Aspects of the Implementation of Type Theory

Robert W Harper

Ph.D. Thesis
TR 85-675
April 1985

Department of Computer Science
Cornell University
Ithaca, New York 14853

ASPECTS OF THE IMPLEMENTATION OF 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

Robert William Harper, Jr.

June, 1985

Contents

List of Figures

Chapter 1

Introduction

This thesis is about building an automated programming logic. For our purposes an automated programming logic consists of:

- A formal system for reasoning about programs.
- A proof development environment, which includes, at least, an editor for the construction of proofs in the logic.
- Mechanized decision methods to assist in the proof development process.
- A library mechanism for managing collections of theorems.

There are numerous examples of systems which, in varying degrees, fall under this definition. Among them are the various implementations of Scott's *Logic for Computable Functions*, most of which stem from the Edinburgh system described in [Gordon, Milner, & Wadsworth 79]. The interactive implementation of PLC/V2 [Constable, Johnson, & Eichenlaub 82] by Krafft [Krafft 81] is another example. Cartwright's typed LISP verifier [Cartwright 76] and Boyer and Moore's theorem prover [Boyer & Moore 79], can also be considered to be automated programming logics in this sense.

The subject of this thesis is the PRL system. PRL stands for *program* (or *proof*) *refinement logic*, an acronym reflecting both the close relationship between programs and proofs which distinguishes the system and the style of proof construction embodied in the proof development environment. We begin with a brief overview of PRL in order to establish the context in which we will be working.

The PRL system was conceived by Joseph Bates for his Ph.D. thesis [Bates 79]. The essential idea behind the thesis is that one can construct programs by concentrating *only* on the specification, reducing program development to proof construction — to build a program satisfying a specification, one constructs a proof, in a suitable logic, that the specification can be met. This idea dates back to [Constable 71] and [Bishop 70]; Bates showed how one could build a computer system for developing correct programs interactively in this setting.

Taking the notion of stepwise refinement seriously, Bates designed the logic of PRL in such a way that proof development occurs in an orderly top–down manner, beginning with the specification and ending with trivial assertions about, say, the data domain. This notion of refinement is naturally cast in terms of trees, so in fact the user views proof construction as the process of building a proof tree whose root is the specification and whose leaves are trivial theorems (where "trivial" means whatever it is that the system can get by itself). Given a proof of a specification, the system then "compiles" the proof into executable code (without user assistance) by a process known as *extraction*.

Readers familiar with the logic of PLC/V2 may notice that the PRL approach is *dual* to that system in the sense that the key idea of the PLC/V2 is that programs can be regarded as proofs whereas PRL treats proofs as programs. This duality is explored in the pair of reports [Constable 82c,Bates & Constable 85].

Bates' ideas were brought to fruition in the λ-PRL[1] system. λ-PRL is a complete proof/program development environment comprising all of the components mentioned above. The logic is a first–order constructive logic of integers and lists. The interactive development environment centers on the proof editor, called a *refinement editor*, which provides substantial facilities for notational extension and automated assistance. There is also a library facility for managing databases of definitions, theorems, extracted programs, and decision methods. The system is described in detail in [PRL Staff 83].

[1]The λ is a version designator, as are μ and ν.

The λ-PRL system is, in the not unbiased opinion of the author, a great success. The ideas espoused by Bates in his thesis were seen to be more than ungrounded theorizing — one *could* build correct programs by refinement of its specification and, what's more, it could be fun. Numerous non-trivial program developments have been carried out with the system. For instance, in [Constable 82c] the maximum segment sum problem posed by Gries in his book [Gries 81] is solved. The automated assistance provided by the refinement editor and the notation extension facility are particularly impressive and crucial to making the system usable.

But lest the reader get the impression that the last word has been said, we discuss some of its deficiencies. The most glaring weakness, and the only one with which we shall concern ourselves, is the logic. Experience has shown [Constable & Zlatin 84,Constable 83a,Constable 80] that a much more expressive logic is desirable. For instance, it is not possible in λ-PRL to create programs which take functions as arguments for the simple reason that the data domain does not include functions. As remarked above, there are exactly two data types provided by λ-PRL, integers and lists (of integers), and there is no means of building more complex types from this basis. This limits not only the class of programs which may be written, but also prevents formalizing any advanced mathematics such as constructive analysis (as described, for example, in [Bishop 67]). Furthermore, the λ-PRL logic is first-order, and as such is incapable of reasoning about propositions. This precludes the possibility of building correct proof-constructing programs *within* the system (as described, for example, in [Constable 82a,Constable 82b]).

The remedy for this deficiency is, of course, to design a more expressive logic on which to base PRL. The choice of logic was based on many considerations, among these being the deficiencies of λ-PRL mentioned above, and on semantic and philosophical principles described in detail in [Bates & Constable 81,Constable & Zlatin 84,Constable 83a,Constable 83b,Constable 80,Constable 82a,Constable 82b]. The result is the ν-PRL logic defined in [Bates & Constable 83]. The logic is based on the *intuitionistic theory of types* defined by Per Martin-Löf in [Martin-Löf 73,Martin-Löf 82], hence the phrase "type theory." The roots of this logic can be traced back to the founders of mathematical logic, most notably Frege, Russell, and Brouwer. The reader unfamiliar with the history of mathematical logic, and especially intuitionism, is referred to [vanHeijenoort 67,Dummet 77] for introductory material and references to other sources. Its design was also influenced by the AUTOMATH project [deBruijn 80] and by Scott's early work on constructive type theory [Scott 70].

The features of the ν-PRL logic which are most relevant to the present context are:

- Rich type structure. A full complement of basic types and type constructors is available. For instance, in addition to the usual pairing and union constructors, ν-PRL has dependent function space and product constructions.

- ω-order reasoning. Arbitrarily higher-order quantification is possible. This includes the ability to define programs which take types as arguments and have types as results and the ability to quantify over propositional functions.

- Intensionality. The type structure is capable of supporting intensional analysis of types such as that in [Constable & Zlatin 84,Constable 82a].

- Extensibility. The semantic principles underlying the logic are well-developed and provide a precise framework for judging proposed extensions.

- Expressiveness. The language is sufficiently powerful as to be able to formalize nearly all of constructive mathematics (such as Bishop's constructive analysis [Bishop 67]).

These features do not come free. The additional generality of type theory over the simple first-order theory of λ-PRL introduces many complications. For instance, in type theory it is undecidable whether or not a given expression denotes a proposition, a considerable departure from ordinary logic. As a result it is incumbent on the user to *demonstrate* that the formulas used in a proof are well-formed, a rather unsavory prospect. Obviously some means of relieving this burden must be provided, and indeed that is the subject of Chapter 2.

There are a host of other complications attributable to the richness of the ν-PRL logic. In order to explain the issues with which we shall be dealing in this work, a more detailed account of proof development in PRL is necessary. The next two sections are devoted to an overview of PRL and of the ν-PRL logic.

Proof Development in PRL

Program development in PRL reduces to proof development in some formal system. This section consists of an overview of proof construction in λ-PRL in order to make the ideas of refinement and extraction more concrete. Our discussion will be as independent of the details of the logic as possible, and so will be faithful to the spirit, not to the letter, of PRL. For a complete introduction to proof construction in PRL, see [PRL Staff 83]. It is assumed that the reader has a nodding acquaintance with Gentzen's *natural deduction* formulation of predicate logic. Introductory accounts can be found in [Bell & Machover 77]; see [Prawitz 65] for more advanced material.

The logic of λ-PRL is a formalization of first–order constructive predicate calculus over the domain of natural numbers and lists of natural numbers. The rules are structured as a *sequent calculus* or *tableau system* similar to those defined by Gentzen (see [Gentzen 35]). The basic unit of inference is the *sequent* or *goal*. A goal has the form $H \gg F$ where F is some formula, callled the *conclusion* of the goal, and H is a sequence of formulas, called the *hypothesis list*. The meaning of $H \gg F$ is that F is true under the assumption that each of the formulas in H is true.

The proof rules define a set of valid inferences. Each rule has zero or more premises (each a goal) and a single conclusion. Occasionally a rule will take some term as a parameter, but most often rule names are single tokens. The rules are presented in a form suggestive of the way that they are used. For instance, the introduction rule for logical conjunction, $\wedge$, is

$$\begin{array}{l} H \gg A \wedge B \quad \text{by intro} \\ \quad H \gg A \\ \quad H \gg B \end{array}$$

The conclusion of the rule appears first, with the premises appearing indented beneath it. The premises are, for this reason, called *subgoals*.

Derivations in the logic are structured as trees, each node a goal labelled with a rule name and its parameters (if any). Each node in the tree has as children the premises of the rule labelling that node. If the rule has no premises, the node is a leaf. We shall have need to consider derivations–in–progress, and so we take an incomplete proof tree to be a proof tree in which some, perhaps none, of the nodes are unlabelled by rules. Such nodes have no children. Notice that by this definition every proof tree has the property that it is correct as far as it goes — no node can have children which are not the premises of the rule instance determined by the node's label and the contents of the node.

The process of constructing a proof tree in PRL is called *refinement*; derivations are built using the *refinement editor*, called RED. The basic idea is that at any one time attention is focused on an unlabelled node of a partial proof tree in order that a rule may be applied at that node. The process of attaching a rule and its parameters to a node and generating its children is called a *refinement step*. RED ensures that the proposed rule applies to the goal, raising an error if not. The process is initialized by presenting an unlabelled goal to RED. At each refinement step, the children generated by RED are left unlabelled, and so are potential foci of attention. As proof construction continues, more leaves are introduced by refinement steps, and the user may freely direct his attention to any of these nodes. This process continues until a complete derivation tree has been constructed.

For instance, we can build a derivation of $A, B \gg A \wedge B$ as follows. First the goal is presented to RED and attention is focused on it. We then apply the "$\wedge$ intro" rule at that node, obtaining the subgoals $A, B \gg A$ and $A, B \gg B$ as defined by the rule given above. To each of these goals we may apply the "hypothesis" rule since in each case the conclusion of the goal appears as a hypothesis. The complete proof tree is depicted in Figure 1.1.

This is an idealized description of proof development in PRL, omitting most of the details of the interaction with the system. But it serves to expose a key property that the proof rules must possess in order to be amenable to refinement — given a goal and a rule (perhaps with some parameters), the refinement editor must be able to determine whether or not the rule applies to that goal and must be able to generate the premises of the rule instance. A logic which has this property is said to have *constructible subgoals*. For the λ-PRL logic, this is relatively easy to achieve

$$\begin{array}{rl} A, B \gg A \wedge B & \text{by introduction} \\ A, B \gg A & \text{by hypothesis} \\ A, B \gg B & \text{by hypothesis} \end{array}$$

Figure 1.1: A Sample Proof Tree

(in fact, one can view Gentzen's sequent calculus as a formalization of his natural deduction system so that it has this property). But, as we shall see, achieving constructible subgoals is quite a lot more difficult in the case of the ν-PRL logic.

Another less obvious property that a PRL logic must possess is that refinement steps must be strictly local. That is, it must not be possible that a proof tree can become invalid by taking a locally valid refinment step. This is more of a restriction on the sorts of proof editing operations that can be accommodated in the PRL setting than it is on the proof rules, but since there is such a close relationship between the two it is worthwhile to make this point here. For instance, this restriction rules out the possibility of modifying interior nodes of a proof tree since any such modification will propagate throughout that subtree, thereby raising the possibility of invalidating previously valid steps. This property guarantees that derivation trees constructed by RED are always valid.

Another central feature of proof development in PRL is extraction, the process of "compiling" proofs into executable code. As one might imagine, this is not something that one would expect to be able to do with an arbitrary proof formalized in, say, set theory. But the various PRL logics are constructive logics, and as such have a computational semantics. A complete discussion of constructive semantics would take us far afield (but see, for example, [Brouwer 23,Heyting 56,Dummet 77]); what follows is a brief overview with particular emphasis on the application to PRL.

In a constructive setting the truth of a proposition is established by building an object, called a *justification*, for the proposition. The justifications themselves are built up from variables and some trivial tokens justifying axioms by constructions such as pairing and λ-abstraction. There will be a more thorough treatment of justifications in the presentation of the ν-PRL logic. Free variables in a justification refer to the hypotheses under which the justification has been constructed. These hypotheses are collected together in a goal, which is one of the major reasons why goals have the form that they do in PRL. We therefore speak of the justification of a goal, rather than a formula, in order that the bindings of the free variables are explicitly determined.

Extraction is the process of building the justification of a goal from the justifications of its premises. If there are no premises to the rule, then the justification must be either a variable or one of the trivial basic justifications. The justification of a goal is referred to as the *extracted code* or *extracted object* of that goal. Extraction is a completely automatic process which is applied to a complete (or in some cases an incomplete) proof tree. The extracted code is never displayed, but in some cases it is made available to be evaluated.

In order to support extraction the logic must have the property that justifications can be computed automatically by the extractor by an inductive process — that is, with reference only to the extracted code of the subgoals. The first prerequisite is that the logic have constructive semantics so that the whole idea of justifications make sense. The second is that the logic must be defined in such a way that extraction can take place given only a valid proof tree. The interested reader is referred to [Bates 79,Sasaki 84] for a thorough discussion of extraction and optimization of extracted code.

In our discussion of extraction we noted that in certain cases extracted objects are available for evaluation. The definition of "certain cases" is "whenever the conclusion of the goal has the form $\forall x{:}\,T.\,\exists y{:}\,T'.\,A$." Here T and T' are type symbols (nat, for instance) and A is some formula potentially involving x and y. The reason that we are interested in such cases is that the justification for such a formula, according to the constructive interpretation of the quantifiers, is an ***effectively computable*** function which, given an object t of type T, returns an ordered pair consisting of an object t' of type T' and a justification for $A[t/x]$ (A with free x's replaced by t). In other words, the justification is a ***program*** which computes a function from T to T' satisfying the specification A. To take a trivial example, A might be $y = x^2$, and so the extracted object would compute the squaring function. Thus programs are obtained directly from constructive proofs.

This is the upshot of the reduction of programming to proving alluded to above. Programs are specified by $\forall\exists$ assertions describing their behavior as functions, a derivation of the assertion is built with RED, and the proof is then compiled into code by extraction.

One might get the idea from the above account that the user must construct complete derivation trees on a rule–by–rule basis. This would be unbearably tedious, and so ample facilities are provided to relieve the burden. PRL provides an extendable facility for building automated proof assistants which serve to "chop off" derivation trees at a level much higher than the axioms of the logic. These decision methods are built up from a basic repertoire of domain–specific decision procedures by a collection of *tactics*. The idea of using tactics for proof development was first explored in the LCF project at Edinburgh [Gordon, Milner, & Wadsworth 79]. Constable and Bates further developed these ideas for the interactive proof–editing environment of PRL, and the ML system was incorporated as a subsystem by Bates and Knoblock. See [Constable, Knoblock, & Bates 84] for a detailed presentation of the facilities for building automatic proof assistants in PRL.

Our experience with proof construction with PRL shows that the use of automated assistants is absolutely crucial to the success of the project. It would be completely unreasonable, as one can easily imagine, to be forced to demonstrate the truths of elementary arithmetic or to continually reprove the substituvity of equality. A large collection of patterns of elementary reasoning has been codified into a single rule called "immediate" which is able to decide the truth of a large, but difficult to specify, class of goals which the average person would agree are trivially true. The use of "immediate" and other combinations of decision methods allows the user to concentrate on the global structure of the proof, rather than on the trivial details.

This completes our overview of PRL. An enormous amount has been left unsaid, and many of the systems features and deficiencies remain unmentioned. The interested reader is encouraged to consult the works cited above for a complete introduction.

Introduction to Type Theory

In this section we give a brief introduction to type theory, largely by way of examples. We will concentrate on presenting a formalization of type theory and on the connections with the ideas discussed above. Such issues as the nature and existence of types and the techniques by which the semantics of type theory are given lie beyond the scope of this thesis. However, several excellent sources are available in the literature, among them the papers of Martin-Löf and Constable cited above, the forthcoming Ph.D. Thesis of Allen [Allen 85], and the notes from a series of lectures by Martin-Löf at Padova [Sambin 80].

