀α:Env. ∀f:Rewrite(α). TopDown(f) in Rewrite(α)

* THM BotUp_
  >> Object
  Extraction:
  λf. letrec g = Sub(g) THEN Try(f)

* DEF BotUp
  BotUp(<f:Rewrite>) == BotUp(<f>)

* THM BotUp__
  >> ∀α:Env. ∀f:Rewrite(α). BotUp(f) in Rewrite(α)

* THM Topmost_
  >> Object
  Extraction:
  λf. letrec g = f ORELSE Sub(g)

* DEF Topmost
  Topmost(<f:Rewrite>) == Topmost(<f>)

* THM Topmost__
  >> ∀α:Env. ∀f:Rewrite(α). Topmost(f) in Rewrite(α)
```

Rewriting Trivially True Equalities

```
* THM eq_term_val_if_wf_
  >> α:Env -> Term0 -> Term0 -> U
  Extraction:
  λ α t1 t2. ↓(wf@(α,t1)) => ↓(wf@(α,t2)) => ↓(val(t1) ={α} val(t2))

* DEF eq_term_val_if_wf
  (wf => val(<t1:Term>) ={<a:Env>} val(<t2:Term>) ==
  eq_term_val_if_wf(<a>)(<t1>)(<t2>)
```

```
* THM is_btrue_decidable
  >> ∀b:Bool. b ∨ ¬(b)

* ML add_is_btrue_decidable
  add_to_Decidable `is_btrue_decidable`
  (Lemma `is_btrue_decidable`)
  ;;

* THM fun_ap_functional_2
  >> ∀α:Env. ∀f1,f2:Atom. f1=f2 in Atom => ∀args1,args2:Term0 list.
        wf@(α,f1(args1)) => wf@(α,f2(args2))
        => ∀∈ com(args1,args2) : Term(α)#Term(α) list.
             (λ t1,t2. val(t1) ={α} val(t2))
        => ↓(val(f1(args1)) ={α} val(f2(args2)))

* ML add_poly_clause_to_autotactic
  add_to_autotactic `poly_clause`
  (\p.
    let [t],T = destruct_equal (concl p) in
    assert almost_poly_defined_term t ;
    (Lemma o append_underscore o destruct_term_of_theorem
       o hd o decompose_ap) t p
  )
  ;;

* THM all_members_of_com
  >> ∀A1,A2:U. ∀P1:A1->U. ∀P2:A2->U. ∀P3,P4:(A1#A2)->U.
     ∀l1:A1 list. ∀l2:A2 list.
       |l1|=|l2| in Int => ∀∈ l1 : A1 list. P1 => ∀∈ l2 : A2 list. P2
       => ∀∈ com(l1,l2) : A1#A2 list. P3
       => (∀a1:A1. ∀a2:A2. P1(a1) & P2(a2) & P3(a1,a2) => P4(a1,a2))
       => ∀∈ com(l1,l2) : A1#A2 list. P4

* THM com_all_elements
  >> ∀A1,A2:U. ∀P1:A1->U. ∀P2:A2->U. ∀l1:A1 list. ∀l2:A2 list.
       |l1|=|l2| in Int => ∀∈ l1 : A1 list. P1 => ∀∈ l2 : A2 list. P2
       => ∀∈ com(l1,l2) : A1#A2 list. λ x1,x2. P1(x1) & P2(x2)

* THM wf_i_pair_fnlty
  >> ∀α:Env. ∀i:Atom. ∀u,v:Term0.
       ↓(wf@(α,v{i u})) => ↓(fnl{type(α,u),type(α,v)}(val(α,i)))

* THM wf_i_pair_injectiveness
  >> ∀α:Env. ∀i:Atom. ∀u,v:Term0.
       ↓(wf@(α,v{i u})) => ↓(injective(val(α,i)∈type(α,u)->type(α,v)))
```

```
* THM inj_ap_eq_lemma_2
  >> ∀S1,S2:SET. ∀i:|S1->S2|. ∀a1,b1:|S1|. ∀a2,b2:|S2|.
       eq_reln{|S2|:Ui_SET}(=_{S2}) => injective(i∈S1->S2)
       => a2=b2 in S2 => i(a1)=a2 in S2 => i(b1)=b2 in S2 => a1=b1 in S1

* THM inj_ap_when_wf_i_pair
  >> ∀α:Env. ∀i:Atom. ∀u,v:Term0. wf@(α,v{i u}) =>
      val(α,i)(val(α,u)) = val(α,v) in type(α,v)

* THM i_pair_fnl
  >> ∀α:Env. ∀i1,i2:Atom. ∀u1,v1,u2,v2:Term0. i1=i2 in Atom =>
       ↓(wf@(α,v1{i1 u1})) => ↓(wf@(α,v2{i2 u2})) => ↓(val(v1) ={α} val(v2))

       => ↓(val(v1{i1 u1}) ={α} val(v2{i2 u2}))

* THM eq_term_
  >> ∀t1,t2:Term0. ∃b:Bool where ∀α:Env. ↓(b) => (wf => val(t1) ={α} val(t2)

  Extraction:
  . . .

* DEF eq_term
  eq(<t1:Term0>,<t2:Term0>) == eq_term(<t1>)(<t2>)

* THM rewrite_true_eq_
  >> Term0 -> ?Term0
  Extraction:
  λt.
  case t to ?Term0
    f,l -> fail
    x,y,A -> let s(z) = eq(x,y) in s(True): ?Term0
    i,x,y -> fail
    n -> fail

* DEF rewrite_true_eq
  true_eq == rewrite_true_eq

* ML add_eq_term_to_member_i
  add_to_member_i `eq_term`
  (\p. let t.(), T = destruct_equal (concl p) in
       if is_set_term T & ext_name t = `eq_term` then EqI p
       else fail
  )
  ;;

* THM eq_true_when_eq_term_val
```

```
  >> ∀α:Env. ∀A:Atom. ∀u,v:Term0.
       ↓(wf@(α,u=v in A)) => ↓(val(u) ={α} val(v)) => ↓(val(α,u=v in A))

* THM rewrite_true_eq_wf_lemma
  >> ∀α:Env. val_inv(α,rewrite_true_eq)

* THM rewrite_true_eq__
  >> ∀α:Env. true_eq in Rewrite(α)
```

Meta-Term Substitution

```
* THM b_null_
  >> Object
  Extraction:
  λl. if null(l) then true else false

* DEF b_null
  null == b_null

* THM b_null__
  >> ∀A:Type. ∀l:A list. null(l) in Bool

* THM b_is_constant_
  >> Term0 -> Bool
  Extraction:
  λt. let (s(x)=df(t)) ? false in null((x).2)

* DEF b_is_constant
  is_constant(<t:Term0>) == b_is_constant(<t>)

* THM id_of_constant_
  >> Object
  Extraction:
  λt. ds(df(t)).1

* DEF id_of_constant
  id_of(<t:Term0>) == id_of_constant(<t>)

* THM id_of_constant__
  >> ∀t:Term0. is_constant(t) => id_of(t) in Atom

* THM f_sub_rec_
  >> (Term0->?Term0->?Term0) -> Term0 -> ?Term0
```

```
  Extraction:
  λ g t. rec_ind(t; h,t. g(t, f_sub_map(h,t)))

* DEF f_sub_rec
  sub_rec(<f:Term0-\>?Term0-\>?Term0>,<t:Term0>) == f_sub_rec(<f>)(<t>)

* THM sub_rec_
  >> (Term0->Term0->Term0) -> Term0 -> Term0
  Extraction:
  λ g t. rec_ind(t; h,t. g(t, sub_map(h,t)))

* DEF sub_rec
  sub_rec(<f:Term0-\>Term0-\>Term0>,<t:Term0>) == sub_rec(<f>)(<t>)

* THM subst_
  >> Atom#Term0 list -> Term0 -> Term0
  Extraction:
  λ alist t.
  sub_rec(λ t sub_mapped_t.
            if is_constant(t) then
              λax. let (s(t2)=alist{Term0}(id_of(t))) ? t in t2
            else λax. sub_mapped_t ,
          t)

* DEF subst
  <l:Atom#Term0 list>(<t:Term0>) == subst(<l>)(<t>)

* THM occurs_
  >> Atom -> Term0 -> U1
  Extraction:
  λ id t. rec_ind(t; Q,t.
  case t to U1
    f,l -> (null(l) & id=f in Atom) ∨ ∃∈ l : Atom list. Q
    x,y,A -> Q(x) ∨ Q(y)
    i,x,y -> Q(x) ∨ Q(y)
    n -> False)