Type theory, as we shall henceforth use the term, is formalized as a set of inference rules, with the basic unit of inference being the *hypothetical judgement* of the form $H \gg t = t' \in T$ where H consists of a (perhaps empty) sequence of *bindings* of the form $x{:}\,A$. To explain the meaning of a judgement, we first consider the case of empty H. Loosely speaking, the judgement $\gg t = t' \in T$ where T is a type means that t and t' are equal in type T, that is, that they denote the same object. A more precise statement of the meaning of a judgement requires that we have in mind a collection of basic types and type constructors whose semantics serve as the background against which we give the meaning of a judgement. The meaning hinges on certain general considerations which inform the specification of *any* type which we may conceive. We can sum these up by saying the specification of a type consists of a specification of its members and the equality relation for objects of that type. Therefore, knowing that T is a type entails that we know what its members are and what it means for two terms to be equal in T, and so we know the meaning of the above judgement. We express simple membership in a type as the reflexive instance of the equality judgement; that is, $t \in T$ abbreviates $t = t \in T$.

The notion of being a type is formalized as membership in a class of *universe* types of the form U_i where $i > 0$. The universe types are themselves types, each U_i a member of U_{i+1}. T is a type exactly when it is a member of some universe; it said to be a type of level i when it is a member of U_i. Furthermore, the universes are *cumulative* in the sense that all members of a universe U_i are also members of all U_j for $j > i$. The reader may wonder why there isn't just a single universe U in place of a countable chain of universes. The reason for this is a rather deep philiosophical issue; the intention is to preserve a predicative type structure, in contrast to, say, the impredicative system of Girard [Stenlund 72]. Types in ν-PRL are stratified in such a way that if T is built from other types T', and $T \in \mathsf{U}_i$, then $T' \in \mathsf{U}_j$ for $j < i$.

The meaning of judgements with H nonempty is given by induction on the length of H with the basis being the case of H empty given above. Let H be $H', x{:}\, A$ where $H' \gg A \in \mathsf{U}_i$ for some i. Then $H \gg t = t' \in T$ where $H \gg T \in \mathsf{U}_k$ means

$$H' \gg t[a/x] = t'[a/x] \in T[a/x]$$

for any a such that $H' \gg a \in A$, and furthermore whenever $H' \gg a = a' \in A$,

$$H' \gg T[a/x] = T[a'/x] \in \mathsf{U}_k$$

and

$$H' \gg t[a/x] = t[a'/x] = t'[a/x] = t'[a'/x] \in T[a/x].$$

In other words, we insist that the judgement $t = t' \in T$ be *functional* in the hypotheses in the sense that any replacement of x by a leads to a valid judgement, and in addition that any replacement by two equal objects of type A (in the context H') leads to an equivalent judgement (in the sense that the components are equal). Notice that our induction is well-founded despite the apparent circularity of stipulating "where $H \gg T \in \mathsf{U}_k$" since the universe types are trivially equal to themselves in any context.

It must be emphasized that the use of the judgement form $H \gg t = t' \in T$ *presupposes* that $H \gg T \in \mathsf{U}_k$ (for some k) and that for each $x{:}\, A$ in H, $H_x \gg A \in \mathsf{U}_i$ (for some i), where H is $H_x, x{:}\, A, H'$. (H_x is called the *left-context* of x in H.) That is, we cannot make a judgement about the members of a type without first having the type in hand (so to speak), and we cannot bind a variable x without having a type to bind it to. When put this way, it seems to be completely obvious, but it is easy to overlook this assumption. One reason is that in most cases (such as the typed λ-calculus) the system of types is given beforehand, separately from the definition of the rest of the language. The restriction of certain expressions to types is then expressed by judicious choice of names of syntactic variables. But in the present context the type structure is much more complex and no such simple mechanism applies.

In order to illustrate these ideas we give some examples of valid rules for making judgements in type theory. These rules are taken from a formalization of the theory developed in Chapter 2. This formalization must itself be justified on the basis of the semantics of the type system, but that is the subject of another thesis [Allen 85]. Here we simply assume that the rules are valid.

The formalization of type theory that we shall employ has much the same flavor as that of the λ-PRL logic — the units of inference are judgements (whose syntactic similarity to the goals of λ-PRL is no accident, as we shall see), and derivations are structured as trees. The rules are presented in essentially the same goal–subgoal style.

For instance, consider the hypothesis rule:

$$H_1, x{:}\, A, H_2 \gg x \in A \quad \text{by hypothesis}$$

Paraphrased, this says that we may judge x to be a member of type A whenever x is bound to A in the antecedent, that is, whenever we have assumed that the type A is inhabited by x. Notice that by our supposition about sequents, we know that A is a type under the assumptions H_1.

For a more involved example, we introduce the (simple) Cartesian product of two types. Informally, the product type $A \times B$ consists of ordered pairs $\langle a, b\rangle$ where a is of type A and b is of type B. The equality relation on pairs is, as usual, component-wise. First, we have a rule for the *formation* of the Cartesian product of two types,

$$\begin{array}{l} H \gg A \times B \in \mathsf{U}_k \quad \text{by intro} \\ \quad H \gg A \in \mathsf{U}_k \\ \quad H \gg B \in \mathsf{U}_k \end{array}$$

That is, $A \times B$ is a type of level k whenever A and B are. Then we have a rule for *introducing* pairs:

$$\begin{array}{l} H \gg \langle a, b\rangle \in A \times B \quad \text{by intro} \\ \quad H \gg a \in A \\ \quad H \gg b \in B \end{array}$$

This is a straightforward formalization of the definition given above. We have cheated a bit here for the sake of clarity — officially, each of these rules ought to be in the form of an equality. So the product introduction rule would actually be:

$$\begin{array}{l} H \gg \langle a, b\rangle = \langle a', b'\rangle \in A \times B \quad \text{by intro} \\ \quad H \gg a = a' \in A \\ \quad H \gg b = b' \in B \end{array}$$

and similarly for the formation rule.

Now we come to a new aspect of type theory, namely the so–called *elimination* rules. These rules define equality and typing for terms whose role is to *compute with* objects of a product type. Loosely speaking, we have so far said what the objects of a product type *are*, but we have given no means of *doing anything* with them. The elimination rule for products involved a term called **spread** whose role is to split a pair into its component parts, as we shall see below. Here is the rule:

$$\begin{array}{l} H \gg \mathsf{spread}(e; u, v.t) \in T \quad \text{by elim} \\ \quad H \gg e \in A \times B \\ \quad H, u\colon A, v\colon B \gg t \in T \end{array}$$

The notation used here is probably unfamiliar. In type theory there are many terms which bind variables (in the usual sense), but many have the property that they sometimes bind more than one variable in more than one component. In the case of **spread**, the variables u and v are bound by **spread** in t, and none are bound in e. Hence the notation, reminiscent of λ-calculus, $u, v.t$. The above rule says that if e is a member of $A \times B$, and if $t \in T$ whenever $u \in A$ and $v \in B$, then $\mathsf{spread}(e; u, v.t)$ is in T (it is implicit in the top–down formulation of the rule that T cannot involve u and v).

The intuition for this rule lies in the meaning of **spread**. We said above that spread "splits" the pair e, but we haven't said how it this happens. That is given by the *computation rule* for spread, as follows:

$$\begin{array}{l} H \gg \mathsf{spread}(\langle a, b\rangle; u, v.t) = t[a, b/u, v] \in T \quad \text{by reduction} \\ \quad H \gg \langle a, b\rangle \in A \times B \\ \quad H, u\colon A, v\colon B \gg t \in T \end{array}$$

That is, spread substitutes the left component of a pair for u and the right component for v in t (the notation $t[a, b/u, v]$ denotes simultaneous, capture-avoiding substitution of a and b for free occurrences of u and v in t).

We have now completely specified the rules for (a simplified version of) the Cartesian product type. The reader is encouraged to convince himself of the truth of these rules on the basis of the semantics outlined above.

Readers familiar with Martin-Löf's formulation of type theory will notice that we have reduced the four forms of judgement down to one by replacing type and type equality judgements by membership and equality in a universe and by taking simple membership judgements to be abbreviations for reflexive equality judgements. We did this mainly for reasons of notational economy and to simplify the formalization of the theory. The other difference is that judgements carry their hypotheses around explicitly. Further discussion of these matters is deferred to Chapter 2.

We conclude this section with a discussion of the connection between type theory and the logic of λ-PRL. The connection between the two can be summarized in one phrase: *propositions–as–types*. This principle, first observed by Curry [Curry 1958] and Howard [Howard 80], amounts to the identification of a proposition with the type of its justifications. We were deliberately vague in our discussion of λ-PRL about the nature of justifications. We are now in a position to explain this notion further.

Recall that we said that a justification is some form of construction. One role of type theory is to clarify the notion, so central to intuitionistic principles, of a construction. Given that the theory is sufficiently rich, we can treat propositions as certain types, the truth of a proposition being established whenever we can judge it to be inhabited in the framework of type theory. Under this conception of propositions, the binding $x{:}\,A$ (where A is a formula) amounts to the same thing as the assumption that A is true (*i.e.*, that it is inhabited). Taking this idea seriously, we shall introduce in Chapter 2 a judgement form "A is inhabited" in which the inhabiting object is suppressed. This should look familiar since it is just the extraction process discussed above in the context of λ-PRL. In this way we can completely absorb λ-PRL into ν-PRL as a restricted case, thereby achieving a smooth transition from the old to the new logic.

For readers unfamiliar with the notion of propositions–as–types, we give a brief summary. Each of the propositional connectives is identified with a type constructor as follows: $\wedge$ with $\times$, the Cartesian product, $\vee$ with $+$, the disjoint union, and $\Rightarrow$ with $\rightarrow$, the (constructive) function space. Negation is expressed as an implication, $\neg A$ is $A \rightarrow \mathsf{void}$, where void is the empty type. The quantifiers $\forall, \exists$ are represented as the dependent function and product constructors, Π and Σ. At the basis we have the empty type void and the equality types $\mathsf{eq}(a, b, A)$ which are inhabited (by axiom) exactly when $a = b \in A$. This identification yields both a precise specification of the justifications and a precise account of the constructive interpretation of the logical connectives and quantifiers.

Finally, we remark that nearly all cases of the judgement form $H \gg t = t' \in T$ are undecidable, even under such restrictions as H empty, $t \equiv t'$, T restricted to specific types such as nat or a universe, or limiting ourselves to derivable (as distinguished from true) judgements. Here are some examples. We cannot decide whether an arbitrary type is inhabited (regardless of whether or not H is empty) because, by the encoding of propositions as types, this would be equivalent to deciding the truth of sentences of, say, first–order constructive arithmetic. The problem remains undecidable even if we limit ourselves to *provable* judgements since provability in arithmetic is undecidable as well. Since we can easily encode primitive recursive arithmetic into type theory, equality for the type of natural numbers is undecidable, even when restricted to the case of provable equalities [Rose 61]. We can use this fact to show that membership in any fixed type is undecidable by building a term whose proof of membership in that type depends on a proof of equality of two terms of type nat. This precludes, for instance, the construction of a decision procedure for "typehood", that is, membership in a universe. While not every type is a proposition, those that are do not form a sufficiently limited class that membership is decidable.

Overview of the Thesis

The subject of Chapter 2 is the formalization of type theory. The main concern will be to develop a set of proof rules which are suitable for use in the context of the PRL development environment described above. This means that the logic must be tailored to have, as much as possible, the properties necessary to support refinement and extraction. Furthermore, the logic will be designed so as to minimize the proof obligations incurred by the restrictions on judgement forms discussed above.

Recall that in formulating the ν-PRL logic it was stipulated that a judgement $H \gg t = t' \in T$ presupposes that T is a type in H and that each of the types in H is a type in its left context. This seemed like a perfectly reasonable supposition, as it certainly is, but from a practical point of view this is an excessive burden. Consider how one would have to proceed. Before one could present a goal to RED, one must first show that the goal is *well-formed* in the sense that the above conditions are met. This means that one must *prove* that each of the expressions used as a type in the goal is in fact a type (in the appropriate context). During the process of building such a derivation, more such obligations are incurred (though only finitely many!), further increasing the burden. And all of this must be done before the user can turn his attention to the problem of interest! Notice that these proof obligations are not unique to the ν-PRL logic — in Martin-Löf's papers they are omitted, for the sake of brevity, from the presentation of the rules.

By way of comparison, note the λ-PRL logic has similar restrictions. In order to use an expression to denote a proposition, one must first show that the expression in fact denotes a proposition. But in the limited case of first-order predicate logic, this is trivially decidable by standard parsing techniques, and so can be automatically checked by RED whenever a goal is presented to it. Since the property of denoting a proposition in ν-PRL is undecidable, no such automatic checking can take place (though checkers can be built for certain cases).

Another reason why the formulation of the ν-PRL logic outlined above is unacceptable as a basis for PRL is that it wantonly violates the "constructible subgoals" principle isolated as a crucial property of a PRL logic. A closely related deficiency is that the logic, as it stands, supports no notion of refinement. As a simple illustration, consider how one would show that $A \wedge B$ is true (it is assumed here that A and B are known to denote propositions). By the propositions–as–types principle, one shows that $A' \times B'$ is inhabited, where A' and B' are the type analogs of A and B. But to do this, one must set up a goal of the form

$$\gg \langle a, b \rangle \in A' \times B'$$

But this means that one must *already know* the justifications a of A' and b of B', clearly an unacceptable situation. What we would like is to be able to show merely that A' and B' are inhabited, leaving it to the system to produce (extract) the inhabiting object.

The subject of Chapter 2 is the construction of a suitable logic for type theory by giving a sequence of transformations, beginning with a simple formalization that possesses almost none of the requisite properties and ending with a formalization suitable for use in the PRL system. The main result is that one can structure the logic in such a way that well-formedness of the consequent type is demonstrated simultaneously with inhabitation, thereby eliminating the need for a separate demonstration. Well-formedness of the antecedent is assumed, rather than presupposed. We prove that this formulation preserves the consistency of the logic and that well-formedness is indeed achieved. The logic is also formulated to support extraction and, to some extent, constructibility of subgoals. Some of the difficulties associated with achieving these properties are discussed.

In Chapter 3 is the foundation of the treatment of automated assistance developed in the remainder of the thesis. Experience with λ-PRL and a prototype implementation of ν-PRL focused attention on the problem of providing automated assistance with equality. This led directly to consideration of the type membership problem, and to the construction of a set of tools on which to build such assistants. These tools are the subject of Chapter 3.

The essential idea is that we can decide the membership relation by encoding derivation information in the terms themselves in such a way that testing membership is reduced to inspecting the type superscript of the term. This technique is familiar from combinatory logic and the typed λ-calculus; our contribution consists largely in extending this idea to the case of a type structure which involved binding operators and in which the class of type terms is not given in advance.

In anticipation of the work in Chapter 4, a logic of typed terms involving a class of *type variables* is also introduced. Type variables are syntactic variables which range over expressions which are demonstrably types (in a context associated with the variable). It will turn out that the replacements for type variables may involve bound variables, and so substitution must deliberately incur capture. A notion of substitution of type expressions for type variables is defined.

Finally, an informal semantics of the logic of typed terms and typed term schemes is given. The justification is given in terms of the principles of type theory given by Martin-Löf to justify his formalisms. This step is necessary since we would like to use the system of typed terms to build decision methods for type theory, and so it must be argued that the (semantic) bounds of the theory which we are formalizing in PRL are not exceeded.

In Chapter 4 the tools of Chapter 3 are put to work to build a decision method for judgements of the form $H \gg t \in T$. Any such method is necessarily incomplete both from a theoretical and practical point of view (that is, it is sure to fail on both the usual diagonal cases and many ordinary cases as well). The intention is not to present the definitive solution to the problem, but rather to outline a flexible, tunable algorithm whose effectiveness will ultimately be determined empirically.

The technique employed is a generalization of Milner-style type inference [Gordon, Milner, & Wadsworth 79,Damas & Milner 82] to our logic of typed terms. To decide $t \in T$, we build, by a process known as *annotation*, typed analogs of t and T (using H as a basis) and check whether or not the results "match." The annotation algorithm uses several heuristics to make plausible guesses of the types of terms. Many well-known techniques can be brought to bear here. For instance, the methods of Boyer and Moore for guessing induction hypotheses can be used to annotate the PRL induction form.

An essential component of the annotation algorithm is a generalization of the unification algorithm [Robinson 65], called *constrained unification*. The idea is to define a notion of unification of typed term schemes where the instantiatable variables are the type variables mentioned above. This requires that an extended notion of instantiation be used and also that potential instances of a type variable be constrained to be well-formed in a context associated with that variable. The algorithm, which reduces smoothly to ordinary unification under certain natural restrictions, should be useful in many contexts outside of type theory such as the extension of the type structure of ML to allow dependent function types.

Finally, Chapter 5 treats the problem of equality. As for membership, the intention is to outline techniques for deciding a subrelation of equality rather than to give a definitive solution. The method considered draws on the results of Chapter 4. A decidable theory of equality for typed terms is defined, incorporated into the unification algorithm, and used as the basis of a decision method for judgements of the form $H \gg t = t' \in T$. This extension of the unification algorithm can also be used by the annotation algorithm, thereby enhancing its effectiveness at typing terms. The algorithm allows for the use of hypothesized equations (drawn from H) as well as the computation rules of the type theory. Preliminary experience with ν-PRL indicates that these features of the method are crucial to its effectiveness in practice.

Chapter 6 summarizes the results of the thesis and suggest directions for future research.

Chapter 2

Structure of the logic

This chapter is devoted to the development of the PRL proof rules. The main concern will be to structure the logic to support simultaneous well–formedness proofs, refinement, and extraction. This will be done in stages, beginning with a base logic whose validity is taken as given, and proceeding by consistency–preserving transformations to a logic suitable for use in the PRL proof development environment. The result of these transformations is a formalization of type theory that conforms to the strictures of the PRL proof development environment, that minimizes the proof obligations associated with well–formedness, and that is consistent relative to a simple Martin-Löf–style base logic.

The aim is not to develop the PRL logic as it is actually implemented since this would require an account of a host of uninteresting details. For instance, the ν-PRL system includes many logically redundant, but convenient, rules. It must also provide mechanisms for such logically trivial but practically important issues as the choice of variable names. See [PRL Staff 85] for a complete description of the PRL system.

Much of the work discussed in this chapter was conducted in close collaboration with R. Constable, J. Bates, and S. Allen. The method of presentation and the statement and proofs of the theorems are the author's.

2.1 Syntax

In order to discuss the various formulations of type theory, we have to set down the syntactic conventions that we will be using, here and in the remainder of the thesis.

To be completely precise, we should begin with a specification of the language of type theory. But for the most part, the inference rules, together with some general remarks to follow, serve just as well to define the set of terms used.

A denumerable class of objects to be used as variables is assumed; let u, v, x, y, and z denote variables. A term is either a variable or a compound term of the form:

$$o(t_1; \ldots; t_m; a_1; \ldots; a_n)$$

where $m, n \geq 0$, o is some term constructor, the t_i are terms, and the a_j are abstractions, to be explained below. Let e, s, t, A, B, S, and T range over terms. Capital letters are usually used for terms intended to denote types, but since this is not a strictly syntactic condition, the reader is warned that this convention serves only as a hint. Each constructor o uniquely determines m and n and the arity of the component abstractions (if any).

An *abstraction* is a configuration of the form:

$$x_1, \ldots, x_k.t$$

The index k is called the *arity* of the abstraction. Abstractions occur only as components of compound terms, and so are not regarded as terms. Abstractions are used to indicate, for a term built with constructor o, which variables are bound by o in which subterms. For instance, in $\lambda(x.t)$, x is bound in t by the constructor λ, and in $\mathsf{spread}(e; u, v.t)$, u and v are bound in t by spread.

A *hypothesis list* consists of a finite, perhaps empty, sequence of *bindings* of the form $x{:}T$. Let H range over hypothesis lists, and write them as comma-separated lists of bindings. If $x{:}T$ occurs in H we say that x is *bound in H to T*. No repetitions may occur among the variables bound in H. In the notation $H, x{:}T$ it is tacitly assumed that x has been chosen so as to avoid repetitions in the result; any concomitant renamings of x in associated terms is also assumed to occur. Given that x is bound in H, write H_x for the initial segment of H up to, but not including, that binding. H_x is known as the *left context* of x in H.

A *sequent* has either the form:

$$H \gg t = t' \in T$$

or the form

$$H \gg T$$

depending on the context. The hypothesis list H is called the *antecedent*, *binding list*, or simply the left–hand side of the sequent, and $t = t' \in T$ is called the *consequent* or just right–hand side. The term T is called the *goal type* of the sequent, and H is called its left context.

We use the term *judgement* to refer to the content (as opposed to the syntactic form) of a sequent according to the semantics of type theory. A judgement $H \gg t = t' \in T$ is said to be *valid* exactly when t and t' denote the same object of type T in context H. This is determined by the meaning of hypothetical judgements given in Chapter 1 and the semantics of the type denoted by T. For a complete treatment of the semantics of type theory, see [PRL Staff 85].

The formalizations of type theory that we have written below are given single–letter names such as **M**. The derivability relation is written $\vdash$, preceded by the name of the logic. For instance, we write

$$\mathbf{M} \vdash H \gg t = t' \in T$$

to mean that the indicated sequent is derivable in the set of rules named **M**.

Simultaneous substitution of $e_1, \ldots, e_n$ for free occurrences of $x_1, \ldots, x_n$ in t, avoiding capture by suitable renaming by bound variables, is written

$$t[e_1, \ldots, e_n / x_1, \ldots, x_n]$$

This notation is extended to a hypothesis list H on a term–by–term basis, avoiding capture and the introduction of repetitions among the variables bound in H.

The symbol $\equiv$ is used to denote the identity relation on terms; identity modulo naming of bound variables is written $\equiv_\alpha$.

$$H \gg A + B \in \mathsf{U}_k \quad \text{by formation}$$
$$\quad H \gg A \in \mathsf{U}_k$$
$$\quad H \gg B \in \mathsf{U}_k$$

$$H \gg \mathsf{inl}(a) \in A + B \quad \text{by intro}$$
$$\quad H \gg a \in A$$
$$\quad H \gg B \in \mathsf{U}_k$$

$$H \gg \mathsf{decide}(e; u.s; v.t) \in T(e)_z \quad \text{by elim}$$
$$\quad H \gg e \in A + B$$
$$\quad H, u{:}\, A \gg s \in T(\mathsf{inl}(u))_z$$
$$\quad H, v{:}\, B \gg t \in T(\mathsf{inr}(v))_z$$
$$\quad H, z{:}\, A + B \gg T \in \mathsf{U}_k$$

$$H \gg A \in \mathsf{U}_k \quad \text{by structure}$$
$$\quad H \gg A + B \in \mathsf{U}_k$$

Figure 2.1: Selected **M** Union Rules

2.2 The base logic

We begin our development with **M**, the base logic for **PRL**. **M** is a formulation of type theory closely related to that given in [Martin-Löf 82], but somewhat more oriented toward the constraints of the **PRL** system. We take **M** to be the "official" formulation of the **PRL** type theory. As such, it will serve as the basis for the construction of the proof rules and will be regarded as the background theory against which the rules are judged.

M is a logic of sequents presented in the "inverted" style introduced in Chapter 1. The complete set of rules appears in Appendix A; a representative set of rules, drawn from those for the union type, appear in Figure 2.1. For the sake of brevity, these rules are presented in membership rather than equality form.

The rules are formulated along the same lines as those in [Martin-Löf 82], with some notable exceptions. First, there is just one form of judgement, the equality judgement, as discussed in Chapter 1. Second, the type structure is somewhat more restricted than that underlying Martin-Löf's system. For him, two types are equal exactly when they have the same members and the same equality relation. In PRL, however, type equality is partially intensional — two types can be equal only if they are built from the same constructors. This is reflected in the "type structure" rules of the M logic which are absent from Martin-Löf's formulation. This intensional aspect of the equality relation is required for several reasons, one of which is that it is crucial to the treatment of well-formedness. Other more philosophical reasons are discussed in [Constable 83a].

When formulating a set of proof rules for type theory, one must come to terms with the issue of *well-formedness*. The term "well-formedness" is used in two related ways, reflecting the two assumptions about sequents made in Chapter 1 (see page 11). Given a sequent $H \gg t = t' \in T$, we say that T is *well-formed* exactly when $H \gg T \in \mathsf{U}_k$ holds for some k. We say that a binding $x{:}T$ in H is well-formed if $H_x \gg T \in \mathsf{U}_k$ holds for some k; a hypothesis list H is well-formed if every binding in H is well-formed. The definition of a judgement presumes that the bindings and goal type are well-formed. A formalization of type theory as a logic of judgements must somehow deal with this stipulation.

In Martin-Löf's formalization this issue is handled by the use of "hidden premises" of the rules. The only indication of the what sorts of premises are omitted from the presentation of the rules appears in [Martin-Löf 82] on page 21:

> Moreover, in those rules whose conclusion has one of the forms $a \in A$ and $a = b \in A$, only those premises will b explicitly shown which have these vert same forms. This is in agreement with the practice of writing, say, the rules of disjunction introduction in predicate calculus ... without showing explicitly the premises ... are formulas.

Since the convention is described in such vague terms it is not at all clear what the omitted premises may be in any given rule. But it seems that at the very least there must be premises which ensure that all expressions used as types in a rule instance must in fact be types. In the case of the union introduction rule this means that the complete version is, using the notation of M,

$$
\begin{array}{ll}
H \gg \mathsf{inl}(a) \in A + B & \text{by intro} \\
\quad H \gg a \in A & \\
\quad H \gg A + B \in \mathsf{U}_i & \\
\quad H \gg A \in \mathsf{U}_j &
\end{array}
$$

In the case of the union elimination rule, it is much less clear what the implied premises of the rule must be. The first formulation of this rule by Constable and Bates did not include the fourth subgoal which ensures that T is functional in $A + B$ (see Figure 2.1). It was thought that the functionality of T follows from the second and third subgoals, together with the fact that any member of a union type is of the form $\mathsf{inl}(e)$ or $\mathsf{inr}(e)$. However, in general it is not the case that any member *is* of this form, but that any member is *equal to* one of this form. This leaves open the possibility that T can exploit this distinction between identity and equality, thereby violating functionality in $A + B$. I know of no way to build such a T in the logic as it stands, nor do I know of any way to prove that this cannot be done. But I *do* know that it is possible to extend the logic so that such a T can be built,[1] so in the interest of safety the fourth subgoal is included. It is not clear to me whether or not this premise is intended to be one of the implicit premises in Martin-Löf's formulation of the rules. Note, however, that the Göteborg formalization of type theory does not include such a premise [Petersson 82].

Once one fills in all of the implicit premises of the rules, one quickly sees that the construction of a derivation in Martin-Löf's formulation of type theory is a rather tedious process. It is certainly the case it is necessary for these premises to be *true* in order that the rules have any meaning at all. But it is possible to formalize type theory in a way that affords considerable economy of proof obligations.

M is formulated in such a way that a derivation of $H \gg t = t' \in T$ includes a derivation of the well-formedness of T. For instance, in the case of the union introduction rule, the second subgoal is removed and the third is replaced by $H \gg B \in \mathsf{U}_i$. See Figure 2.1. By construction, the first subgoal establishes that A is a type; taken together with the second, we have sufficient grounds for $A + B$ to be a type. This technique of incorporating a derivation of the well-formedness of the goal type of a sequent into a

[1] Stuart Allen, November, 1984, private communication.

derivation of equality in that type considerably shortens proofs. Note, however, that we cannot do away with all of the implied subgoals. For instance, the fourth subgoal of the union elimination rule remains in M.

Note that we can easily take advantage of prior derivations of well-formedness of the goal type. Given the goal $H \gg t = t' \in T$ for which $H \gg T \in \mathsf{U}_k$ has been established, the goal type T can be marked "known to be well-formed." This marking can be automatically propagated to the subgoals as follows. Consider an arbitrary rule whose main goal is of the form $H \gg t = t' \in T$ where T is marked as well-formed and let $H' \gg s = s' \in S$ be any subgoal of that rule. If the well-formedness of S follows from that of T, then S can be marked well-formed. Furthermore, any subgoal of the form $H' \gg S \in \mathsf{U}_i$ which follows from the truth of $H \gg T \in \mathsf{U}_k$ may be suppressed. For instance, in the case of the union introduction rule, if it is known that $A+B$ is well-formed, then it is known, by the type structure rule, that both A and B are well-formed, and hence A can be marked as such and the second subgoal can be suppressed.

This method is adequate to ensure the well-formedness of the goal type of a sequent derived in M, but it does not address the issue of well-formedness of the hypothesis list. Recall that expressions used in bindings are presupposed to be types in their left context. In M this is enforced by adding a subgoal to the hypothesis rule which guarantees that this is the case:

$$\begin{array}{ll} H_1, x\colon T, H_2 \gg x \in T & \text{by hypothesis} \\ \quad H_1 \gg T \in \mathsf{U}_i & \end{array}$$

One can view this addition as providing a place for the well-formedness derivations which are assumed to exist according to the stipulations of Chapter 1. Notice that this formulation of the rule is slightly more liberal than the presupposition of well-formedness since an unused binding need never be shown to be well-formed.

An essential property of M is summarized in the following theorem:

Theorem 2.1 *If* $\mathsf{M} \vdash H \gg t = t' \in T$, *then* $\mathsf{M} \vdash H \gg T \in \mathsf{U}_k$ *for some* k.

Proof The proof is by a routine induction on derivations. The base cases are the hypothesis rule and the universe formation rule. In the case of the hypothesis rule, it follows immediately that if $H_1 \gg T \in \mathsf{U}_k$, then $H_1, x{:}\,T, H_2 \gg T \in \mathsf{U}_k$, as required. Notice that all of the formation rules (those with goal type some universe) are trivial since $\mathsf{U}_i \in \mathsf{U}_{i+1}$ for any i and so each is well-formed in any context. For the induction, simply ensure that the induction hypothesis applied to each of the subgoals, together with the subgoals themselves, are sufficient to guarantee that the goal type is well-formed. For an example, see the discussion of the union introduction rule above. ▮

By way of comparison with Martin-Löf's formalization of type theory, note that the sequent

$$x_1{:}\,T_1, \ldots, x_n{:}\,T_n \gg t = t' \in T$$

is almost equivalent to

$$t = t' \in T \ (x_1 \in T_1, \ldots, x_n \in T_n)$$

The main difference is that in Martin-Löf's notation the parenthetical list of typings of the variables is a *summary* of the undischarged hypotheses in the derivation tree rooted at $t = t' \in T$. In M, however, the hypothesis list H can contain "extra" bindings that are unused in the derivation. Thus, we can regard the derivability of $H \gg t = t' \in T$ in M to mean that a derivation of $t = t' \in T$ can be constructed in Martin-Löf's system with undischarged hypotheses among those in H.

2.3 Taking well–formedness seriously

M goes a long way toward the goal of minimizing the proof obligations incurred by well–formedness considerations. In this section we will develop a formalization J of type theory that improves on M in two ways:

1. Well–formedness is treated directly in the logic, rather than as a proof–theoretic property of the logic.

2. The hypothesis rule is further liberalized by assuming that hypothesis lists are well–formed.

In keeping with the transformation methodology of this chapter, J will be explained in terms of M. This means that its validity is verified by a proof–theoretic reduction of derivations in J to derivations in M, thereby establishing relative consistency. The alternative, in some ways philosophically superior, is to validate J with respect to an informal semantics of type theory, just as we have done (implicitly) with M. The steps of the verification are essentially the same in both cases, so the proof of Theorem 2.2 (given below) can be taken as an outline of such a verification. The approach taken here is founded on the premise that the issues that motivate the structure of the rules are, for the most part, proof–theoretic, based on the desire to fit the logic into the PRL proof development environment. In this section, therefore, M is taken to be the "official" specification of type theory.