* DEF occurs
  <x:Atom>∈<t:Term0> == occurs(<x>)(<t>)

* THM covering_subst_
  >> Atom list -> (Atom#Term0 list) -> Term0 -> U1
  Extraction:
  λ vars subst t.
  ∀a:Atom. ∃∈ vars : Atom list. λx. a=x in Atom =>
    a∈t => ∃∈ subst : Atom#Term0 list. λx. a=x.1 in Atom
```

```
* DEF covering_subst
  (<al:subst> covers <t:Term0> wrt <vl:Atom list>) ==
  covering_subst(<vl>)(<al>)(<t>)

* THM limited_subst_
  >> Atom list -> Atom#Term0 list -> U1
  Extraction:
  λ vars s. ∀∈ s : Atom#Term0 list.
              λpr. ∃∈ vars : Atom list. λvar. pr.1=var in Atom

* DEF limited_subst
  <s:subst> limited to <vars:ids> == limited_subst(<vars>)(<s>)

* THM complete_subst_
  >> Atom list -> Atom#Term0 list -> Term0 -> U1
  Extraction:
  λvars. λ s t. (s covers t wrt vars) & s limited to vars

* DEF complete_subst
  (<subst:subst>{<vars:Atom list>} complete on <t1:Term0>) ==
  complete_subst(<vars>)(<subst>)(<t1>)

* THM f_identical_terms_
  >> ∀t1,t2:Term0. ?(t1=t2 in Term0)
  Extraction:
  . . .

* DEF f_identical_terms
  <t:Term0>≡<tt:Term0> == f_identical_terms(<t>)(<tt>)

* THM f_subst_cst_
  >> ∀subst1,subst2:Atom#Term0 list. ?(↓(cst{Term0}(subst1,subst2)))
  Extraction:
  . . .

* DEF f_subst_cst
  cst(<s:subst>,<ss:subst>) == f_subst_cst(<s>)(<ss>)

* THM covering_subst_mono
  >> ∀vars:Atom list. ∀s1,s2:Atom#Term0 list. (s1⊂s2 ∈ (Atom#Term0 list))
        => ∀t:Term0. (s1 covers t wrt vars) => (s2 covers t wrt vars)

* THM subst_succeeds
  >> ∀vars:Atom list. ∀s:Atom#Term0 list. ∀t:Term0. is_constant(t)
        => (s covers t wrt vars)
```

```
        => ∃∈ vars : Atom list. λx. id_of(t)=x in Atom
        => succeeds(s{Term0}(id_of(t)))

* THM subst_base
  >> ∀s:Atom#Term0 list. ∀t:Term0. is_constant(t) =>
        succeeds(s{Term0}(id_of(t)))
        => s(t) = ds(s{Term0}(id_of(t))) in Term0

* THM subst_fails
  >> ∀vars:Atom list. ∀s:Atom#Term0 list. ∀t:Term0. is_constant(t)
        => s limited to vars => ¬(∃∈ vars : Atom list. λx. id_of(t)=x in Atom)

        => fails(s{Term0}(id_of(t)))

* THM subst_base_2
  >> ∀s:Atom#Term0 list. ∀t:Term0. is_constant(t) => fails(s{Term0}(id_of(t)))

        => s(t) = t in Term0

* THM subst_on_fun_ap
  >> ∀f:Atom. ∀l:Term0 list. ¬(null(l)) => ∀s:Atom#Term0 list.
        s(f(l)) = f((map subst(s) on l to Term0 list)) in Term0

* THM subst_on_eq_ap
  >> ∀A:Atom. ∀u,v:Term0. ∀s:Atom#Term0 list.
        s(u=v in A) = (s(u)=s(v) in A) in Term0

* THM subst_on_i_pair
  >> ∀i:Atom. ∀u,v:Term0. ∀s:Atom#Term0 list.
        s(v{i u}) = (s(v){i s(u)}) in Term0

* THM covering_subst_hereditary
  >> ∀vars:Atom list. ∀s:Atom#Term0 list.
       ∀f:Atom. ∀l:Term0 list. (s covers f(l) wrt vars)
           => ∀∈ l : Term0 list. λt. (s covers t wrt vars)
       & ∀t,u:Term0. ∀A:Atom. (s covers t=u in A wrt vars)
           => (s covers t wrt vars) & (s covers u wrt vars)
       & ∀i:Atom. ∀t,u:Term0. (s covers u{i t} wrt vars)
           => (s covers t wrt vars) & (s covers u wrt vars)

* THM subst_mono
  >> ∀vars:Atom list. ∀s1,s2:Atom#Term0 list. ↓((s1⊂s2 ∈ (Atom#Term0 list)))

        => ∀t:Term0. ↓((s1{vars} complete on t))
        => ↓((s2{vars} complete on t))
        => s1(t)=s2(t) in Term0
```

```
* THM limited_appended_subst
  >> ∀vars:Atom list. ∀s2,s1:Atom#Term0 list.
       s1 limited to vars => s2 limited to vars => s1@s2 limited to vars

* THM list_subst_lemma
  >> ∀vars:Atom list. ∀l1:Term0 list.
       ∀∈ l1 : Term0 list.
         (λt1. ∀t2:Term0. ?∃s:Atom#Term0 list
             where (s{vars} complete on t1) & s(t1)=t2 in Term0)
       => ∀l2:Term0 list. ?∃s:Atom#Term0 list where
             ∀∈ l1 : Term0 list. λt. (s{vars} complete on t)
             & (map subst(s) on l1 to Term0 list) = l2 in Term0 list

* THM complete_subst_on_fun_ap
  >> ∀vars:Atom list. ∀f:Atom. ∀l:Term0 list. ¬(null(l)) =>
     ∀s:Atom#Term0 list.
       ∀∈ l : Term0 list. λt. (s{vars} complete on t)
       => (s{vars} complete on f(l))

* THM complete_subst_on_eq_ap
  >> ∀vars:Atom list. ∀A:Atom. ∀u,v:Term0. ∀s:Atom#Term0 list.
       (s{vars} complete on u) => (s{vars} complete on v)
       => (s{vars} complete on (u=v in A))

* THM complete_subst_on_i_pair
  >> ∀vars:Atom list. ∀i:Atom. ∀u,v:Term0. ∀s:Atom#Term0 list.
       (s{vars} complete on u) => (s{vars} complete on v)
       => (s{vars} complete on (v{i u}))
```

Obtaining Rewriting Functions from Equations

```
* THM match_
  >> ∀vars:Atom list. ∀t1,t2:Term0.
     ?(∃s:Atom#Term0 list where (s{vars} complete on t1) & s(t1)=t2 in Term0)

  Extraction:
  . . .

* DEF match
  match(<p:pattern>,<i:instance>,<ids:ids>) == match(<ids>,<p>,<i>)

* THM rewrite_from_eqn_
```

```
  >> Atom list -> Term0 -> Term0 -> (Term0->?Term0)
  Extraction:
  λ vars u v. λt. let s(subst) = match(u,t,vars) in s(subst(v)): ?Term0

* DEF rewrite_from_eqn
  rewrite{<ids:ids>}(<lhs:Term0>-\><rhs:Term0>) ==
  rewrite_from_eqn(<ids>,<lhs>,<rhs>)

* THM full_subst_
  >> Atom list -> Atom#Term0 list -> U1
  Extraction:
  λ vars s. ∃l:Term0 list where |vars|=|l| in Int
                                & s = com(vars,l) in Atom#Term0 list

* DEF full_subst
  <s:subst> full over <v:ids> == full_subst(<v>)(<s>)

* THM full_subst_from_subst
  >> ∀vars:Atom list. ∀s1:Atom#Term0 list. ∃s2:Atom#Term0 list where
       s2 full over vars &
       ∀∈ vars : Atom list. λa.
           succeeds(s1{Term0}(a)) => s1{Term0}(a) = s2{Term0}(a) in ?Term0 &

           fails(s1{Term0}(a)) => s2{Term0}(a) = s(a) in ?Term0

* THM length_of_com
  >> ∀A1,A2:U. ∀l1:A1 list. ∀l2:A2 list. |l1|=|l2| in Int =>
       |com(l1,l2)|=|l1| in Int & |com(l1,l2)|=|l2| in Int

* THM limited_subst_if_full
  >> ∀vars:Atom list. ∀s:Atom#Term0 list.
       s full over vars => s limited to vars

* THM all_elements_elim
  >> ∀A:U. ∀P:A->U. ∀l:A list.
       (∀∈ l : A list. P) => ∀a:A. (∃∈ l : A list. λx. a=x in A) => P(a)

* THM constant_char
  >> ∀t:Term0. is_constant(t) => t=id_of(t) in Term0

* THM limited_subst_failure
  >> ∀vars:Atom list. ∀s:Atom#Term0 list. s limited to vars =>
       ∀a:Atom. ¬(∃∈ vars : Atom list. λx. a=x in Atom) => fails(s{Term0}(a))


* THM full_subst_from_limited_subst
```

```
  >> ∀vars:Atom list. ∀s1:Atom#Term0 list. ↓(s1 limited to vars) =>
        ∃s2:Atom#Term0 list where s2 full over vars
                                  & ∀t:Term0. s1(t)=s2(t) in Term0

* THM rewrite_from_eqn_lemma
  >> ∀α1:Env. ∀vars:Atom list. ∀u,v:Term0.
        ∀α2:Env. cst(α1,α2) => ∀s:Atom#Term0 list.
                 (s{vars} complete on u) => wf@(α1@α2,s(u))
                 => ↓(wf@(α1@α2,s(v))) & ↓(val(s(u)) ={α1@α2} val(s(v)))
        => ∀α2:Env. cst(α1,α2) => val_inv(α1@α2,rewrite{vars}(u->v))

* THM rewrite_from_eqn_lemma_2
  >> ∀α1:Env. ∀vars:Atom list. ∀u,v:Term0.
        ∀α2:Env. cst(α1,α2) => ∀s:Atom#Term0 list.
                 s full over vars => wf@(α1@α2,s(u))
                 => ↓(wf@(α1@α2,s(v))) & ↓(val(s(u)) ={α1@α2} val(s(v)))
        => ∀α2:Env. cst(α1,α2) => val_inv(α1@α2,rewrite{vars}(u->v))

* THM rewrite_from_eqn__
  >> ∀α1:Env. ∀vars:Atom list. ∀u,v:Term0.
        ↓(∀α2:Env. cst(α1,α2) => ∀s:Atom#Term0 list.
                 s full over vars => wf@(α1@α2,s(u))
                 => ↓(wf@(α1@α2,s(v))) & ↓(val(s(u)) ={α1@α2} val(s(v))))
        => rewrite{vars}(u->v) in Rewrite(α1)

* THM parallel_list_unroll
  >> ∀A,B:Type. ∀P: l1:A list -> {l2:B list||l1|=|l2| in Int} -> Type .
        P(nil,nil) =>
        ∀h1:A. ∀h2:B. ∀t1:A list. ∀t2:B list.
           |t1|=|t2| in Int => P(h1.t1, h2.t2)  =>
        ∀l1:A list. ∀l2:B list. |l1|=|l2| in Int => P(l1,l2)

* THM full_1subst
  >> ∀x:Atom. ∀s:Atom#Term0 list. s full over [x] =>
        ∃t:Term0 where s=[<x,t>] in Atom#Term0 list

* THM full_2subst
  >> ∀x,y:Atom. ∀s:Atom#Term0 list. s full over [x;y]
        => ∃t,u:Term0 where s=[<x,t>;<y,u>] in Atom#Term0 list

* DEF list3
  [<x:*>;<y:*>;<z:*>] == ((<x>).(<y>).(<z>).nil)

* THM full_3subst
  >> ∀x,y,z:Atom. ∀s:Atom#Term0 list. s full over [x;y;z]
        => ∃t:Term0. ∃u,v:Term0 where s = [<x,t>;<y,u>;<z,v>]
```

```
                                         in Atom#Term0 list

* THM rewrite_from_1eqn
  >> ∀α1:Env. ∀x:Atom. ∀u,v:Term0.
       ↓(∀α2:Env. cst(α1,α2) => ∀t:Term0.
          let s = [<x,t>] in
          wf@(α1@α2,s(u)) =>
          ↓(wf@(α1@α2,s(v))) & ↓(val(s(u)) ={α1@α2} val(s(v))))
       => rewrite{[x]}(u->v) in Rewrite(α1)


* THM rewrite_from_2eqn
  >> ∀α1:Env. ∀x,y:Atom. ∀u,v:Term0.
       ↓(∀α2:Env. cst(α1,α2) => ∀t1,t2:Term0.
          let s = [<x,t1>;<y,t2>] in
          wf@(α1@α2,s(u)) =>
          ↓(wf@(α1@α2,s(v))) & ↓(val(s(u)) ={α1@α2} val(s(v))))
       => rewrite{[x;y]}(u->v) in Rewrite(α1)


* THM rewrite_from_3eqn
  >> ∀α1:Env. ∀x,y,z:Atom. ∀u,v:Term0.
       ↓(∀α2:Env. cst(α1,α2) => ∀t1,t2,t3:Term0.
          let s = [<x,t1>;<y,t2>;<z,t3>] in
          wf@(α1@α2,s(u)) =>
          ↓(wf@(α1@α2,s(v))) & ↓(val(s(u)) ={α1@α2} val(s(v))))
       => rewrite{[x;y;z]}(u->v) in Rewrite(α1)
```

Applying Rewriting Functions

```
* THM special_ap_
  >> Object
  Extraction:
  λ f x. f(x)

* THM val_member_char_2
  >> ∀α:Env. ∀t:Term(α). ∀A:AtomicMType(α).
       <mtype(α,t),val(α,t)> ∈{α} A => val(α,t) in |val(α,A)|

* THM type_when_prop
  >> ∀α:Env. ∀t:Term(α). mtype(α,t) = "Prop" in Atom => val(α,t) in U
```

```
* THM eq_term_val_when_props
  >> ∀α:Env. ∀x,y:Term(α). mtype(α,x) = "Prop" in Atom =>
       val(x) ={α} val(y) =>
       mtype(α,y) = "Prop" in Atom & val(α,x) in U
       & val(α,y) in U & val(α,x) <=> val(α,y)

* THM trivial_env_append
  >> ∀α:Env. α = α@<nil,nil> in Env

* THM trivially_cst_envs
  >> ∀α:Env. cst(α,<nil,nil>)

* THM rewrite_ap_lemma
  >> ∀α:Env. ∀f:Rewrite(α). ∀x:Term0.
        ↓(wf(α,x)) => mtype(α,x) = "Prop" in Atom =>
        if s(y)=f(x) . y in Term0 & ↓(wf(α,y)) & mtype(α,y) = "Prop" in Atom

                        & ↓(val(α,x)) <=> ↓(val(α,y))


* THM triv_eq_pf_
  >> α:Env -> A:AtomicMType(α) -> ?triv_eq(val(α,A))
  Extraction:
  λ α A. α(A).2

* DEF triv_eq_pf
  triv_eq_pf(<a:Env>,<A:AtomicMType>) == triv_eq_pf(<a>,<A>)

* THM triv_eq_pf_char
  >> ∀α:Env. ∀A:AtomicMType(α). succeeds(triv_eq_pf(α,A)) => triv_eq(val(α,A))


* THM triv_term_truth
  >> ∀α:Env. ∀t:Term(α). ?( mtype(α,t)="Prop" in Atom & ↓(t{α}) => t{α} )

* THM triv_term_truth_2
  >> ∀α1,α2:Env. α1=α2 in Env => ∀t:Term0.
       ↓(wf(α1,t)) => succeeds(triv_term_truth(α2,t))
       => mtype(α1,t)="Prop" in Atom & ↓(val(α1,t)) => val(α1,t)

* THM Int_plus_
  >> Int->Int->Int
  Extraction:
  λ x y. x+y

* DEF Int_plus
```

```
<x>+<y> == Int_plus(<x>,<y>)

* DEF Int_plus_m
  <x>+<y> == "Int_plus"(<x>.<y>.nil)

* THM Int_plus_v
  >> FEnvUnit(nil)

* THM a_Int_
  >> Env
  Extraction:
  <nil,
    Int_plus_v
    .nil
   >

* THM rewrite_test
  >> rewrite{[x;y]}(x+y -> y+x)
       in Rewrite(a_Int)

* ML add_to_member_i
  add_to_member_i `Rewrite membership`
  (\p. if 'Rewrite' = fst (destruct_apply (eq_type (concl p))) then
          (ReduceConcl THEN ApplyMembershipThm append_underscore) p
       else fail
  )
  ;;
```

C.1.16 Equality and Monoids

The Equality Tactic

```
* THM PropTerm_
  >> Env -> U
  Extraction:
  λα. { t:Term(α) | mtype(α,t)="Prop" in Atom }