We begin by noting that whereas M ensures well–formedness of the goal type of derivable sequents, it is unable to take advantage of it. It is desirable to be able to infer that a term is a type from the fact that it is inhabited. But that is not part of the *meaning* of a goal in M, so there cannot be a rule of the form

$$\begin{array}{l} H \gg T \in \mathsf{U}_i \quad \text{by inhabitation} \\ \quad H \gg t = t' \in T \end{array}$$

since the premise ensures only that *some*, not *any* index i is acceptable. For instance, T might be $\mathsf{U}_1 + \mathsf{U}_1$, a level 2 type which is not a level 1 type, and so we ought not be able to conclude $T \in \mathsf{U}_1$ from, say, $\mathsf{inl}(\mathsf{nat}) \in T$!

The information built into M derivations can be recovered by defining a new form of sequent which records the universe level of expressions used in bindings and as goal types. To this end, the syntax of bindings and equalities is extended by adding a *level tag*, the resulting forms now being $x{:}T @ i$ and $t = t' \in T @ i$, respectively. To distinguish the new forms from the old, we call these bindings *tagged bindings*, and hypothesis lists built from these are called *tagged hypothesis lists.* Let H range over tagged hypothesis lists, and write H° to denote its untagged counterpart obtained by "erasing" the level tags.

Sequents now have the form $H \gg t = t' \in T @ i$, where H is a tagged hypothesis list. There are now two components to the meanings of bindings and equalities. A binding $x{:}T @ i$ is intended to mean both $T \in \mathsf{U}_i$ and

$x \in T$ and the sequent $H \gg t = t' \in T @ i$ means both $H^\circ \gg T \in \mathsf{U}_i$ and $H^\circ \gg t = t' \in T$. As remarked above, when we say "means" in this context, we mean that the indicated sequents are to be derivable in **M**.

Given this new form of sequent, the rule

$$\begin{array}{l} H \gg T \in \mathsf{U}_k @ l \quad \text{by inhabitation} \quad [k < l] \\ \quad H \gg t = t' \in T @ k \end{array}$$

can be safely added. This rule affords the ability to use well-formedness within derivations.

The second issue that we will address in J is related to refinement. Recall that proof construction in PRL is a top-down process, beginning with a goal, proceeding by refinement steps to the leaves of a derivation tree. With this in mind we see that M's hypothesis rule is rather cumbersome to use since it requires that all premises be shown to be well-formed each time they are used. Furthermore, as we shall see below, all hypotheses introduced by a refinement step will be well-formed. Therefore, the obligation to demonstrate this whenever such a binding is used is strictly redundant. This difficulty is avoided by taking the hypothesis rule to be without subgoals, further paring down the well-formedness obligations of the proof developer. The new form of the rule is:

$$H_1, x{:}T @ i, H_2 \gg x \in T @ i \quad \text{by hypothesis}$$

But there is the difficulty that the antecedent of an instance of this rule may not be well-formed, apparently leading to invalid derivations. The answer is that the truth of the judgement expressed by a sequent derived in J is predicated on the assumption that the antecedent is well-formed. This leads to the following definition:

Definition 2.1 *A tagged hypothesis list H is* well-formed *iff for each binding $x{:}T @ i$ in H, $\mathsf{M} \vdash H_x^\circ \gg T \in \mathsf{U}_i$. In particular, the empty hypothesis list is always well-formed.*

The derivability relation for J is restricted to goals whose antecedent is well-formed; that is, it is implicit in

$$\mathsf{J} \vdash H \gg t = t' \in T @ i$$

$H \gg A + B \in \mathsf{U}_k @ l$ by formation $[k < l]$
  $H \gg A \in \mathsf{U}_k @ l$
  $H \gg B \in \mathsf{U}_k @ l$

$H \gg \mathsf{inl}(a) \in A + B @ k$ by intro
  $H \gg a \in A @ k$
  $H \gg B \in \mathsf{U}_k @ l$

$H \gg \mathsf{decide}(e; u.s; v.t) \in T(e)_z @ k$ by elim
  $H \gg e \in A + B @ i$
  $H, u{:}\, A @ i \gg s \in T(\mathsf{inl}(u))_z @ k$
  $H, v{:}\, B @ i \gg t \in T(\mathsf{inr}(v))_z @ k$
  $H, z{:}\, A + B @ i \gg T \in \mathsf{U}_k @ l$

$H \gg A \in \mathsf{U}_k @ l$ by structure
  $H \gg A + B \in \mathsf{U}_k @ l$

Figure 2.2: Selected J Union Rules

that H is well–formed. Only derivations satisfying this condition are considered because nothing of interest can be said about goals which fail to have well–formed hypotheses.

The intended meaning of derivations in J is given by the following theorem.

Theorem 2.2 *If* $\mathsf{J} \vdash H \gg s = t \in T @ i$, *then* $\mathsf{M} \vdash H^{\circ} \gg T \in \mathsf{U}_i$ *and* $\mathsf{M} \vdash H^{\circ} \gg s = t \in T$.

Immediate consequences of Theorem 2.2 are:

Corollary 2.1 J *is relatively consistent with* M.

Corollary 2.2 *The inhabitation rule for* J *is valid.*

Proof of Theorem 2.2 The proof is by induction on derivations in J with a case analysis on the last step of the proof. Assume that the theorem holds for all subderivations whose antecedents are well–formed, and show that it holds for the entire derivation. At each step the main concern is to ensure that the antecedents of the subgoals of a rule are well–formed so that the induction hypothesis applies. It is then a simple matter to recombine the M–derivations yielded by the induction hypothesis to obtain the result. For the sake of brevity, only some representative cases, reproduced in Figure 2.2, are considered. The other cases are proved by similar methods, and are therefore omitted. A complete presentation of J appears in Appendix B.

- (Hypothesis) By the well–formedness of H, $\mathsf{M} \vdash H^\circ_x \gg T \in \mathsf{U}_i$, hence $\mathsf{M} \vdash H^\circ \gg T \in \mathsf{U}_i$. $\mathsf{M} \vdash H^\circ \gg x \in T$ is obtained by an application of the hypothesis rule.

- (Union formation) $\mathsf{M} \vdash H^\circ \gg \mathsf{U}_k \in \mathsf{U}_l$ is immediate since $k < l$. $\mathsf{M} \vdash H^\circ \gg A+B = A'+B' \in \mathsf{U}_k$ follows directly from the induction hypothesis.

- (Union introduction left) Apply the induction hypothesis to both subgoals and union formation to obtain $\mathsf{M} \vdash H^\circ \gg A + B \in \mathsf{U}_k$. Similarly, induction and union introduction yields $\mathsf{M} \vdash H^\circ \gg \mathsf{inl}(a) \in A + B$.

- (Union elimination) The interesting part of this case is showing that the antecedents of the subgoals are well–formed. The rest follows immediately by induction. By the induction hypothesis applied to the first subgoal, $\mathsf{M} \vdash H^\circ \gg A + B \in \mathsf{U}_i$, so $H, z{:}\, A + B\,@\,i$ is well–formed. By the type structure rule, $\mathsf{M} \vdash H^\circ \gg A \in \mathsf{U}_i$ and $\mathsf{M} \vdash H^\circ \gg B \in \mathsf{U}_i$, giving well–formedness of $H, u{:}\, A\,@\,i$ and $H, v{:}\, B\,@\,i$, respectively. The rest follows by the induction hypothesis applied to each of the subgoals.

- (Inhabitation) Directly from the induction hypothesis and the fact that $i < j$.

The remaining cases are proved using the same methods. ∎

This completes the development of J. In the next section the transformation process is completed with R, a refinement logic.

2.4 A refinement logic

In this section we take the last step toward a formulation of a refinement logic by taking into account the issues of extraction and constructible subgoals. Recall from the introduction that the PRL system is oriented toward proving propositions, typically specifications of programs. By the propositions–as–types principle, this means that the orientation of a logic for PRL must be toward demonstrating that types are inhabited. The inhabiting object occupies a secondary position — one is generally interested in the inhabiting object only once a proof has been carried out sufficiently far that extraction can occur and even then only for the purposes of evaluating it. Furthermore, the logic must have constructible subgoals; that is, the subgoals of a rule instance must be completely determined by the goal and the rule name together with its parameters.

The main task of this section is to reformulate type theory to be oriented toward propositions and extraction, rather than judgements of equality in a type. This is not to say that judgements are thrown out, just that they are relegated to a secondary status. With this in mind, sequents are redefined to be of the form

$$H \gg T\,@i \ \mathit{ext}\ t$$

where H is a tagged binding list, T is an expression (intended to denote a type of level i), and t is any term. The intended meaning of such a sequent is that the sequent $H \gg t \in T\,@i$ is derivable in J. This claim will be substantiated in Theorem 2.4.

The *ext* clause specifies the extracted object justifying T. In keeping with λ-PRL the extracted object is not displayed. Furthermore, since it is defined in terms of the extracted code associated with each of the subgoals, it is not available to RED at the time the refinement step is made. That is, the contents of the subgoals of a rule instance cannot depend on the *ext* clause associated with the goal. It is interesting to note that in the case of the λ-PRL logic, it is always possible to instantiate the subgoals without reference to this information — that is the reason why extraction works! But in the case of the ν-PRL logic, there is sufficient generality that this cannot always be achieved without difficulty.

It is still possible, in fact necessary, to retain the ability to deal directly with judgements. Extraction is a technique for showing that a type is inhabited by some hidden object. This is useful when the type in question happens to denote a proposition. But when the type is being used as a data type, it is often the case that one has an object in hand. For instance, one may wish to demonstrate that $\langle 1,2\rangle$ is an object of type $\mathsf{nat} \times \mathsf{nat}$ in the course of developing a program. A system which is completely oriented toward extraction would be unable to handle this. Another reason that we shall have need of judgements is for the treatment of equality.

Both of these issues are handled by reflecting the judgements of equality *into* the logic in the form of the **eq** type. Recall from Chapter 1 that the equality type $\mathsf{eq}(s,t,T)$ is inhabited iff $s = t \in T$, and so it expresses the *proposition* (as distinct from judgement) that s and t denote the same object in type T. So, to show $H \gg s = t \in T$, show instead that $\mathsf{eq}(s,t,T)$ is inhabited, *i.e.*, derive $H \gg \mathsf{eq}(s,t,T) \mathbin{@} k$ in R.

To further the analogy to the point of confusion, we write $s = t \ \mathsf{in}\ T$ for $\mathsf{eq}(s,t,T)$ so that an equality assertion has the form $H \gg s = t \ \mathsf{in}\ T$.

This use of equality types is connected to extraction by the following rule:

$$
\begin{array}{ll}
H \gg t = t \ \mathsf{in}\ T \mathbin{@} k \ \mathit{ext}\ \mathsf{axiom} & \text{by inhabitation} \\
\quad H \gg T \mathbin{@} k &
\end{array}
$$

Notice that the converse of this rule

$$
\begin{array}{ll}
H \gg T \mathbin{@} k \ \mathit{ext}\ t & \text{by explicit introduction} \\
\quad H \gg t \ \mathsf{in}\ T \mathbin{@} k \ \mathit{ext}\ \mathsf{axiom} &
\end{array}
$$

though valid, cannot be easily incorporated into R since it fails to have constructible subgoals. See the discussion about product introduction below for a treatment of this issue.

The complete R logic is presented in Appendix C; the union rules are reproduced in Figure 2.3 for the reader's convenience. These rules may seem incomplete since they lack, among other things, the equality rules for the various term constructors. For the sake of brevity, we adopt the convention that all of the equality rules from J are incorporated into R in the manner described above. For instance, the union introduction rule of J appears in R in the form:

$$
\begin{array}{ll}
H \gg \mathsf{inl}(a) = \mathsf{inl}(a') \ \mathsf{in}\ A + B \mathbin{@} k \ \mathit{ext}\ \mathsf{axiom} & \text{by intro} \\
\quad H \gg a = a' \ \mathsf{in}\ A \mathbin{@} k & \\
\quad H \gg B \ \mathsf{in}\ \mathsf{U}_k \mathbin{@} l &
\end{array}
$$

$$H \gg \mathsf{U}_k @ l \ \mathit{ext}\ A+B \quad [k<l] \quad \text{by formation}$$
$$\quad H \gg \mathsf{U}_k @ l \ \mathit{ext}\ A$$
$$\quad H \gg \mathsf{U}_k @ l \ \mathit{ext}\ B$$

$$H \gg A+B @ k \ \mathit{ext}\ \mathsf{inl}(a) \quad \text{by intro left}$$
$$\quad H \gg A @ k \ \mathit{ext}\ a$$
$$\quad H \gg B \ \mathsf{in}\ \mathsf{U}_k @ l$$

$$H \gg T @ k \ \mathit{ext}\ \mathsf{decide}(z; u.s; v.t) \quad \text{by elim on } z$$
$$\quad H, u{:}\, A @ i \gg T(\mathsf{inl}(u))_z @ k \ \mathit{ext}\ s$$
$$\quad H, v{:}\, B @ i \gg T(\mathsf{inr}(v))_z @ k \ \mathit{ext}\ t$$
$$\quad H, z{:}\, A+B @ i \gg T \ \mathsf{in}\ \mathsf{U}_k @ l$$
$$\mathit{where}\ z{:}\, A+B @ i \ \mathit{occurs\ in}\ H.$$

Figure 2.3: Selected R Union Rules

Notice that the rule shows a certain equality type to be inhabited, but this is equivalent to the equality judgement by the semantics of equality types.

To see how extraction is formalized, consider first the simple case of the union introduction rule. In refinement form this rule becomes:

$$H \gg A+B @ k \ \mathit{ext}\ \mathit{inl}(a) \quad \text{by intro left}$$
$$\quad H \gg A @ k \ \mathit{ext}\ a$$
$$\quad H \gg B \ \mathsf{in}\ \mathsf{U}_k @ l$$

The parameter "left" indicates that we will show A to be true; "right" would be used to show B inhabited. By the meaning assigned to sequents the first subgoal establishes $H \gg a \in A$. The second subgoal is present to ensure that $A+B$ is a type of level k, serving the same purpose as in J. Notice that by convention extract clauses are omitted from subgoals when they are not relevant to the determination of the extract clause of the goal; they are also omitted from goals when the extracted object is **axiom**.

The careful reader will notice that the union introduction rule still fails to have constructible subgoals — where does the level tag l in the second subgoal come from? One solution is to provide a means of specifying the tag to use as part of the rule invocation. Another is to allow the system to generate a new identifier, keeping track of the constraints which are imposed by rules with notations like $[k < l]$ attached, and ensuring that these constraints are always met. We will tacitly assume that one or another of these solutions has been adopted and will not treat the issue any further.

For an example of the complications which arise, consider the dependent product introduction rule.[2] In J, the rule is

$$
\begin{array}{l}
H \gg \langle a,b\rangle \in x{:}\,A \times B\,@\,k \quad \text{by intro} \\
\quad H \gg a \in A\,@\,k \\
\quad H \gg b \in B(a)_x
\end{array}
$$

As it stands, this rule has constructible subgoals since the pair $\langle a,b\rangle$ is available in the goal. But if we translate this into "extract" form, we get

$$
\begin{array}{l}
H \gg x{:}\,A \times B\,@\,k\ \mathit{ext}\ \langle a,b\rangle \quad \text{by intro} \\
\quad H \gg A\,@\,k\ \mathit{ext}\ a \\
\quad H \gg B(a)_x\,@\,k\ \mathit{ext}\ b
\end{array}
$$

Notice that the second subgoal cannot be instantiated without knowing a. Naïvely one would think that the solution is to simply make a be a parameter of the rule so that the rule becomes

$$
\begin{array}{l}
H \gg x{:}\,A \times B\,@\,k\ \mathit{ext}\ \langle a,b\rangle \quad \text{by intro using } a \\
\quad H \gg a \in A\,@\,k\ \mathit{ext}\ \mathsf{axiom} \\
\quad H \gg B(a)_x\,@\,k\ \mathit{ext}\ b
\end{array}
$$

But since A can be an arbitrary type and since type membership is undecidable in the general case, there is no way to ensure that the parameter a is correct except by proving the first subgoal.