* DEF PropTerm
  PropTerm(<a:Env>) == PropTerm(<a>)

* THM true_prop_term_
  >> α:Env -> PropTerm(α) -> U
  Extraction:
```

```
  λ α t. val(α,t)

* DEF true_prop_term
  <t:PropTerm>{<a:Env>} == true_prop_term(<a>,<t>)

* THM Complete_
  >> Env -> U
  Extraction:
  λα. ∀hyps: {t:PropTerm(α)|t{α}} list. ∀concl:PropTerm(α). ?↓(concl{α})

* DEF Complete
  Complete(<a:Env>) == Complete(<a>)

* THM complete_ap_lemma
  >> ∀α:Env. ∀f:Complete(α). ∀hyps:Term0 list. ∀concl:Term0.
      ∀∈ hyps : Term0 list.
         (λt. ↓(wf(α,t)) & mtype(α,t)="Prop" in Atom & val(α,t))
      => ↓(wf(α,concl)) & mtype(α,concl)="Prop" in Atom
      => succeeds(f(hyps,concl))
      => ↓(val(α,concl))

* THM term_is_eq
  >> ∀α:Env. ∀t:Term(α).
     ? ∃x,y:Term(α). ∃A:Atom where wf@(α,x=y in A)
                                  & t = (x=y in A) in Term(α)

* THM term_in_mtype_
  >> α:Env -> Term(α) -> AtomicMType(α) -> U
  Extraction:
  λ α t A. <mtype(α,t),val(α,t)> ∈{α} A

* DEF term_in_mtype
  <t:Term> ∈{<a:Env>} <A:AtomicMType> == term_in_mtype(<a>,<t>,<A>)

* THM eq_terms_in_mtype_
  >> α:Env -> Term(α) -> Term(α) -> AtomicMType(α) -> U
  Extraction:
  λα. λ x y A. x ∈{α} A & y ∈{α} A & val(α,x) = val(α,y) in val(α,A)

* DEF eq_terms_in_mtype
  <t:Term> = <tt:Term> ∈{<a:Env>} <A:AtomicMType> ==
  eq_terms_in_mtype(<a>)(<t>,<tt>,<A>)

* ML FixedPointInd
  let FixedPointInd p =
  let n = number_of_hyps p + 1 in
```

```
( InstantiateLemma 'fake_fixed_point_' [concl p] THEN
  OnLastHyp λi. Try (Refine (hyp i)) THEN
  (\pp. if is_membership_goal pp then (Id...) pp
        if concl pp = concl p then Thin n pp
        else (I THENW Thin n ...∈) pp
  )
) p
;;
```

```
* THM terms_are_eq_in_mtype
  >> ∀α:Env. ∀t,u:Term(α). ∀A:AtomicMType(α) where wf@(α,t=u in A).
        ?↓(t = u ∈{α} A)

* THM terms_are_eq_in_mtype_2
  >> ∀α:Env. ∀t,u:Term(α). ∀A:AtomicMType(α) where t ∈{α} A & u ∈{α} A.
        ?↓(t = u ∈{α} A)

* ML Assume_additions
  add_to_Assume 'terms_are_eq_in_mtype_2';;
  add_to_Assume 'term_is_eq' ;;
  add_to_Assume 'Atom_eq_decidable' ;;

* THM term_in_mtype_char
  >> ∀α:Env. ∀A:AtomicMType(α). ∀t:Term(α). t ∈{α} A => val(α,t) in |val(α,A)|


* THM terms_eq_in_mtype_sym
  >> ∀α:Env. ∀A:AtomicMType(α). ∀t1,t2:Term(α).
       ↓(t1 = t2 ∈{α} A => t2 = t1 ∈{α} A)

* THM terms_eq_in_mtype_sym_2
  >> ∀α:Env. ∀A:AtomicMType(α). ∀t1,t2:Term(α).
       ↓(t1 = t2 ∈{α} A) => ↓(t2 = t1 ∈{α} A)

* THM terms_eq_in_mtype_trans
  >> ∀α:Env. ∀A:AtomicMType(α). ∀t1,t2,t3:Term(α).
       ↓(t1 = t2 ∈{α} A => t2 = t3 ∈{α} A => t1 = t3 ∈{α} A)

* THM terms_eq_in_mtype_trans_2
  >> ∀α:Env. ∀A:AtomicMType(α). ∀t1,t2,t3:Term(α).
       ↓(t1 = t2 ∈{α} A) => ↓(t2 = t3 ∈{α} A) => ↓(t1 = t3 ∈{α} A)

* THM equality_
  >> α:Env -> Complete(α)
  Extraction:
  . . .
```

Monoids and Meta-Monoids

```
* DEF triv_mtype_constant
  <A:Atom> == <nil,<A>>

* DEF mtype_constant
  <A:Atom>-\><B:Atom> == <[<A>],<B>>

* DEF mtype_constant_2
  <A:Atom>#<B:Atom>-\><C:Atom> == <[<A>;<B>],<C>>

* THM m_bin_op_
  >> Env -> Atom -> Atom -> U
  Extraction:
  λ α A f. type_atom(α.1,A) & fun_atom(α,f) & mtype(α,f) = A#A->A in MType(α)


* DEF m_bin_op
  <f:Atom> bin op over <A:Atom>{<a:Env>} == m_bin_op(<a>,<A>,<f>)

* THM m_member_
  >> Env -> Atom -> Atom -> U
  Extraction:
  λ α A e. type_atom(α.1,A) & fun_atom(α,e) & mtype(α,e) = A in MType(α)

* DEF m_member
  <e:Atom> ∈{<a:Env>} <A:Atom> == m_member(<a>,<A>,<e>)

* THM m_bin_op_ap_
  >> α:Env -> A:Atom -> f: {f:Atom|f bin op over A{α}} ->
     ({t:Term(α)|t ∈{α} A} -> {t:Term(α)|t ∈{α} A} -> |val(α,A)|)
  Extraction:
  λ α A f. λ x y. val(α,f)(val(α,x),val(α,y))

* DEF m_bin_op_ap
  ap{<a:Env>,<A:Atom>}(<f:Atom>) == m_bin_op_ap(<a>,<A>,<f>)

* DEF infix
  <x:*> <f:*#*→*> <y:*> == <f>(<x>,<y>)

* THM monoid_
  >> S:SET -> |S| -> |S#S->S| -> U
```

```
  Extraction:
  λ S e o.
  ∀x,y,z:|S|. x o (y o z) = (x o y) o z in S & ∀x:|S|.
     x o e = x in S & e o x = x in S

* DEF monoid
  monoid(<S:S:SET>,<e:|S|>,<o:|S#S→S|>) == monoid(<S>,<e>,<o>)

* THM com_monoid_
  >> S:SET -> |S| -> |S#S->S| -> U
  Extraction:
  λ S e o. monoid(S,e,o) & ∀x,y:|S|. x o y = y o x in S

* DEF com_monoid
  com_monoid(<S:S:SET>,<e:|S|>,<o:|S#S→S|>) == com_monoid(<S>,<e>,<o>)

* THM m_bin_op_type
  >> ∀α:Env. ∀A,f:Atom. f bin op over A{α} =>
       val(α,f) in |val(α,A)#val(α,A)->val(α,A)|

* DEF minfix
  <x:Term> <f:MFun> <y:Term> == <f>([<x>;<y>])

* THM m_bin_op_ap_wf
  >> ∀α:Env. ∀A,o:Atom. o bin op over A{α} =>
       ∀x,y:Term0. wf@(α,x) & wf@(α,y) & x ∈{α} A & y ∈{α} A
            => wf@(α,x o y)

* THM m_member_type
  >> ∀α:Env. ∀A,e:Atom. e ∈{α} A =>
       e in Term(α) & mtype(α,e) = A in Atom
       & e ∈{α} A & val(α,e) in |val(α,A)| & wf@(α,e)

* THM m_monoid_
  >> Env -> Atom -> Atom -> Atom -> U
  Extraction:
  λα. λ A e o. e ∈{α} A & o bin op over A{α}
             & monoid(val(α,A), val(α,e), val(α,o))

* DEF m_monoid
  monoid(<a:Env>,<A:A:Atom>,<e:e:Atom>,<o:o:Atom>) == m_monoid(<a>,<A>,<e>,<o>)


* THM m_bin_op_ap_type
  >> ∀α:Env. ∀A,o:Atom. o bin op over A{α} =>
       ∀x,y:Term(α). x ∈{α} A & y ∈{α} A
```

```
          => x o y in Term(α) & mtype(α,x o y) = A in Atom
             & x o y ∈{α} A & val(α,x o y) in |val(α,A)|
             & val(α,o)(val(α,x),val(α,y)) in |val(α,A)|
      & ∀u,v:|val(α,A)|. val(α,o)(u,v) in |val(α,A)|

* THM m_bin_op_ap_eq
  >> ∀α:Env. ∀A,o:Atom. o bin op over A{α} =>
        ∀x,y:Term(α). x ∈{α} A & y ∈{α} A
        => val(α,x o y) = val(α,o)(val(α,x),val(α,y)) in |val(α,A)|
           & ∀u,v:|val(α,A)|. val(α,x)=u in |val(α,A)| =>
                val(α,y)=v in |val(α,A)|
                => val(α,x o y) = val(α,o)(u,v) in |val(α,A)|

* THM m_member_eq
  >> ∀α:Env. ∀A,e:Atom. e ∈{α} A => val(α,e) = val(α,e) in |val(α,A)|

* ML MBinOpApMember
  let match_m_bin_op_to_hyp H =
    match 'f bin op over A{α}' (type_of_declaration H) ``f A α`` ;;

  let get_A_lists p =
    collect ((\t. [t]) o C lookup `A` o match_m_bin_op_to_hyp) (hyps p) ;;

  let MBinOpApMemberI p =
  ( Lemma `m_bin_op_ap_type`
    ORELSE First (map (LemmaUsing `m_bin_op_ap_type`) (get_A_lists p))
    ORELSE First (map (LemmaUsing `m_bin_op_ap_wf`) (get_A_lists p))
    THEN Try Trivial THEN Try CanonicalI THEN Try Trivial
  ) p
  ;;

  let MBinOpApMember =
    Try Trivial THEN Try ReduceEquandicity THEN Repeat MBinOpApMemberI ;;

* ML MMemberMember
  let match_m_member_to_hyp H =
    match 'e ∈{α} A' (type_of_declaration H) ``e A α`` ;;

  let get_A_lists_2 p =
    collect ((\t. [t]) o C lookup `A` o match_m_member_to_hyp) (hyps p) ;;

  let MMemberMemberI p =
  ( Lemma `m_member_type`
    ORELSE First (map (LemmaUsing `m_member_type`) (get_A_lists_2 p))
    THEN Try Trivial THEN Try CanonicalI THEN Try Trivial
  ) p
```

```
  ;;

  let MMemberMember =
    Try Trivial THEN Try ReduceEquandicity THEN Repeat MMemberMemberI ;;

* THM m_monoid_char
  >> ∀α:Env. ∀A,e,o:Atom. monoid(α,A,e,o)
      => ∀x,y,z:Term(α). x ∈{α} A & y ∈{α} A & z ∈{α} A  =>
         x o (y o z) = (x o y) o z  ∈{α}  A
         & ∀x:Term(α). x ∈{α} A =>  x o e = x  ∈{α}  A  &  e o x = x  ∈{α}
A

* THM m_com_monoid_
  >> Env -> Atom -> Atom -> Atom -> U
  Extraction:
  λα. λ A e o. e ∈{α} A & o bin op over A{α}
              & com_monoid(val(α,A), val(α,e), val(α,o))

* DEF m_com_monoid
  com_monoid(<a:Env>,<A:A:Atom>,<e:e:Atom>,<o:o:Atom>) ==
  m_com_monoid(<a>,<A>,<e>,<o>)
```

Lists Over Meta-Monoids

```
* THM all_wf_and_members_
  >> α:Env -> AtomicMType(α) -> Term0 list -> U
  Extraction:
  λ α A l. ∀∈ l : Term0 list. λt. wf@(α,t) & t ∈{α} A

* DEF all_wf_and_members
  ∀t∈<l:Term0 list>. wf(t) & t ∈{<a:Env>} <A:AtomicMType> ==
  all_wf_and_members(<a>,<A>,<l>)

* ML add_two_membership_hacks
  add_to_member_i `AtomicMType when type of m_bin_op, etc`
  (\p. if `AtomicMType` = head_of_application (eq_type (concl p)) then
         (let i = find_hyp (\x,A. member (head_of_application A)
                              [`m_bin_op`; `m_member`;
                               `m_monoid`; `m_com_monoid`]) p in
          Complete (UnfoldsInHyp
                      ``m_com_monoid m_monoid m_member m_bin_op`` i
                    THEN (Id ...)) p)
       else fail
```

```
  )
  ;;

  add_to_member_i 'all_elements over Term0 membership' Fail
  %(\p. let [t],T = destruct_equal (concl p) in
       if 'all_elements' = head_of_application t then
         (if second (decompose_ap t) = 'Term0'
          then (ComputeEquands THEN Try MemberI) p else fail)
       else fail
  )%
  ;;

* THM implode_m_monoid_term_
  >> Object
  Extraction:
  λ e o l. [ nil → e; h.t,v → h o v; @ l]

* DEF implode_m_monoid_term
  implode <l:Term list> using <e:e:Atom>,<o:o:Atom> ==
  implode_m_monoid_term(<e>,<o>,<l>)

* THM implode_m_monoid_term_lemma
  >> ∀α:Env. ∀e,o,A:Atom. ∀l:Term0 list. e ∈{α} A => o bin op over A{α}
       => ∀t∈l. wf(t) & t ∈{α} A
       =>  wf@(α,(implode l using e,o)) & (implode l using e,o) ∈{α} A

* THM implode_m_monoid_term__
  >> ∀α:Env. ∀e,o,A:Atom. ∀l:Term0 list.
      e ∈{α} A => o bin op over A{α} => ∀t∈l. wf(t) & t ∈{α} A
      => (implode l using e,o) in Term(α)

* THM eq_m_monoid_lists_
  >> Env -> Atom -> Atom -> Atom -> Term0 list -> Term0 list -> U
  Extraction:
  λα. λ A e o. λ l1 l2.
  o bin op over A{α} & e ∈{α} A &
  ∀t∈l1. wf(t) & t ∈{α} A & ∀t∈l2. wf(t) & t ∈{α} A &
  (implode l1 using e,o) = (implode l2 using e,o) ∈{α} A

* DEF eq_m_monoid_lists
  <l1:Term0 list>=<l2:Term0 list>
  (mod <e:e:Atom>,<o:o:Atom>) ∈{<a:Env>} <A:A:Atom> list ==
  eq_m_monoid_lists(<a>)(<A>)(<e>)(<o>)(<l1>)(<l2>)

* THM term_is_constant
  >> ∀t:Term0. ∀e:Atom. ?(t=e in Term0)
```

```
* THM m_monoid_term_elim
  >> ∀e,o:Atom. ∀P:Term0->U.
      P(e) => (∀u,v:Term0. P(u o v)) => (∀t:Term0. P(t)) => ∀t:Term0. P(t)

* THM eq_terms_in_mtype_refl
  >> ∀α:Env. ∀A:AtomicMType(α). ∀t:Term(α). ↓(t ∈{α} A) => ↓(t = t ∈{α} A)


* THM m_bin_op_ap_subterms
  >> ∀α:Env. ∀A,o:Atom. o bin op over A{α} => ∀u,v:Term0. wf@(α,u o v) =>
      wf@(α,u) & wf@(α,v) & u ∈{α} A & v ∈{α} A

* THM m_bin_op_fnlty
  >> ∀α:Env. ∀A,o:Atom. ∀u,v,x,y:Term0.
       o bin op over A{α} => ∀t∈(u.v.x.y.nil). wf(t) & t ∈{α} A
       => ↓(u = x ∈{α} A) => ↓(v = y ∈{α} A) => ↓(u o v = x o y ∈{α} A)

* THM eq_lemma_for_explode_m_monoid_term
  >> ∀α:Env. ∀A,e,o:Atom. ∀u,v:Term0. ∀x,y,z:Term0.
       monoid(α,A,e,o) => (∀t∈(u.v.x.y.z.nil). wf(t) & t ∈{α} A)
       =>  v o x = y ∈{α} A  =>  u o y = z ∈{α} A
       => ↓( (u o v) o x = z ∈{α} A)

* THM explode_m_monoid_term
  >> ∀A,e,o:Atom. ∀t:Term0. ∃l:Term0 list where
       ∀α:Env. monoid(α,A,e,o) => wf@(α,t) => t ∈{α} A
       => ↓([t]=l (mod e,o) ∈{α} A list)