Besides being inconvenient, taking this approach leads to a violation of the principle that each refinement step ought not have its correctness depend on further independent development of the derivation tree. Suppose that the a given as parameter is in fact not of type A. Then this refinement step will be *invalidated* when the derivation of the first subgoal fails to

[2] This terminology is unfortunate; it is best to think of this type as an infinitary disjoint union.

be completed. The user, thinking that a is "clearly" of type A may omit working on this step until the very end, only to discover that the entire proof tree from that point down is incorrect when he finally turns his attention to this subproof!

Notice that in the case of λ-PRL, no such problem arises. The reason is directly attributable to the simplicity of that logic. The dependent product introduction rule corresponds to the introduction rule for the existential quantifier. But in λ-PRL, A can only be N or List, and, even in the presence of bindings in the antecedent, membership for these types is quite easily decidable. Therefore, insisting that a be a parameter to the rule is no inconvenience and leads to no difficulties.

One solution to this problem for ν-PRL is to expand RED so as to allow derivations in rule invocations. That is, rules can take derivation as parameters and are not complete until that derivation has been completed. In the case of product introduction, the parameter would be the first subgoal above. This has the effect of forcing the user to complete the derivation of that subgoal before continuing to the second. This is somewhat inconvenient. The final solution appears to be to relax the definition of the refinement process so as to allow derivations to "pass through" inconsistent states. I shall have more to say about this in Section 2.5.

In the presentation of the rules in Appendix C, those rules which take a term as parameter have the property that there is a subgoal whose role is to ensure that the parameter term is of the right type. For instance, the product introduction rule takes a as parameter; the first subgoal ensures that $a \in A$ in H. In the prototype implementation of ν-PRL the user is required to provide a when the refinement step is taken; in the full implementation the first subgoal will itself be the parameter of the rule.

Extraction in ν-PRL works, from the point of view of the user, in much the same way as it does in λ-PRL except that the extracted object for *any* derivation is available for evaluation, not just when the goal is in $\forall\exists$ form. The evaluator is an implementation of the computation rules for each of the term constructors, and so any term of the logic may be evaluated using these rules.

One more aspect of R remains to be discussed, namely the elimination rules. Consider the union elimination rule. The formulation of this rule in J (Figure 2.2) is quite unwieldy from the point of view of refinement since both the term e which we are "eliminating" and the range type T_z must be taken as parameters (we cannot recover T from $T(e)_z$!) The discussion of product introduction above demonstrates the reason why we would like as far as possible to avoid this. The solution is inspired by λ-PRL.

The elimination rules in λ-PRL are formulated to "work on the left." For example, here is an instance of the ∨-elim rule:

$$\begin{array}{l} A \vee B \gg C \quad \text{by elim} \\ \quad A \gg C \\ \quad B \gg C \end{array}$$

By using a similar technique in ν-PRL we can achieve constructibility of subgoals without cumbersome rule parameters. Let the binding $z\colon A + B \mathbin{@} i$ occur in H. Then the union elimination rule is:

$$\begin{array}{l} H \gg T \mathbin{@} k \ \mathit{ext}\ \mathsf{decide}(z; u.s; v.t) \quad \text{by elim on } z \\ \quad H, u\colon A \mathbin{@} i \gg T(\mathsf{inl}(u))_z \mathbin{@} k \ \mathit{ext}\ s \\ \quad H, v\colon B \mathbin{@} i \gg T(\mathsf{inr}(v))_z \mathbin{@} k \ \mathit{ext}\ t \\ \quad H, z\colon A + B \mathbin{@} i \gg T \text{ in } \mathsf{U}_k \mathbin{@} l \end{array}$$

Notice that this rule has the constructible subgoals property; both z and T can be directly obtained from the goal.

There is no loss of generality in taking the elimination rule to have this form. To eliminate on an arbitrary term e of some union type, simply use the rule of substitution to replace z by e. Also recall that by the convention of incorporating J into R via equality types the seemingly more general elimination rule of J is available. It must, however, take e and T as parameters.

Recall that a central concern in the construction of R is to ensure that derivations can be constructed using only locally valid refinement steps. Since R was constructed with this in mind, it is primarily a technical matter to verify that R is amenable to refinement.

Theorem 2.3 *The logic* R *has the constructible subgoals property.*

Proof By inspection of each of the rules in Appendix C. ∎

The semantics of R in terms of J is given in Theorem 2.4. As with J, derivations in R are understood to be valid provided that the hypothesis list is well-formed in the following sense:

Definition 2.2 *A hypothesis list H is* well-formed *iff for each binding $x{:}T @ i$ in H,* $\mathsf{J}\ \vdash\ H_x \gg T \in \mathsf{U}_i @ i+1$. *In particular, the empty hypothesis list is always well-formed.*

The term "well-formed" is being used here in a sense closely related to, but not identical with, that of Section 2.3. No confusion should arise since we shall not have occasion to use both senses in the same context. As with J, the derivability relation for R is restricted to those sequents whose antecedents are well-formed.

The main result of this section is the statement of the semantics of R in terms of J.

Theorem 2.4 *If* $\mathsf{R}\ \vdash\ H \gg T @ k$ *ext* t, *then* $\mathsf{J}\ \vdash\ H \gg t \in T @ k$.

As immediate corollaries we have:

Corollary 2.3 R *is relatively consistent with* J, *and therefore with* M.

Corollary 2.4 *The inhabitation rule for* R *is valid.*

Proof of Theorem 2.4. The union rules given in Figure 2.3 illustrate all of the considerations that figure in the proof of this theorem. Refer to Figure 2.2 for the corresponding rules in J and for the equality rules of R. The proof proceeds by induction on derivations with a case analysis on the last step. In order to apply the induction hypothesis, the well-formedness (in the sense of Definition 2.2) of the antecedent must be established. Then a short derivation in J is built to establish the result. In each such J derivation, subgoals labelled "by I.H. i" are obtained by application of the induction hypothesis to the ith subgoal of the rule instance in question.

- (Hypothesis) Let the binding $x{:}T @ i$ occur in H. Then $\mathsf{J}\ \vdash\ H \gg x \in T @ i$ follows directly by the hypothesis rule applied to x.

- (Union formation) By the induction hypothesis we obtain

  $$\mathsf{J}\ \vdash\ H \gg A \in \mathsf{U}_k @ l$$

  and

  $$\mathsf{J}\ \vdash\ H \gg B \in \mathsf{U}_k @ l$$

  The result follows immediately by an application of the union formation rule.

- (Union introduction) By the induction hypothesis we have

$$\mathsf{J} \vdash H \gg a \in A \,@\, k$$

and

$$\mathsf{J} \vdash H \gg \mathsf{axiom} \in \mathsf{eq}(B, B, \mathsf{U}_k) \,@\, l$$

From the latter it follows that

$$\mathsf{J} \vdash H \gg B \in \mathsf{U}_k \,@\, l$$

and so the result follows by union introduction.

- (Union introduction, equality form) The following derivation in J gives the desired result:

$H \gg \mathsf{axiom} \in \mathsf{eq}(\mathsf{inl}(a), \mathsf{inl}(a'), A + B) \,@\, k$ by intro
  $H \gg \mathsf{inl}(a) = \mathsf{inl}(a') \in A + B \,@\, k$ by intro
    $H \gg a = a' \in A \,@\, k$ by elim
      $H \gg \mathsf{axiom} \in \mathsf{eq}(a, a', A) \,@\, k$ by I.H. 1
    $H \gg B \in \mathsf{U}_k \,@\, l$ by elim
      $H \gg \mathsf{axiom} \in \mathsf{eq}(B, B, \mathsf{U}_k) \,@\, l$ by I.H. 2

- (Union elimination) First, note that the antecedents of the subgoals are well-formed because of the assumption that H is well-formed, the fact that $z{:}\,@\, A + B \,@\, i$ occurs in H, and, in the case of subgoals 1 and 2, the type structure rules in J for the union type. Therefore we can freely apply the induction hypothesis to the subgoals. The following derivation establishes the result:

$H \gg \mathsf{decide}(z; u.s; v.t) \in T \,@\, k$ by elimination
  $H \gg z \in A + B \,@\, i$ by hypothesis
  $H, u{:}\, A \,@\, i \gg s \in T(\mathsf{inl}(u))_z \,@\, k$ by I.H. 1
  $H, v{:}\, B \,@\, i \gg t \in T(\mathsf{inr}(v))_z \,@\, k$ by I.H. 2
  $H, z{:}\, A + B \,@\, i \gg T \in \mathsf{U}_k \,@\, l$ by elim
    $H, z{:}\, A + B \,@\, i \gg \mathsf{axiom} \in \mathsf{eq}(T, T, \mathsf{U}_k) \,@\, l$ by I.H. 3

- (Explicit introduction) By the induction hypothesis,

$$\mathsf{J} \vdash H \gg \mathsf{axiom} \in \mathsf{eq}(a, a, A) \, @ \, k$$

and therefore

$$\mathsf{J} \vdash H \gg a \in A \, @ \, k$$

by equality elimination.

- (Inhabitation) By the induction hypothesis and the inhabitation rule in J.

This completes the proof. ∎

2.5 Concluding remarks

The foregoing development of the ν-PRL logic was primarily concerned with structuring the rules in such a way that they would be suitable as a basis for a refinement–style editor. This constraint was achieved, by and large, by careful consideration of the issue of well–formedness of goals, and by exploiting the semantics of the equality type. This effort has turned out to be successful from the point of view of building a prototype ν-PRL system on the λ-PRL basis. But for the long run, I think that the approach to refinement which works so well for λ-PRL must be re–thought for more complex logics such as type theory.

The main difficulty is that "refinement" ought to designate a property of a proof editor rather than a property of a set of inference rules. The PRL acronym is therefore something of a misnomer. The notion of refinement ought to be focussed at a level above that of the individual rules of the logic, and more on the user's breakdown of the problem at hand. There is no reason to believe that a rule–level notion of refinement corresponds to a problem–level notion of refinement; in fact there are reasons to believe that this is not the case.

Experience with λ-PRL and ν-PRL has shown that there is some overhead involved in conducting a derivation which is not directly related to the problem at hand. Proof obligations related to well–formedness of types are good examples of this. It is often desirable to be able to choose the order of development of the proof tree in accordance with demands other than the need for local consistency of a proof tree. Enforcement of such ordering places an undue burden on the user, distracting him from the "real" issues.

I think that I was misled into using the term "refinement logic" because in the simple case of λ-PRL there are much fewer distracting proof obligations than there are in PRL, and so proof construction parallels problem solution to a much greater extent. I was also misled by the use of tableaux in predicate calculus and arithmetic to prescribe a proof search strategy which will often work. It appeared to be possible to do for type theory what tableaux did for arithmetic. However, the considerations discussed above show that building a refinement logic (in the sense discussed above) for type theory is more of a shoe–horning than an elegant structuring of the rules.

The solution lies in a careful reconsideration of the notion of refinement as is relates to the design of a proof editor. This is a non–trivial task, but one which, I think, is amenable to a well–defined solution. A full–scale proof editor ought to allow, at least, for proof trees to pass through inconsistent states during construction. This ability is one of the most intriguing aspects of the Hoare logic system described in [Reps & Alpern 84]. But in addition it seems desirable to have the ability for automatic readjustment of a proof tree so that it conforms to some sort of consistency constraints. For instance, if the goal type of a sequent is $A + B$, and the node has been built by "intro", then changing $A + B$ to, say, $C + D$ might imply, at the user's discretion, that occurrents of A in the subgoals be changed to C. In fact, one can subsume extraction into this framework by insisting that it be invariant that the extracted code for a goal relate in a way prescribed by the rule instance to the extracted code for the subgoals. Extraction then becomes just another "clean–up" operation on proof trees!

There are many exciting possibilies in the design of such an editor, but, alas, treatment of this area lies beyond the scope of this work.

Chapter 3

A Theory of Typed Terms

The subject of this chapter is a formalization of type theory as a calculus of *typed terms*. Roughly speaking, typed terms are terms that are "decorated" with enough information to encode a derivation of membership of that term in a type (and, in fact, of that type expression in a universe type). The primary interest in typed terms is that they will prove useful as a basis for decision methods for the membership and equality relation of type theory, as will be seen in Chapters 4 and 5. Many of the techniques explored here will also be useful in the context of Allen's *term logics* [Allen 85] which allow for user-specified annotation of terms with type expressions.

The idea of associating a type expression directly with a term is familiar from the typed λ-calculus. In that system each term has associated with it a unique type expression, usually written as a superscript, which is *the* type of that term. Each term determines a unique type because the constants and all (free and bound) variable occurrences come with a type expression; this information is sufficient to determine the type of each term. For instance the term $\lambda x{:}\tau.x$ has type $\tau \rightarrow \tau$ where τ is any legal expression in the simple theory of types.

When viewing typed λ-terms as arising from untyped terms by assigning type expressions which describe the computational behavior of the untyped term, the property of possessing a *unique* type expression lapses. However, it is replaced by the closely-related notion of a "principal" or "most-general" type *scheme* which describes the structural similarity that all types of a given untyped term must share [Hindley 69,Milner 78]. For instance, all correct typings of $\lambda x.x$ are of the form $\alpha \rightarrow \alpha$, where α is a scheme variable ranging over the set of type expressions.

The property of possessing a principal type scheme is directly related to the simplicity of the language of types. The sorts of properties of functions that can be expressed in the type system of the typed λ-calculus are limited to a rather coarse–grained categorization according to the input/output behavior of the function. If the type system is sufficiently strong, however, much more detailed descriptions of a function can be made, ranging from a very simple type assignment to something approaching a complete "specification" of its characteristics. In a type system which is strong enough to encode ordinary logic it is clear that no particular structural relationship need hold between any two typings of a function.

Intuitionistic type theory is such a system. Not only is there no principal type of complex λ-terms, due to the range of things which we might care to say about them, but also there is no reason to expect, even for the atomic terms of the theory, that they will possess any particular type or collection of related types. For example, 0 might occur as a member of the type of natural numbers, as an element of a Pascal–style enumeration type, as a real number, as a universe index, or, in fact, as a member of any number of types which may be defined. The determination of which types a given expression may inhabit is strictly a matter of the semantics of the various types which comprise a given formulation of type theory.

Therefore, in contrast to the case of the λ-calculus, the annotation of a term with type expressions does not serve to define *the* type of a term, but rather *a* type of the term. The calculus of typed terms presented below encodes a certain sort of derivation which, it is hoped, will prove to be an effective means of providing automatic assistance with proofs of membership and equality. The type assigned to a term in this calculus will always be *semantically* correct, but it will not always be will be what anyone had in mind when the term was written down. For this reason, the process of annotating untyped terms with types will be a matter of informed guesswork aimed at discovering typings which are useful in practice.

Notice that in the case of the typed λ-calculus, the annotation of a term is with expressions drawn from a separate, context–free class of type expressions. No such treatment is possible in the case of the PRL type theory: types are denoted by any expression which can be correctly judged to inhabit a universe type. This is not a context–free condition, nor is it, in general, decidable. Furthermore, the presence of the dependent function space constructor (among others) blurs the customary distinction between "types" and "objects" by indexing the range type of a function by the domain object.

Therefore, the annotations themselves are not subject to a separate treatment and so must be accounted for in the logic of typed terms. This is handled in the system presented below by requiring that the type expressions used to annotate a term are themselves required to be typed terms, albeit ones whose annotation is a universe type. There is no infinite regress implied by this move because universe types are taken to denote types without justification by an annotation.

Beginning in the next section we define a logic of typed terms which is based on the ideas outlined above. The annotation chosen is motivated by the structural similarity shared by the introduction and equality rules for each of the term constructors and by the desire to define a decidable sub-relation of the full membership relation of type theory. This amounts to limiting attention to only certain sorts of membership derivations. The entire theory is parameterized by an arbitrary (decidable) collection of basis typings, called "atoms." This basis covers such typings as $0 \in \mathsf{nat}$ which, while semantically sound, may not be "right" for the context. The selection of basis typings is largely heuristic, tuned by experience with the decision methods. Further discussion of this issue is deferred to the next chapter where the process of obtaining a well-formed typed counterpart of an untyped term is considered.

3.1 Syntax of typed terms

The treatment of typed terms is parameterized by the collection of term constructors of the underlying language of type theory. In the following grammar, o ranges over the set of term constructors, x ranges over the set of variables, and t over the set of untyped terms.

$$
\begin{array}{rcll}
\tau & ::= & \mathsf{U}_i & \text{universe types} \\
 & | & x{\uparrow}\tau & \text{variables} \\
 & | & t{\uparrow}\tau & \text{atoms} \\
 & | & x{:}\,\tau.\tau & \text{abstractions} \\
 & | & o(\tau,\ldots,\tau){\uparrow}\tau & \text{compound terms}
\end{array}
$$

The syntactic variables μ, ν, ρ, σ, and τ range over the class TT of typed terms. In a linear presentation of a typed term, $\uparrow$'s associate to the right; parentheses are used to indicate otherwise. So $x{\uparrow}\mathsf{U}_1{\uparrow}\mathsf{U}_2 \equiv x{\uparrow}(\mathsf{U}_1{\uparrow}\mathsf{U}_2)$.

The "type field" of a typed term τ, TF$[\tau]$, or simply $\uparrow\tau$, is defined as follows:

$$
\begin{array}{l}
\textbf{if } \tau \equiv \mathsf{U}_i \;\rightarrow\; \mathsf{U}_{i+1} \\
\Box\; \tau \equiv x{\uparrow}\sigma \;\rightarrow\; \sigma \\
\Box\; \tau \equiv t{\uparrow}\sigma \;\rightarrow\; \sigma \\
\Box\; \tau \equiv x{:}\,\rho.\sigma \;\rightarrow\; \rho \\
\Box\; \tau \equiv o(\sigma_1,\ldots,\sigma_n){\uparrow}\sigma \;\rightarrow\; \sigma \\
\textbf{fi}
\end{array}
$$

The "erasure" of a typed term τ, Erase$[\tau]$, or τ°, is an untyped term obtained by eliminating the type fields. It is defined as follows:

$$
\begin{array}{l}
\textbf{if } \tau \equiv \mathsf{U}_i \;\rightarrow\; \mathsf{U}_i \\
\Box\; \tau \equiv x{\uparrow}\sigma \;\rightarrow\; x \\
\Box\; \tau \equiv t{\uparrow}\sigma \;\rightarrow\; t \\
\Box\; \tau \equiv x{:}\,\rho.\sigma \;\rightarrow\; x.\sigma^\circ \\
\Box\; \tau \equiv o(\sigma_1,\ldots,\sigma_n){\uparrow}\sigma \;\rightarrow\; o(\sigma_1^\circ,\ldots,\sigma_n^\circ){\uparrow}\sigma^\circ \\
\textbf{fi}
\end{array}
$$

Typed terms are defined in such a way that the erasure of a typed term is a syntactically well-formed untyped term, reflecting the idea that typed terms are an "elaboration" on untyped terms.

The ordinary notions of free and bound occurrences of variables and the relation of identity modulo renaming of bound variables, $\equiv_\alpha$, are defined in the obvious way, given that the only variable-binding construct is the abstraction. Ordinary capture-avoiding substitution of σ for free occurrences of x in τ, provided that TF$[\sigma] \equiv$ TF$[x]$, is written $\tau[\sigma/x]$ (each occurrence of $x \uparrow \sigma$ is replaced by τ). As a special case the operation of replacing a free occurrence of y by x is written $\tau[x/y]$ (the type superscript on x is omitted).

Typed terms are subject to some non-context-free syntactic constraints. First, the type field of any term must be a non-abstraction whose type field is a universe. Second, all occurrences of a variable must have the same type field (modulo renaming of bound variables); in particular, all occurrences of x within the scope of an abstraction must have a type field which agrees with the type expression at the binding occurrence. Thus each occurrence of x in the body, τ of an abstraction $x{:}\sigma.\tau$ must have type field σ.

The grammar of typed terms is abstract with respect to a decidable set of term constructors which includes, at least, the universe types as nullary constructors. It is assumed that each constructor, o, carries with it a set of syntactic constraints that determine the number of component terms and which, if any, are abstractions. For components that are abstractions, o must specify the arity of the abstraction as well. In fact, abstractions are restricted to occur only as determined by some constructor o, and so are not considered to be "first-class" terms. As a technical convenience, we consider only unary abstractions, that is, those which bind exactly one variable. Abstractions of higer arity can be obtained by nesting unary abstractions.

The class of typed terms is also parameterized by a decidable set $\mathcal{A}$ of *atoms* built by attaching a typed term to an untyped term (think of $t \uparrow \tau$ as an abbreviation for $\mathsf{atom}(t, \tau)$ where **atom** is a special term constructor). The word "atom" is meant to suggest that the untyped component t is considered to be unanalyzable. The atom $t \uparrow \tau$ is a syntactically legal typed term only if it is a member of $\mathcal{A}$.

To make this more concrete, let us consider an example of a language of typed terms. The term constructors are **apply**, λ, Π, and **nat**; the atom set, $\mathcal{A}$, is empty. The constructor **apply** takes two arguments, neither of which is an abstraction; λ and Π each take a unary abstraction as argument; **nat** takes no arguments. Then the following are all syntactically correct typed terms:

$$x \uparrow \mathsf{U}_1$$
$$x \uparrow \mathsf{U}_1 \uparrow \mathsf{U}_2$$
$$\lambda(x{:}\, \mathsf{U}_1.x \uparrow \mathsf{U}_1) \uparrow \Pi(x{:}\, \mathsf{U}_1.\mathsf{U}_1) \uparrow \mathsf{U}_2$$
$$\mathsf{apply}(\lambda(x{:}\, \mathsf{U}_1.x \uparrow \mathsf{U}_1) \uparrow \Pi(x{:}\, \mathsf{U}_1.\mathsf{U}_1) \uparrow \mathsf{U}_2; \mathsf{nat} \uparrow \mathsf{U}_1) \uparrow \mathsf{U}_1$$

The erasure of the last of these terms is:

$$\mathsf{apply}(\lambda(x.x);\mathsf{nat}).$$

This example raises a minor technical point about the erasure of abstractions: $\text{Erase}[\Pi(x{:}\,\sigma.\tau)] \equiv \Pi(x.t)$ where $t \equiv \text{Erase}[\tau]$. Assuming that Π is the dependent product type constructor, this is not quite what one would like — the domain expression of the abstraction ought not be deleted, just erased: $\Pi(x{:}\,\sigma.\tau)^{\circ}$ should be $\Pi(x{:}\,\sigma^{\circ}.\tau^{\circ})$. This difficulty may be circumvented by any number of methods. One is to make Erase be sensitive to the context in which an abstraction occurs. Another is to use a different abstraction operator with terms like Π whose erasure preserves the domain expression of the abstraction in its erased form. Since the treatment of typed terms is abstract with respect to the particular term constructors, such details will not be considered. It is assumed that some convention is adopted so that the erasure of a typed term is a syntactically well-formed untyped term.

A *binding* is a pair $x{:}\,\tau$ such that $\uparrow\tau$ is a universe type. A *basis*, B, is a finite sequence of bindings such that no variable may occur on the left of more than one binding. Given a variable x bound in B, let B_x denote the prefix of B up to, but not including, the binding for x. We say that B *covers* the typed term σ iff each free variable $x\uparrow\tau$ in σ there is a binding $x{:}\,\tau$ in B. Bases are further restricted so that B_x covers σ for each binding $x{:}\,\sigma$ in B. The extension of B by the binding $x{:}\,\sigma$ is written $B, x{:}\,\sigma$; it is tacitly assumed that the resulting basis adheres to the foregoing restrictions. For an untyped term t, the notation B_t stands for the longest prefix of B in which the atom t does not occur. It is assumed that B_t covers σ for each $t\uparrow\sigma$ in the atom set $\mathcal{A}$. Capture–avoiding substitution is extended to bases term–by–term.

Bases are used to record the types of variables when descending into binding contexts. Just as with bound variables, no significance is assigned to the names of the variables in a basis, only to their position in the sequence. Bases B and B' which differ only in the names of the bound variables are identified; the relation $B \equiv_{\alpha} B'$ is used for this relation. Terms are most often considered in the context of some basis. The notation $\langle B, \sigma\rangle$ is used to denote the basis–term pair consisting of basis B and typed term σ. The relation $\langle B, \sigma\rangle \equiv_{\alpha} \langle B', \tau\rangle$ holds exactly when $B, x{:}\,\sigma \equiv_{\alpha} B', x{:}\,\tau$

holds. The idea is that neither the names of the bound variables in the two terms σ and σ', nor the names of the free variables referring into their respective bases matter. For instance (taking liberties with the syntax so as not to obscure the point),

$$\langle(x{:}\,\tau, z{:}\,\sigma), o(x; u.u)\rangle \equiv_\alpha \langle(y{:}\,\tau; w{:}\,\rho), o(y; v.v)\rangle$$

provided that $\rho \equiv_\alpha \sigma[y/x]$.

To make some of the technical details of the material presented below more palatable, we shall assume that bases are in *standard form*, meaning that the variables are named $v_1,\ldots,v_k$, from left–to–right. Identity of standarized bases identity, and therefore identity of basis–term pairs, reduces to term–by–term identity modulo α-conversion. For example, the following two bases are considered to be identical:

$$\begin{array}{l} x{:}\,\mathsf{U}_1, y{:}\,x\!\uparrow\!\mathsf{U}_1, z{:}\,\Pi(w{:}\,\mathsf{nat}\!\uparrow\!\mathsf{U}_1.\mathsf{nat}\!\uparrow\!\mathsf{U}_1)\!\uparrow\!\mathsf{U}_1 \\ u{:}\,\mathsf{U}_1, v{:}\,u\!\uparrow\!\mathsf{U}_1, w{:}\,\Pi(z{:}\,\mathsf{nat}\!\uparrow\!\mathsf{U}_1.\mathsf{nat}\!\uparrow\!\mathsf{U}_1)\!\uparrow\!\mathsf{U}_1 \end{array}$$

The standard form of the first of these is:

$$v_1{:}\,\mathsf{U}_1, v_2{:}\,v_1\!\uparrow\!\mathsf{U}_1, v_3{:}\,\Pi(w{:}\,\mathsf{nat}\!\uparrow\!\mathsf{U}_1.\mathsf{nat}\!\uparrow\!\mathsf{U}_1)\!\uparrow\!\mathsf{U}_1$$

The relation $B \sqsubseteq B'$ means that B is a prefix of B', ignoring the names of the bound variables. In other words, the standard form of B is a prefix of the standard form of B', considering individual terms modulo α-conversion. Notice that if $B \sqsubseteq B'$ and $B' \sqsubseteq B$, then $B \equiv_\alpha B'$. The motivation behind this definition of approximation will be discussed in Section 3.3.

3.2 A logic of typed terms

With the syntactic conventions out of the way, we can proceed to define the *well-formedness* of a typed term τ with respect to a basis B, written $B \rhd \tau$. This relation is defined by the following rules:

- $B \rhd \mathsf{U}_i$;
- $B \rhd x\!\uparrow\!\tau$ if $x{:}\,\tau$ is in B and $B_x \rhd \tau$;
- $B \rhd t\!\uparrow\!\tau$ if $t\!\uparrow\!\tau$ is in $\mathcal{A}$ and $B_t \rhd \tau$;

- $B \rhd x{:}\sigma.\tau$ if $B \rhd \sigma$ and $B, v_k{:}\sigma \rhd \tau[v_k/x]$ where k is one more than the length of B;

- $B \rhd o(\tau_1, \ldots, \tau_{n-1})\uparrow\tau_n$ if $B \rhd \tau_i$ for each $1 \leq i \leq n$ and the τ_i match up by o–introduction (see below for a discussion of this condition).

The premise of the well–formedness rules for variables and atoms can be dropped if we assume that B is a "legitimate" basis in the following sense:

Definition 3.1 *A basis B is orderly iff (1) for each $x{:}\sigma$ in B, $B_x \rhd \sigma$; and (2) for each atom $t\uparrow\sigma$ in $\mathcal{A}$, $B_t \rhd \sigma$.*

This idea is analagous to the assumption in the R logic that the antecedents of sequents are well–formed. Notice that the rule for well–formedness of abstractions ensures that the orderliness of B is preserved. Hereafter, all bases are taken to be orderly.

The condition on the rule for terms built by the constructor o needs explanation. The idea is that the introduction rule for terms constructed by o defines a relationship which must hold between the component terms, their respective types, and the type of the term. Taking o to be λ, the above rule states that in order for $\lambda(x{:}\,\rho.\sigma)\uparrow\tau$ to be well–formed, it must be the case that $\tau \equiv \Pi(x{:}\,\rho.\uparrow\sigma)$,[1] exactly as the λ introduction rule requires. Similarly, in order for $\mathsf{inl}(\sigma)\uparrow\tau$ to be well–formed, τ must be a union type whose left component is the type field of σ, *i.e.*, $\tau \equiv \uparrow\sigma + \rho$ for some ρ. In order for $(\rho + \rho')\uparrow\tau$ to be well–formed, it must be that $\tau \equiv \mathsf{U}_i$ for some i and that $\uparrow\rho \equiv \uparrow\rho' \equiv \tau$.

The **decide** constructor illustrates an important point about the structure of the annotation. Unlike the other terms considered so far, the typed counterpart of a **decide** will contain an additional component, called an *auxiliary field*, which does not appear in the untyped form. This field must, then, be removed by the erasure operation for the term. The **decide** introduction rule (*i.e.*, the union elimination rule) takes a parameter defining the range type; therefore, in order to encode an "intro" derivation this parameter must be carried in an auxiliary field. To see how this works,

[1] We will omit the type field of a term whenever it is not relevant to the context. It is assumed that such terms are well–formed.

consider the term:

$$\mathsf{decide}(\nu; u{:}\,\rho.\rho'; v{:}\,\sigma.\sigma'; z{:}\,\mu.\mu')\uparrow\tau$$

(where the third component is the encoding of the parameter $z.T$ of the union elimination rule). Taking o to be **decide**, and keeping the union elimination rule in mind, According to the rules of well-formedness, this term is well-formed if each of its components is well-formed and

1. $\uparrow\nu \equiv \rho + \sigma$: the principal argument must be some union type;
2. $\mu \equiv \uparrow\nu$: z ranges over that union type;
3. $\uparrow\rho' \equiv \mu'[\mathsf{inl}(u)/z]$: the requirement for the "if left" part as defined by the semantics of type theory;
4. $\uparrow\sigma' \equiv \mu'[\mathsf{inr}(v)/z]$: the "if right" part;
5. $\tau \equiv \mu'[\nu/z]$: the result type must be the instance of the range type μ' with the principal argument "plugged in."

This definition may seem daunting, but remember that it is not intended to be used by a person but rather as a component of a decision procedure. What we are doing here is spelling out in bitter detail the rules for checking the validity of an application of **decide** introduction, given the four purported subderivations. Incidentally, this is the sort of validity check that must be done by a refinement editor which supports internal modification of proof trees in order to ensure that a particular node of the tree is valid. In general, by the very nature of proof-checking, we expect (and insist) that only decidable matching conditions be associated with each constructor o. Given this, it is easy to see that the following theorem holds:

Theorem 3.1 *For any basis B and typed term τ, the relation $B \rhd \tau$ is decidable.*

Proof By induction on the definition of the well-formedness relation. ∎

The requirements given in equations (1) through (5) above are stronger than is strictly necessary. The semantics of type equality is such that equality (in some universe) can be used instead of strict term identity. The equality relation properly includes the identity relation and the congruence relation generated by the computation rules, and so the use of equality liberalizes the above restrictions considerably. Lacking a theory of equality for typed terms (beyond identity), we cannot state (1) through (5) in these terms. However, in Chapter refEqChap such a theory will be defined and the restrictions can be correspondingly relaxed.