* THM m_com_monoid_char
  >> ∀α:Env. ∀A,e,o:Atom. com_monoid(α,A,e,o)
      => monoid(α,A,e,o)
         & ∀x,y:Term(α). x ∈{α} A & y ∈{α} A => x o y = y o x  ∈{α}  A

* THM eq_m_monoid_lists_refl
  >> ∀α:Env. ∀A,e,o:Atom. monoid(α,A,e,o) => ∀l:Term0 list.
       ∀t∈l. wf(t) & t ∈{α} A => ↓(l=l (mod e,o) ∈{α} A list)

* THM eq_lemma_for_insert_in_m_monoid_term_list
  >> ∀α:Env. ∀A,e,o:Atom. ∀u,v,x,y:Term0.
       com_monoid(α,A,e,o) => (∀t∈(u.v.x.y.nil). wf(t) & t ∈{α} A)
       =>  u o x = y ∈{α} A  => ↓(u o (v o x) = v o y ∈{α} A)

* THM insert_in_m_monoid_term_list
  >> ∀A,e,o:Atom. ∀ord:Term0->Term0->Bool. ∀t:Term0. ∀l1:Term0 list.
     ∃l2:Term0 list where
```

```
      ∀α:Env. com_monoid(α,A,e,o) => wf@(α,t) => t ∈{α} A
              => ∀t∈l1. wf(t) & t ∈{α} A
              => ↓(t.l1=l2 (mod e,o) ∈{α} A list)

* THM sort_m_com_monoid_term_list
  >> ∀A,e,o:Atom. ∀ord:Term0->Term0->Bool. ∀l1:Term0 list.
     ∃l2:Term0 list where
      ∀α:Env. com_monoid(α,A,e,o) => ∀t∈l1. wf(t) & t ∈{α} A
              => ↓(l1=l2 (mod e,o) ∈{α} A list)

* THM is_m_monoid_term
  >> ∀A,e,o:Atom. ∀t:Term0.
      ?↓(∀α:Env. monoid(α,A,e,o) => wf@(α,t) => mtype(α,t)=A in Atom)

* THM typecheck
  >> ∀A:Atom. ∀α:Env. ∀t:Term0.
      ?↓(∀α2:Env. α⊂α2 => wf@(α2,t) => mtype(α2,t) = A in Atom)

* THM m_monoid_mono
  >> ∀α1⊂α2. ∀A,e,o:Atom. monoid(α1,A,e,o) => monoid(α2,A,e,o)

* THM m_com_monoid_mono
  >> ∀α1⊂α2. ∀A,e,o:Atom. com_monoid(α1,A,e,o) => com_monoid(α2,A,e,o)

* THM eq_term_val_if_eq_terms_in_mtype
  >> ∀α:Env. ∀x,y:Term(α). ∀A:AtomicMType(α).
      mtype(α,x) = A in Atom => mtype(α,y) = A in Atom
      => x = y ∈{α} A => val(x) ={α} val(y)

* THM finish_m_monoid_implosion
  >> ∀A,e,o:Atom. ∀α1:Env. ∀l:Term0 list. ?∃t:Term0 where
      ∀α2:Env. monoid(α2,A,e,o) => α1⊂α2 => ∀t∈l. wf(t) & t ∈{α2} A =>
          wf@(α2,t) & t ∈{α2} A & ↓(val(implode l using e,o) ={α2} val(t))
```

A Normalizer for Commutative Monoids

```
* THM band_
  >> Bool->Bool->Bool
  Extraction:
  λ b1 b2. d(b1;u.b2;u.false)

* DEF band
```

```
<P:PBool> & <Q:PBool> == band(<P>)(<Q>)

* THM less_wrt_alist_
  >> A:Type -> Atom#A list -> Atom -> Atom -> Bool
  Extraction:
  λA. λ l x y.
  if x=y then false
  else [ nil → false;
         h.t,v → if h.1=x then true else if h.1=y then false else v; @ l]

* DEF less_wrt_alist
  <x:Atom> \<{<l:alist>: Atom#<A:Type> list} <y:Atom> ==
  less_wrt_alist(<A>)(<l>,<x>,<y>)

* THM type_atom_less_
  >> α:Env -> Atom -> Atom -> Bool
  Extraction:
  λ α x y. x <{α.1: Atom#TEnvVal list} y

* THM fun_atom_less_
  >> α:Env -> Atom -> Atom -> Bool
  Extraction:
  λ α x y. x <{α.2: Atom#FEnvVal(α.1) list} y

* THM term_less_
  >> ∀α:Env. ∀t1,t2:Term0. Bool
  Extraction:
  . . .

* THM norm_m_com_monoid_term_lemma
  >> ∀α1:Env. ∀A,e,o:Atom. ∀t1:Term0. ?∃t2:Term0 where
       ∀α2:Env. α1⊂α2 => com_monoid(α2,A,e,o) =>
         wf@(α2,t1) => ↓(wf@(α2,t2) & val(t1) ={α2} val(t2))

* THM norm_com_monoid_term_
  >> Env -> Atom -> Atom -> Atom -> Term0 -> ?Term0
  Extraction:
  norm_m_com_monoid_term_lemma

* DEF norm_com_monoid_term
  norm{<a:Env>,<A:A>,<e:e>,<o:o>} == norm_com_monoid_term(<a>,<A>,<e>,<o>)

* THM norm_com_monoid_term__
  >> ∀α1:Env. ∀A,e,o:Atom. com_monoid(α1,A,e,o) =>
       ∀α2:Env. α1⊂α2 => norm{α2,A,e,o} in Rewrite(α2)
```

C.1.17 Examples

The Rational Numbers

```
* THM Int_cancellation
  >> ∀x,y,z:Int. ¬(z = 0) => x*z = y*z => x = y

* THM Int_cancellation_2
  >> ∀w,x,y,z:Int. ¬(y = 0) => ¬(z = 0) => (w*y)*z = (x*y)*z => w = x

* THM Q_
  >> Set(1)
  Extraction:
  <Int#N+, λ x,y. x.1*y.2 = y.1*x.2 in Int>

* DEF Q
  Q == Q

* THM Q_eq_triv
  >> triv_eq(Q)

* DEF Q_0_m
  0 == Q_0

* DEF Q_1_m
  1 == Q_1

* THM rat_
  >> Int -> N+ -> |Q|
  Extraction:
  λ x y. <x,y>

* DEF rat
  <x:Int>/<y:N+> == rat(<x>,<y>)

* THM rat_of_int_
  >> Int -> |Q|
  Extraction:
  λn. <n,1>

* DEF rat_of_int
  <n:int> == rat_of_int(<n>)
```

```
* ML ExpandQ
  let ExpandQ i p =
  ( if `NIL` = id_of_hyp i p then failwith `ExpandQ` ;
    let ExpandMember i p =
      if h i p = '|Q|'
      then (E i THEN Thin i THEN OnLastHyp E THEN ReduceConcl) p
      else failwith `ExpandQ` in
    if h i p = '|Q#Q|' then
      E i THEN Thin i THEN OnLastHyp λi. ExpandMember i
      THEN ExpandMember (i-1)
    else ExpandMember i
  ) p
  ;;

* THM Q_plus_
  >> |Q#Q -> Q|
  Extraction:
  λ x,y. < x.1*y.2 + y.1*x.2, x.2*y.2 >

* DEF Q_plus
  <x:Q>+<y:Q> == Q_plus(<x>,<y>)

* DEF Q_plus_m
  <x:mQ>+<y:mQ> == <x> Q_plus <y>

* THM Q_plus_assoc
  >> ∀x,y,z:|Q|. x+(y+z) = (x+y)+z in Q

* THM Q_plus_com
  >> ∀x,y:|Q|. x+y = y+x in Q

* THM Q_neg_
  >> |Q->Q|
  Extraction:
  λx. <-x.1,x.2>

* DEF Q_neg
  -<x:Q> == Q_neg(<x>)

* DEF Q_neg_m
  -(<x:mQ>) == Q_neg([<x>])

* THM Q_neg_over_plus
  >> ∀x,y:|Q|. -(x+y) = -x+-y in Q
```

```
* THM Q_mult_
  >> |Q#Q->Q|
  Extraction:
  λ x,y. <x.1*y.1, x.2*y.2>

* DEF Q_mult
  <x:Q>*<y:Q> == Q_mult(<x>,<y>)