Notice that the definition of $B \rhd \tau$ depends on B only to provide typings for the free variables in τ. Therefore, any other basis covering τ will do just as well.

Lemma 3.1 *If $B \rhd \tau$ and B' covers τ, then $B' \rhd \tau$.*

This lemma is useful to justify a speed-up of well-formedness checking: if we know that τ is well-formed in *some* basis, then it is well-formed in a given basis B provided that B covers τ.

It is immediately obvious from the definition of approximation that if $B \sqsubseteq B'$ and B covers some term τ, then B' covers τ as well. Therefore we have:

Lemma 3.2 *If $B \rhd \tau$ and $B \sqsubseteq B'$, then $B' \rhd \tau$.*

It is, perhaps, a bit inelegant to annotate terms as liberally as we have done here. It is obvious from the above examples that the annotations contain a substantial amount of redundant information. This is mitigated somewhat if we consider expressions to be directed acyclic graphs, rather than trees, since this representation allows for sharing of common components. Another approach is to define a *minimal* annotation that is sufficient to allow the reconstruction of the annotation defined above. The function TF must then *compute* the type field of a term by recursion on the structure of the term, rather than simply returning the "type field." The presentation below is independent of how TF is defined, and so either approach is acceptable.

3.3 Typed term schemes

In this section we discuss the extension of the system of typed terms to admit a new class of variables, the *type variables*. Type variables are scheme variables that range over the set of typed terms whose type field is some universe. The basic idea is that these variables are universally quantified, so replacement of a type variable by a suitable term will result in a well-formed typed term. The definition of "suitable replacement" is complicated by the fact that the type expressions are not a separable, decidable class of expressions, as is the case with the typed λ-calculus.

Since the property of denoting a type is context-dependent, a type variable must have associated with it a basis in which any replacement for that variable must be well-formed. In addition, a type variable must specify the universe level of acceptable replacements. The most straightforward way in which one might associate this information with a type variable is by a binding of the form $\alpha\colon \mathsf{U}_i$ in some basis B. Replacement would be handled by a substitution rule of the form:

- $B_1, B_2[\sigma/\alpha] \rhd \tau[\sigma/\alpha]$ if $B_1 \rhd \sigma$, $\uparrow\sigma \equiv \mathsf{U}_i$, and $B_1, \alpha\colon \mathsf{U}_i, B_2 \rhd \tau$.

This method is perfectly adequate from a theoretical point of view, but for the purposes of the work in Chapter 4, it is not suitable. If type variables are declared explicitly, then the only way to eliminate these declarations (other than by substitution) is to introduce a λ-binding for the variables. Since we shall be constructing an annotation algorithm which works by replacing type variables by type expressions, this method is undesirable. What we would like is for typed terms involving type variables to be typed term *schemes* in the sense that any suitable instance of such a scheme is a typed term. Instantation involves no structural modification of the term beyond the replacement of the type variable by some typed term.

Implicit declaration is accomplished by the association of a basis $B(\alpha)$ and a universe $U(\alpha)$ with each type variable α. A replacement σ for α must be well-formed in basis $B(\alpha)$ and must have type field $U(\alpha)$. The basis and universe associated with each type variable places certain constraints on the contexts in which that type variable may occur. For instance, consider the the term $\tau \equiv \lambda(x\colon \alpha.x\uparrow\alpha)\uparrow\Pi(x\colon \alpha.\alpha)$. This term cannot be well-formed in, say, the empty basis unless $B(\alpha)$ were also empty. For if $B(\alpha)$ were $y\colon \mathsf{U}_1$, and $U(\alpha) \equiv \mathsf{U}_1$, then an acceptable replacement for α, according to the above criteria, would be $y\uparrow\mathsf{U}_1$ (since $B(\alpha) \rhd y\uparrow\mathsf{U}_1$ and $\mathrm{TF}[y\uparrow\mathsf{U}_1] \equiv U(\alpha)$). But then the result of the replacement would be

$$\lambda(x\colon y\uparrow\mathsf{U}_1.x\uparrow y\uparrow\mathsf{U}_1)\uparrow(y\uparrow\mathsf{U}_1 \rightarrow y\uparrow\mathsf{U}_1)$$

which is malformed in the empty basis since y would have no binding. Thus the definition of well–formedness of a term with type variables must take into account the bases associated with those type variables in order to ensure that substitution preserves well–formedness.

To make all of this precise, we must first lay down some syntactic conventions. The class TTS of typed term schemes is an extension of the class of typed terms obtained by admitting elements of the class TV of type variables at the basis. The symbols α, β, and γ range over TV. Bases are extended so as to admit typed term schemes as components. The identity relations on terms, $\equiv$ and $\equiv_\alpha$, are extended to be reflexive on type variables. The erasure of a type variable is itself; the type field of α is defined to be $U(\alpha)$.

The well–formedness relation is extended by adding the rule:

- $B \triangleright \alpha$ if $B(\alpha) \sqsubseteq B$.

The resulting relation is also written $B \triangleright \tau$; the symbol $\triangleright_0$ is used to refer to the original (no type variables) formulation. This new well–formedness relation is clearly a conservative extension of $\triangleright_0$ in the sense that if neither B nor τ involve any type variables and $B \triangleright \tau$, then $B \triangleright_0 \tau$. Note also that $\triangleright$ remains decidable.

Recall that the approximation relation on bases was defined to be the prefix relation on sequences, assuming that bases are in standard form. The reason for this can now be explained by appeal to the relationship between expicit and implicit declaration of type variables. First, we introduce some terminology. Consider a derivation of $B \triangleright \tau$ and fix an occurrence of some variable x in τ. The basis B' which is "in effect" when x is reached in the derivation of well–formedness of τ is called the *basis of occurrence* of (that occurrence of) x. By the ordinary scope rules for binding operators, the basis of occurrence of a variable is an extension of the basis in effect when the variable was declared by adding its binding to the basis. For instance, consider a proof of

$$B \triangleright x{:}\sigma.o(x{\uparrow}\sigma; v{:}\rho.x{\uparrow}\sigma)$$

The first occurrence of $x{\uparrow}\sigma$ has basis of occurrence $B, x{:}\sigma$; the second has $B, x{:}\sigma, v{:}\rho$. Each extends B, the basis in effect at the time that x is declared.

The idea underlying the definition of the well–formedness relation given above is that an occurrence of α must lie within the scope of its implicit declaration. This scope is defined in terms of a basis, $B(\alpha)$. In order to lie within that scope, B must be an extension of $B(\alpha)$. This accords with the intuition that the basis of occurrence of a declared variable is an extension of the left–context of the declaration. Thus, an occurrence of α is legitimate only if the basis in which it occurs is an extension of the left–context of its implicit declaration, $B(\alpha)$.

To make this clearer, consider some examples. Let α and β be two type variables, with $U(\alpha) \equiv U(\beta) \equiv \mathsf{U}_1$, $B(\alpha) \equiv v_1{:}\,\sigma$, and $B(\beta) \equiv v_1{:}\,\tau$. Then the following typed term scheme is well–formed in the empty basis:

$$o(\lambda(v_1{:}\,\sigma.\alpha);\lambda(v_1{:}\,\tau.\beta))$$

whereas this term is not:

$$o(\lambda(v_1{:}\,\tau.\beta);\lambda(v_1{:}\,\tau.\beta))$$

as is easily verified by the rules of well–formedness. The names of the bound variables have been chosen so as to make the bases of occurrence of α and β agree with $B(\alpha)$ and $B(\beta)$. This is purely a matter of expository convenience; the conventions of Section 3.1 are defined so that there is no dependence on bound variable names.

There is, however, a subtlety which is easily overlooked. Namely, each of the terms appearing in a basis must lie properly within the scope of the type variables appearing in this term. For instance, we cannot allow the following situation: $B \equiv (x{:}\,\rho, y{:}\,\tau)$ where $\tau \equiv o(z{:}\,\sigma.\alpha))$ and $B(\alpha) \equiv (x{:}\,\rho, z{:}\,\sigma)$ because the "defining occurrence" of α lies *within* τ. In essence, there is an implicit λ binding for α which lies just before the occurrence of α. Therefore, this α is a *distinct* type variable for each occurrence of τ, and the basis associated with it changes relative to each occurrence. For instance, $B \rhd y{\uparrow}\tau$ but $B \not\rhd \tau$, since $B(\alpha) \not\sqsubseteq B, y{:}\,\tau$. But this does not accord with our intuition about the use of type variables — they are treated as universally quantified variables which can be freely replaced with (suitably constructed) typed terms. In the case of the occurrence of α within τ, the scope scope of α has not been entered (in fact, it has no fixed scope as the example illustrates), and so we have no license to replace it with a typed term.

The solution is to define the notion of a basis B *covering* a typed term scheme. For typed terms (without type variables), we insist, as before, that each free variable be declared appropriately in B. When type variables are introduced, this definition is extended to ensure that the ***implicit*** declaration of α lie within B. The precise definition is as follows:

Definition 3.2 *A basis B* covers *typed term scheme σ if and only if (1) for each free occurrence of $x{\uparrow}\tau$ in σ, there is a binding $x{:}\tau$ in B; and (2) for each α occurring in σ, the declaring basis $B(\alpha)$ is a prefix of B.*

As before, we stipulate that B_x cover σ for each binding $x{:}\sigma$ in B and that B_t cover σ for each $t{\uparrow}\sigma$ in $\mathcal{A}$. This is tantamount to insisting that the types of all variables must be fixed (with respect to the type variables) — the implicit declaration of each α must lie within the left–context of the declaration $x{:}\sigma$, namely B_x.

We write $B \mathrel{!\triangleright} \sigma$ to mean that B covers σ and $B \triangleright \sigma$. By the convention that B_x cover σ for each $x{:}\sigma$ in B, the well–formedness rule for variables and atoms become:

- $B \triangleright x{\uparrow}\sigma$ if $x{:}\sigma$ occurs in B and $B_x \mathrel{!\triangleright} \sigma$;
- $B \triangleright t{\uparrow}\sigma$ if $t{\uparrow}\sigma$ is in $\mathcal{A}$ and $B_t \mathrel{!\triangleright} \sigma$

The well–formedness rule for abstractions must preserve this property, hence it becomes:

- $B \triangleright x{:}\,\rho.\sigma$ if $B \mathrel{!\triangleright} \rho$ and $B, x{:}\,\rho \triangleright \sigma$.

Just as for the system of typed terms, the premise of the well–formedness rule for variables can be dropped by adopting the convention that bases be orderly in the following sense:

Definition 3.3 *A basis B is orderly if (1) for each $x{:}\sigma$ in B, $B \mathrel{!\triangleright} \sigma$; and (2) for each atom $t{\uparrow}\tau$ in $\mathcal{A}$, $B \mathrel{!\triangleright} \tau$.*

It is henceforth assumed that bases are orderly in this sense.

It is a trivial, but important, fact that covering is preserved by extension:

Lemma 3.3 *If B covers σ and $B \sqsubseteq B'$, then B' covers σ.*

We then have the following crucial lemma:

Lemma 3.4 *If $B \mathrel{!\triangleright} \tau$ and B' covers τ, then $B' \triangleright \tau$ (and hence $B \mathrel{!\triangleright} \tau$).*

Proof By induction on derivations.

- $\tau \equiv \mathsf{U}_i$: Immediate.
- $\tau \equiv \alpha$: Since B' covers τ, $B(\alpha) \sqsubseteq B$, so $B(\alpha) \sqsubseteq B'$, and hence $B' \triangleright \tau$.
- $\tau \equiv x{\uparrow}\sigma$: Since $B \sqsubseteq B'$, the binding $x{:}\,\sigma$ appears in B' as well. By the convention on bases, B'_x covers σ, so it follows by induction that $B'_x \triangleright \sigma$ and therefore $B' \triangleright x{\uparrow}\sigma$.
- $\tau \equiv t{\uparrow}\sigma$: Similar to the previous case.
- $\tau \equiv x{:}\,\rho.\sigma$: By the induction hypothesis applied to the first premise of the well–formedness rule for abstractions we obtain $B' \mathrel{!\triangleright} \rho$. Now since B' covers $x{:}\,\rho.\sigma$, $B', x{:}\,\rho$ covers σ, and so $B', x{:}\,\rho \triangleright \sigma$ follows by induction. Therefore $B' \triangleright x{:}\,\rho.\sigma$.
- $\tau \equiv o(\tau_1, \ldots, \tau_{n-1}){\uparrow}\tau_n$: Immediate from the induction hypothesis.

This completes the proof. ▮

As an immediate corollary of these two facts we obtain:

Lemma 3.5 *If $B \mathrel{!\triangleright} \tau$ and $B \sqsubseteq B'$, then $B' \mathrel{!\triangleright} \tau$.*

It is worth noting that the requirement that $B \mathrel{!\triangleright} \tau$ is necessary. Consider again the example used to motivate the extension of the definition of covering. In the absence of the covering restriction it can happen that $B_x \triangleright \tau$ yet not $B \triangleright \tau$, despite the fact that $B_x \sqsubseteq B$. This observation is the key step in the proof of the following proposition:

Lemma 3.6 *If $B \triangleright \tau$, then $B \triangleright {\uparrow}\tau$.*

Proof By induction on derivations. ▮

3.4 Substitutions

In order to make precise the informal notion of an instance of a typed term scheme we must define the operation of substitution of terms for type variables. This operation is different from ordinary substitution in that capture is deliberately incurred, rather than avoided. There are two aspects to the discussion of substitutions, one being the mechanics of replacement and the other being the invariants which a substitution must satisfy.

A *substitution* is a finite function $S\colon \mathrm{TV} \to \mathrm{TTS}$. There are some important things to notice about substitutions. Suppose $S\colon \alpha \mapsto \tau$. The term τ may contain free occurrences of $v_1,\ldots,v_n$, where n is the length of $B(\alpha)$. The variable v_i refers to the ith variable from the left in the basis of occurrence of α in a term. In order to keep track of this basis, the operation of applying a substitution to a term must be defined with respect to a basis. Another observation about S is that it is also possible that τ contain occurrences of type variables β other than α (we do not admit circular substitutions). Since S may be defined on β, it is necessary that the application of S to a term "chase down" these bindings.

This motivates the extension of S to $S^*\colon \mathrm{BASIS} \times \mathrm{TTS} \to \mathrm{TTS}$ which applies S to a typed term τ relative to a basis B. The definition of $S^*[B,\tau]$ appears in Figure 3.1; it is assumed that B covers τ. For the sake of clarity that definition does not preserve standard–naming of bases. In order to do this, the definition of application two an abstraction must be changed so that σ' is bound to:

$$S^*[(B, v_k\colon \rho'), \sigma[v_k/x]][x/v_k]$$

where k is the length of B.

In order to tame this notation as much as possible, we adopt some simplifying notational conventions. We shall usually write S for S^* and omit the argument B whenever it is apparent from context. Whenever this is done, the brackets around the argument will be omitted. In particular, $S\alpha$ will be used to mean $S^*[B(\alpha),\alpha]$. The function S^* is further extended to bases, written $S^*[B]$, on a term–by–term basis, each term instantiated in its left–context in B.