* DEF Q_mult_m
  <x:mQ>*<y:mQ> == <x> Q_mult <y>

* THM Q_mult_assoc
  >> ∀x,y,z:|Q|. x*(y*z) = (x*y)*z in Q

* THM Q_mult_com
  >> ∀x,y:|Q|. x*y = y*x in Q

* THM a_Q_
  >> Env
  Extraction:
  let γ = [<Q,Q,s(Q_eq_triv)>] in
  <γ,
   make_simple_fenv_unit(γ, Q_0, Q, 0).
   make_simple_fenv_unit(γ, Q_1, Q, 1).
   make_simple_fenv_unit(γ, Q_plus, Q#Q->Q, Q_plus).
   make_simple_fenv_unit(γ, Q_mult, Q#Q->Q, Q_mult).
   make_simple_fenv_unit(γ, Q_neg, Q->Q, Q_neg).
   nil>

* DEF a_Q
  α_Q == a_Q
```

Examples

```
* THM Q_additive_m_com_monoid
  >> com_monoid(α_Q, Q, Q_0, Q_plus)

* THM Q_multiplicative_m_com_monoid
  >> com_monoid(α_Q, Q, Q_1, Q_mult)

* THM Q_mult_over_plus
  >> ∀x,y,z:|Q|. x*(y+z) = x*y+x*z in Q  &  (x+y)*z = x*z+y*z in Q
```

```
* DEF lw
  (<T:tactic>...) ==
  ( (\p. (<T>) p) THEN
    (\p. if is_term_val (destruct_squash (concl p)) ? false then
           Try CompleteWeakAutotactic p
         else WeakAutotactic p
    )
  )

* THM equality_test
  >> ∀x,y,z:|Q|. z=x in Q => x=y in Q => y=z in Q

* THM norm_wrt_Q_neg
  >> Rewrite(α_Q)

* THM test_norm_wrt_Q_neg
  >> ∀w,x,y,z:|Q|. -(w+x+y*z+z) = -w+-x+-y*z+-z in Q

* THM norm_Q_term_wrt_plus_
  >> Object
  Extraction:
  λα. Topmost(norm{α,Q,Q_0,Q_plus})

* DEF norm_Q_term_wrt_plus
  norm_wrt_Q_plus(<a:Env>) == norm_Q_term_wrt_plus(<a>)

* THM norm_Q_term_wrt_plus__
  >> ∀α:Env. α_Q⊂α => norm_wrt_Q_plus(α) in Rewrite(α)

* THM test_norm_Q_term_wrt_plus
  >> ∀w,x,y,z:|Q|. x+(y+(w+z)) = ((y+z)+y)+z in Q  => (y+(0+y))+(z+z) = 0 in
Q
                 => 0+0 = (z+x)+(w+y) in Q
```

C.2 Part of the match Proof

Below is complete listing of a subproof of the proof of the theorem `match_` that was described in Section 5.7.

The listing was produced by an ML function that prints out the nodes of its proof argument in the order visited by a depth first traversal of the proof tree. Indenting is used to indicate depth, and redundant hypotheses are not displayed. Vertical lines (or an approximation thereof) are printed to help the reader keep track of levels, and tick marks (hyphens) are put on the lines to indicate the beginning of a conclusion of a proof node. To the right of any turnstile (>>) appears the conclusion of a node of the proof tree. Directly above is a numbered list of the hypotheses of the proof node that are different from the hypotheses of the node's parent. The parent is found by following upward the first vertical line to the left of the turnstile. Immediately after the conclusion of the node is the refinement rule, and below that, indented once from the turnstile, are the children of the node.

The listing below was manually beautified by adjusting some whitespace and by restoring a few definitions that the system had failed to maintain.

```
1. vars:Atom list
2. t1:Term0
3. f:Atom
4. l:Term0 list
5. ∀∈ l : Term0 list.
     λx3. ∀t2:Term0. ?{ s:(Atom#Term0) list | s{vars} complete on x3
                                             & s(x3)=t2 in Term0 }
6. t2:Term0
>> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
                            & s(f(l))=t2 in Term0 }
|  BY (Decide 'null(l)' ...)
|  7. null(l)
|- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
|  |                           & s(f(l))=t2 in Term0 }
|  |  BY (Decide '∃∈ vars : Atom list. λx. f=x in Atom' ...)
|  |  8. ∃∈ vars : Atom list. λx. f=x in Atom
|  |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
|  |  |                           & s(f(l))=t2 in Term0 }
|  |  |  BY (ILeft THENW ITerm '[<f,t2>]' ...)
|  |  |- >> ([<f,t2>]{vars} complete on f(l))
|  |  |  |  BY (EvalConclExcept (defs 'some_element')
|  |  |  |      THEN ReduceConcl...)
|  |  |  |  9. a:Atom
|  |  |  |  10. ∃∈ vars : Atom list. λx. a=x in Atom
|  |  |  |  11. a∈f(l)
|  |  |  |- >> ∃∈ [<f,t2>] : Atom#Term0 list. λx. a=x.1 in Atom
|  |  |  |  |  BY (EvalConcl THEN ILeft ...)
|  |  |  |  |- >> a=f in Atom
|  |  |  |  |  |  BY BringHyps [7;11] THEN E 4
|  |  |  |  |  |- >> null(nil) => a∈f(nil) => a=f in Atom
|  |  |  |  |  |  |  BY (EvalConcl ...)
|  |  |  |  |  |  |  12. True
|  |  |  |  |  |  |  13. (True & a=f in Atom) ∨ False
```

```
| | | | | | |- >> a=f in Atom
| | | | | | | | BY (E 13 ...)
| | | | | | 12. h4:Term0
| | | | | | 13. t5:Term0 list
| | | | | | 14. null(t5) => a∈f(t5) => a=f in Atom
| | | | | |- >> null(h4.t5) => a∈f(h4.t5) => a=f in Atom
| | | | | | | BY (UnrollDefsInConcl ``null`` ...)
| | | 9. [<f,t2>]{vars} complete on f(l)
| | |- >> [<f,t2>](f(l))=t2 in Term0
| | | | BY BringHyp 7 THEN E 4
| | | |- >> null(nil) => ([<f,t2>](f(nil)) = t2 in Term0)
| | | | | BY (EvalConclExcept (defs `Term0`) ...)
| | | | | 10. True
| | | | |- >> decide(atom_eq(f;f;inl(t2);inr(axiom));
| | | | | |        t2.t2;_.f(nil))
| | | | | | = t2 in Term0
| | | | | | BY (SubstFor 'if f=f then inl(t2) else inr(axiom)
| | | | | |                = inl(t2) in ?Term0' ...)
| | | | | |- >> atom_eq(f;f;inl(t2);inr(axiom))
| | | | | | | = inl(t2) in Term0|True
| | | | | | | BY (ReduceDecisionTerm 1 true ...)
| | | | 10. h4:Term0
| | | | 11. t5:Term0 list
| | | | 12. null(t5) => ([<f,t2>](f(t5)) = t2 in Term0)
| | | |- >> null(h4.t5) => ([<f,t2>](f(h4.t5))=t2 in Term0)
| | | | | BY (UnrollDefsInConcl ``null`` ...)
| | 8. ¬(∃∈ vars : Atom list. λx. f=x in Atom)
| |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
| | |                           & s(f(l))=t2 in Term0 }
| | | BY (OnVar `t2` Term0Unroll ...)
| | | 6. null(l)
| | | 7. ¬(∃∈ vars : Atom list. λx. f=x in Atom)
| | | 8. f1:Atom
| | | 9. l1:Term0 list
| | |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
| | | |                           & s(f(l))=inl(<f1,l1>) in Term0 }
| | | | BY (Decide 'f=f1 in Atom & null(l1)' ...)
| | | | 10. f=f1 in Atom
| | | | 11. null(l1)
| | | |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
| | | |                             & s(f(l))=inl(<f1,l1>) in Term0 }
| | | | | BY (ILeft THENW ITerm 'nil' ...)
| | | | |- >> (nil{vars} complete on f(l))
```

```
|  |  |  |  |  |  BY (EvalConcl ...)
|  |  |  |  |  |  12. a:Atom
|  |  |  |  |  |  13. ∃∈ vars : Atom list. λx. a=x in Atom
|  |  |  |  |  |  14. a∈f(l)
|  |  |  |  |  |- >> ∃∈ nil : Atom#Term0 list. λx. a=x.1 in Atom
|  |  |  |  |  |  |  BY (Assert 'False' THENO
|  |  |  |  |  |  |       (UnrollTermFunInHyp `occurs` 14
|  |  |  |  |  |  |        THEN RepeatOrE 14) ...)
|  |  |  |  |  |  |  14. null(l)
|  |  |  |  |  |  |  15. a=f in Atom
|  |  |  |  |  |  |- >> False
|  |  |  |  |  |  |  |  BY (SubstForInHyp (h 15 p) 13 ...)
|  |  |  |  |  |  |  |     THEN (E 7 ...)
|  |  |  |  |  |  |  14. ∃∈ l : Atom list. occurs(a)
|  |  |  |  |  |  |- >> False
|  |  |  |  |  |  |  |  BY BringHyps [6;14] THEN E 4
|  |  |  |  |  |  |  |- >> null(nil) => ¬(∃∈ nil : Atom list. occurs(a))
|  |  |  |  |  |  |  |  |  BY (EvalConcl ...)
|  |  |  |  |  |  |  |  15. h3:Term0
|  |  |  |  |  |  |  |  16. t4:Term0 list
|  |  |  |  |  |  |  |  17. null(t4) => ¬(∃∈ t4 : Atom list. occurs(a))
|  |  |  |  |  |  |  |- >> null(h3.t4) =>
|  |  |  |  |  |  |  |  |  ¬(∃∈ h3.t4 : Atom list. occurs(a))
|  |  |  |  |  |  |  |  |  BY (UnrollDefsInConcl ``null`` ...)
|  |  |  |  |  12. (nil{vars} complete on f(l))
|  |  |  |  |- >> nil(f(l))=inl(<f1,l1>) in Term0
|  |  |  |  |  |  BY BringHyp 6 THEN E 4
|  |  |  |  |  |- >> null(nil) => (nil(f(nil)) = inl(<f1,l1>) in Term0)
|  |  |  |  |  |  |  BY (EvalConclExcept (defs `Term0`) ...)
|  |  |  |  |  |  |  13. True
|  |  |  |  |  |  |- >> inl(<f,nil>)=inl(<f1,l1>) in Term0
|  |  |  |  |  |  |  |  BY RepeatFor 3 (MemberI ...)
|  |  |  |  |  |  |  |- >> nil=l1 in Term0 list
|  |  |  |  |  |  |  |  |  BY (BringHyp 11 THEN E 9 THEN
|  |  |  |  |  |  |  |  |       EvalConclExcept (defs `Term0`) ...)
|  |  |  |  |  |  13. h3:Term0
|  |  |  |  |  |  14. t4:Term0 list
|  |  |  |  |  |  15. null(t4) => (nil(f(t4)) = inl(<f1,l1>) in Term0)
|  |  |  |  |  |- >> null(h3.t4) => (nil(f(h3.t4)) = inl(<f1,l1>)
|  |  |  |  |  |  |                   in Term0)
|  |  |  |  |  |  |  BY (UnrollDefsInConcl ``null`` ...)
|  |  |  |  10. ¬((f=f1 in Atom) & null(l1))
|  |  |  |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
```

```
| | | | |                         & s(f(l))=inl(<f1,l1>) in Term0 }
| | | | | BY (IRight ...)
| | | 6. null(l)
| | | 7. ¬(∃∈ vars : Atom list. λx.(f=x in Atom))
| | | 8. u1:Term0
| | | 9. v1:Term0
| | | 10. A1:Atom
| | |- >> ?{ s:(Atom#Term0) list| s{vars} complete on f(l)
| | | |                       & s(f(l)) = inr(inl(<u1,<v1,A1>>))
| | | |                         in Term0 }
| | | | BY (IRight ...)
| | | 6. null(l)
| | | 7. ¬(∃∈ vars : Atom list. λx.(f=x in Atom))
| | | 8. i1:Atom
| | | 9. u1:Term0
| | | 10. v1:Term0
| | |- >> ?{ s:(Atom#Term0) list |
| | | |     s{vars} complete on f(l)
| | | |     & s(f(l))=inr(inr(inl(<i1,<u1,v1>>))) in Term0 }
| | | | BY (IRight ...)
| | | 6. null(l)
| | | 7. ¬(∃∈ vars : Atom list. λx.(f=x in Atom))
| | | 8. b7:Int
| | | 9. n1:Int
| | |- >> { s:(Atom#Term0) list |
| | | |     s{vars} complete on f(l)
| | | |     & s(f(l)) = inr(inr(inr(n1))) in Term0 }
| | | | BY (IRight ...)
| 7. ¬(null(l))
|- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
| |                          & s(f(l))=t2 in Term0 }
| | BY (OnVar 't2' Term0Unroll ...)
| | 6. ¬(null(l))
| | 7. f1:Atom
| | 8. l1:Term0 list
| |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
| | |                          & s(f(l))=inl(<f1,l1>) in Term0 }
| | | BY (Decide 'f=f1 in Atom' ...)
| | | 9. f=f1 in Atom
| | |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
| | | |                          & s(f(l))=inl(<f1,l1>) in Term0 }
| | | | BY (CaseLemma 'list_subst_lemma' ['vars'; 'l'; 'l1'] ...)
| | | | 10. { s:(Atom#Term0) list |
```

```
| | | |          ∀∈ l : Term0 list. λt. s{vars} complete on t
| | | |          & (map subst(s) on l to Term0 list)
| | | |            = l1 in Term0 list }
| | | |- >> ?{ s:(Atom#Term0) list |
| | | | |        s{vars} complete on f(l)
| | | | |        & s(f(l))=inl(<f1,l1>) in Term0 }
| | | | | BY (Thin 5 THEN Unsetify THEN ILeft THENW ITerm 's' ...)
| | | | | 5. ¬(null(l))
| | | | | 6. f1:Atom
| | | | | 7. l1:Term0 list
| | | | | 8. f=f1 in Atom
| | | | | 9. s:(Atom#Term0) list
| | | | | 10. ∀∈ l : Term0 list. λt. s{vars} complete on t
| | | | | 11. (map subst(s) on l to Term0 list) = l1 in Term0 list
| | | | |- >> s{vars} complete on f(l)
| | | | | | BY (Lemma 'complete_subst_on_fun_ap' ...)
| | | | | 5. ¬(null(l))
| | | | | 6. f1:Atom
| | | | | 7. l1:Term0 list
| | | | | 8. f=f1 in Atom
| | | | | 9. s:(Atom#Term0) list
| | | | | 10. ∀∈ l : Term0 list. λt.(s{vars} complete on t)
| | | | | 11. (map subst(s) on l to Term0 list)=l1 in Term0 list
| | | | | 12. s{vars} complete on f(l)
| | | | |- >> s(f(l))=inl(<f1,l1>) in Term0
| | | | | | BY (SplitEq 'f((map subst(s) on l to Term0 list))'
| | | | | |     THEN Try (Lemma 'subst_on_fun_ap') ...)
| | | | | |- >> f((map subst(s) on l to Term0 list))
| | | | | | |  = inl(<f1,l1>) in Term0
| | | | | | | BY (ComputeEquands THEN MemberI ...)
| | | | | | |- >> inl(<f,(map subst(s) on l to Term0 list)>)
| | | | | | | |  = inl(<f1,l1>)
| | | | | | | |  in Atom#(Term0 list) | Term0#Term0#Atom
| | | | | | | |     | Atom#Term0#Term0 | Int
| | | | | | | | BY RepeatFor 2 (MemberI...)
| | | | 10. True
| | | |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
| | | | |                           & s(f(l))=inl(<f1,l1>) in Term0 }
| | | | | BY (IRight ...)
| | | 9. ¬(f=f1 in Atom)
| | |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
| | | |                           & s(f(l))=inl(<f1,l1>) in Term0 }
| | | | BY (IRight ...)
```

```
|   |  6. ¬(null(l))
|   |  7. u1:Term0
|   |  8. v1:Term0
|   |  9. A1:Atom
|   |- >> ?{ s:(Atom#Term0) list |
|   |  |       s{vars} complete on f(l)
|   |  |       & s(f(l)) = inr(inl(<u1,<v1,A1>>)) in Term0 }
|   |  |  BY (IRight ...)
|   |  6. ¬(null(l))
|   |  7. i1:Atom
|   |  8. u1:Term0
|   |  9. v1:Term0
|   |- >> ?{ s:(Atom#Term0) list |
|   |  |       s{vars} complete on f(l)
|   |  |       & s(f(l))=inr(inr(inl(<i1,<u1,v1>>))) in Term0 }
|   |  |  BY (IRight ...)
|   |  6. ¬(null(l))
|   |  7. b7:Int
|   |  8. n1:Int
|   |- >> ?{ s:(Atom#Term0) list | s{vars} complete on f(l)
|   |  |                           & s(f(l))=inr(inr(inr(n1))) in Term0) }
|   |  |  BY (IRight ...)
```