For an example of the application of a substitution, let $B(\alpha) \equiv v_1\colon \sigma$,

$$
\begin{array}{l}
\textbf{if } \tau \equiv \mathsf{U}_i \;\rightarrow\; \mathsf{U}_i \\
\Box\; \tau \equiv \alpha \wedge S \textit{ is defined on } \alpha \;\rightarrow\; S^*[B, S[\alpha]] \\
\Box\; \tau \equiv \alpha \wedge S \textit{ is not defined on } \alpha \;\rightarrow\; \alpha \\
\Box\; \tau \equiv x{\uparrow}\sigma \;\rightarrow\; x{\uparrow}S^*[B_x, \sigma] \\
\Box\; \tau \equiv t{\uparrow}\sigma \;\rightarrow\; t{\uparrow}S^*[B_t, \sigma] \\
\Box\; \tau \equiv x{:}\,\rho.\sigma \;\rightarrow \\
\quad \textbf{let } \rho' \equiv S^*[B, \rho];\; \sigma' \equiv S^*[B, x{:}\,\rho'.\sigma] \\
\quad \textbf{in } x{:}\,\rho'.\sigma' \\
\Box\; \tau \equiv o(\sigma_1, \ldots, \sigma_{n-1}){\uparrow}\sigma_n \;\rightarrow \\
\quad \textbf{let } \sigma_1' \equiv S^*[B, \sigma_1]\;; \\
\qquad\quad \vdots \\
\qquad\quad \sigma_n' \equiv S^*[B, \sigma_n] \\
\quad \textbf{in } o(\sigma_1', \ldots, \sigma_{n-1}'){\uparrow}\sigma_n' \\
\textbf{fi}
\end{array}
$$

Figure 3.1: Application of a Substitution

$U(\alpha) \equiv \mathsf{U}_1$, $S{:}\,\alpha \mapsto v_1{\uparrow}\sigma$, and

$$\tau \equiv o(\lambda(v_1{:}\,\sigma.\alpha), \lambda(v_1{:}\,\sigma.o'(\alpha, \rho)))$$

Then $() \triangleright \tau$ (where $()$ is the empty basis), and

$$S^*[(), \tau] \equiv o(\lambda(v_1{:}\,\sigma.v_1{\uparrow}\sigma); \lambda(v_1{:}\,\sigma.o'(v_1{\uparrow}\sigma, \rho)))$$

The following are some trivial, but important, facts about substitutions:

Lemma 3.7 *If $B \sqsubseteq B'$, then $SB \sqsubseteq SB'$.*

It follows immediately that

Lemma 3.8 *If $B \equiv_\alpha B'$, then $SB \equiv_\alpha SB'$.*

and therefore

Lemma 3.9 *If $\langle B, \sigma\rangle \equiv_\alpha \langle B', \sigma'\rangle$, then $\langle SB, S\sigma\rangle \equiv_\alpha \langle SB', S\sigma'\rangle$.*

Suppose that $B \triangleright \sigma$ and S is a substitution. The notation $B \triangleright_S \sigma$ is almost the same as $SB \triangleright S\sigma$, except that the latter notation neglects the bases which are implicitly associated with the type variables occurring in B and σ. For instance, consider what happens if $B(\alpha)$ contains a free occurrence of type variable β and the substitution S is defined on β but not in α. Then we may have $B \triangleright \alpha$, but not $SB \triangleright S\alpha$ for the simple reason that whereas β has been replaced in B, it has not been replaced in $B(\alpha)$. This is precisely what $B \triangleright_S \sigma$ remedies — not only is S applied to B and σ, but also to all $B(\alpha)$ such that α occurs in B or σ. Under this convention, $B \triangleright_S \sigma$ is the same as $SB \triangleright S\sigma$, except that in addition all $B(\alpha)$ become $SB(\alpha)$.

The invariants that substitutions must satisfy can now be specified. Whenever S is defined on a type variable α, we insist that

1. α does not occur in $S\alpha$;

2. $\uparrow S\alpha \equiv U(\alpha)$;

3. $B(\alpha) \mathrel{!\triangleright_S} \alpha$.

So to update a valid substitution S with the binding $\alpha \mapsto \tau$, check that α not in $S\tau$, $\uparrow S\tau \equiv U(\alpha)$, and $B \mathrel{!\triangleright_S} \tau$. These are all easily–decidable conditions. The first is the familiar "occur–check" which ensures that no circular substitutions are admitted. The second ensures that the term τ has the appropriate universe as type. The third ensures not only that τ be well–formed in $B(\alpha)$, but also that $B(\alpha)$ cover τ. We insist that $B(\alpha)$ cover τ in order to conform to the implicit declaration model of type variables. The motivation is much the same as for bindings — terms which "move around" must have a fixed meaning. To see what happens if the requirement that $B(\alpha)$ cover τ were lifted, consider the basis $B \equiv (x{:}\rho, y{:}\tau)$ with $B(\alpha)$ empty and $B(\beta) \equiv x{:}\sigma$. Then α could be replaced by $\Pi(x{:}\sigma.\beta)\uparrow\mathsf{U}_1$ since it is well–formed in $B(\alpha)$. Yet despite the fact that $B \triangleright \alpha$, it is not the case that $B \triangleright \Pi(x{:}\sigma.\beta)$ since $B(\beta) \not\sqsubseteq B, x{:}\sigma$! The trouble is, once again, that the implicit declaration of β lies within the replacement for α, and hence functions as a bound, rather than free, variable.

3.5 Semantics

In order for the calculus of typed terms to be useful as the basis for decision methods for type theory, we must show that the derivations encoded in the annotation are meaningful with respect to the semantics of type theory. The sense in which it is argued that the logic of typed terms is valid is, roughly, that the well–formedness of τ is equivalent to $\tau^\circ \in (\uparrow\tau)^\circ$ according to the semantics of type theory. We begin by justifying the logic of typed term schemes in terms of the logic of pure typed terms.

The semantics of typed term schemes is given by the following theorem:

Theorem 3.2 *For any basis B and typed term scheme τ, $B \rhd \tau$ if and only if for any substitution S, $B \rhd_S \tau$.*

Proof The "if" part is trivial: take S to be the empty substitution. For the converse, proceed by induction on the definition of $B \rhd \tau$. Recall that this means that $SB \rhd S\tau$ with all $B(\alpha)$ replaced by $SB(\alpha)$. We show simultaneously (for the benefit of the induction) that if B covers τ, then SB covers $S\tau$ as well.

- $\tau \equiv \mathsf{U}_i$: Trivial.

- $\tau \equiv \alpha$: Since $B \rhd \alpha$, it follows that $B(\alpha) \sqsubseteq B$, and therefore that $SB(\alpha) \sqsubseteq SB$. Now $B(\alpha)\ !\!\rhd_S\ \alpha$ by the substitution invariants, so $B\ !\!\rhd_S\ \alpha$ by Lemma 3.5.

- $\tau \equiv x\uparrow\sigma$: By the induction hypothesis we have $B_x\ !\!\rhd_S\ \sigma$ and therefore $B \rhd_S x\uparrow\sigma$. Since SB_x covers $S\sigma$, it follows directly that SB covers $x\uparrow\sigma$.

- $\tau \equiv t\uparrow\sigma$: Similar to the case of a variable.

- $\tau \equiv x{:}\,\rho.\sigma$: By induction we have that $B\ !\!\rhd_S\ \rho$ and $B, x{:}\,\rho\ !\!\rhd_S\ \sigma$, and therefore $B \rhd_S x{:}\,\rho.\sigma$. If B covers τ, then $B, x{:}\,\rho$ covers σ, so by the induction hypothesis, $S(B, x{:}\,\rho)$ covers $S\sigma$, and thus SB covers $S\tau$.

- $\tau \equiv o(\sigma_1, \ldots, \sigma_{n-1})\uparrow\sigma_n$: By the induction hypothesis we have $B \rhd_S \tau_i$ for each $1 \leq i \leq n$, so the result follows directly by the definition of well–formedness. ∎

It follows immediately that well–formedness of typed term schemes corresponds to well–formedness of all ground instances:

Corollary 3.1 *If S maps all type variables in B and τ to typed terms without type variables, and $B \triangleright \tau$, then $SB \triangleright_0 S\tau$.*

Preservation of orderliness under substitution is also a direct result of Theorem 3.2:

Corollary 3.2 *If B is orderly, then so is SB.*

A useful technical result about substitutions also follows from Theorem 3.2:

Corollary 3.3 *If $B \triangleright_S \tau$, then $B \triangleright_{S' \circ S} \tau$ for any substitution S'.*

Proof Suppose that $B \triangleright_S \tau$, that is, $SB \triangleright S\tau$ with all $B(\alpha)$ replaced by $SB(\alpha)$. Then by the theorem, $SB \triangleright_{S'} S\tau$. But this means $S'SB \triangleright S'S\tau$ with the $SB(\alpha)$ replaced by $S'SB(\alpha)$, which is just $B \triangleright_{S' \circ S} \tau$. ∎

Finally, observe that if a valid substitution is updated according to the criteria given in Section 3.4, then the result is a valid substitution:

Corollary 3.4 *If S satisfies the substitution invariants, then so does $S' = S[\alpha \mapsto \tau]$, provided that τ satisfies the conditions: α not in $S\tau$, $\uparrow S\tau \equiv U(\alpha)$, and $B(\alpha) \,!\triangleright_S \tau$.*

Proof To see that the bindings in S are not invalidated, consider any type variable β in the domain of S, say $S{:}\,\beta \mapsto \sigma$. By the validity of S, $B(\beta) \triangleright_S \sigma$, and so $B(\beta) \triangleright_{S'} \sigma$ as well since we can regard S' as the composition of S with $\alpha \mapsto \tau$. The result holds for α by hypothesis. ∎

The semantics of typed terms is more–or–less straightforward, with only minor technical complications causing difficulty. The claim that we would like to make is that the theory of typed terms described in Section 3.2 is a valid formalization of type theory. By this we mean, roughly, that if $B \triangleright \tau$, then $\tau^\circ \in (\uparrow\tau)^\circ$ is a valid judgement, "modulo the base cases." The base cases, variables and atoms, are covered by taking this judgement to be relative to a hypothesis list H which "justifies", in a sense to be defined, the basis B and the atom set A. The structure of H is determined, in part, by the structure of B and by the atoms in $\mathcal{A}$. So to be more precise, if

$B \rhd \tau$, then $H \gg \tau^\circ \in (\uparrow\tau)^\circ$ will is a valid judgement for any H justifying the B and $\mathcal{A}$.

This formulation of the meaning of typed terms is correct provided that τ° is a syntactically well-formed untyped term, which is not the case for abstractions: the judgement $H \gg x.t \in T$ makes no sense. This deficiency is easily remedied by treating abstractions as a special case. The above statement of the meaning of well-formedness stands for non-abstractions; when $\tau \equiv x{:}\,\rho.\sigma$, the meaning of $B \rhd \tau$ is taken to be the meaning of the two premises, $B \rhd \rho$ and $B, x{:}\,\rho \rhd \sigma$.

In order for a well-formed typed variable or atom to determine a valid judgement, the hypothesis list of the judgement must justify the typing. Consider the case of a variable, say $B \rhd x\uparrow\tau$. In order that $x\uparrow\tau$ determine a valid judgement $H \gg x \in \tau^\circ$, H must bind x to τ°. Furthermore, H_x must justify B_x so as to preserve the dependency structure implicit in B. Therefore we make the following definition:

Definition 3.4 *Hypothesis list H* justifies *orderly basis B iff for each $x{:}\,\tau$ in B, the binding $x{:}\,\tau^\circ$ appears in H and H_x justifies B_x.*

Similarly, if $B \rhd t\uparrow\tau$, then the hypothesis list H must guarantee the truth of $t \in \tau^\circ$:

Definition 3.5 *Hypothesis list H* justifies *atom set $\mathcal{A}$ iff $H_t \gg t \in \tau^\circ$ for each $t\uparrow\tau \in \mathcal{A}$ and H_t is the longest prefix of H justifying B_t.*

The validity of the well-formedness rules for compound terms is based on the observation that in Martin-Löf's type theory there is a certain regularity in the form of the introduction rules which we illustrate as follows:

$$\begin{array}{ll} H \gg o(r; v.s) \in T & \text{by } o \text{ intro} \\ \quad H \gg r \in R & \\ \quad H, v{:}\,A \gg s \in S & \end{array}$$

It is implicit in this rule that T, R, and A are types in context H, and that S is a type in context $H, v{:}\,A$. The manner in which this is enforced is a matter of the organization of the proof theory as discussed in Chapter 2. If the introduction rule takes a parameter, then there is (explicitly or implicitly) a premise showing the parameter to be of a certain type. For instance, if the parameter is of the form $z.T$, then there is a subgoal of the form $H, z{:}\,B \gg T \in \mathsf{U}_k$ for some B a type in H.

This regularity is exploited by the logic of typed terms presented above. Abstractions are treated independently of the contexts in which they may occur (as defined by the constructors) because they are treated the same way by all of the introduction rules. The rule of well-formedness for abstractions is formulated so as to reflect this regularity.

To make this more concrete, consider the λ introduction rule:

$$\begin{array}{l} H \gg \lambda(x.b) \in \Pi(x{:}\,A.B) \quad \text{by intro} \\ \quad H, x{:}\,A \gg b \in B \end{array}$$

In order to apply this rule we must already know that $\Pi(x{:}\,A.B)$ and A are types in H, and that B is a type in $H, x{:}\,A$. This derivation is encoded by the typed term:

$$\lambda(x{:}\,\rho.\sigma)\uparrow\Pi(x{:}\,\rho.\uparrow\sigma)\uparrow\mathsf{U}_k$$

where $\rho^\circ \equiv A$, $\sigma^\circ \equiv b$, and $(\uparrow\sigma)^\circ \equiv B$. In order for this term to be well-formed in basis B, we must have

1. $B \rhd \rho$;
2. $B, x{:}\,\rho \rhd \sigma$;
3. $B \rhd \Pi(x{:}\,\rho.\uparrow\sigma)\uparrow\mathsf{U}_k$.

Now if H is any hypothesis list justifying B, then $H, x{:}\,\rho^\circ$ justifies $B, x{:}\,\rho$, and so we have:

1. $H \gg \rho^\circ \in (\uparrow\rho)^\circ$;
2. $H, x{:}\,\rho^\circ \gg \sigma^\circ \in (\uparrow\sigma)^\circ$;
3. $H \gg \Pi(x{:}\,\rho.\uparrow\sigma)^\circ \in \mathsf{U}_k$.

Rewriting these in terms of their erasures, we have

1. $H \gg A \in \mathsf{U}_k$;
2. $H, x{:}\,A \gg b \in B$;
3. $H \gg \Pi(x{:}\,A.B) \in \mathsf{U}_k$.

But these are (more than) enough to ensure that

$$H \gg \lambda(x.b) \in \Pi(x\colon A.B)$$

is a well–formed, valid judgement.

The claim that the logic of typed terms is a valid formulation of type theory is summarized in the following proposition:

Proposition 3.1 *Let B be an orderly basis, τ a non–abstraction typed term, and H a hypothesis list. If $B \triangleright \tau$ and H justifies B and the atom set $\mathcal{A}$, then $H \gg \tau^\circ \in (\uparrow\tau)^\circ$ is a valid judgement.*

We state this as a proposition rather than theorem because it plays a normative, rather than a descriptive, role. That is, this proposition *must* be true for any instance of the logic of typed terms obtained by fixing a set of term constructors and a set of atoms. The example of the justification of the well–formedness rule for λ's is an illustration of the sort of argument that must be carried out for any such instance.

Chapter 4

A Decision Method for Membership

The logic of typed terms presented in Chapter 3 was conceived as a tool for the construction of mechanical proof assistants for the ν-PRL system. In this chapter that machinery is used as the basis for a decision heuristic for the type membership relation. The idea is that the truth of the goal $H \gg s \in T@i$ can be tested by producing *typed analogs* of σ of s and τ of T and checking that $\uparrow\sigma$ matches τ. A typed analog of an untyped term s is a typed term σ such that $\sigma^\circ \equiv s$. Typed analogs of terms are built by a process called *annotation*. The annotation algorithm presented below is based on the type-inference algorithm of [Hindley 69,Milner 78], a unification-based method for determining the principle type scheme of an untyped λ-term. In this setting there is no notion of a principle type scheme; a given term can inhabit any number of structurally unrelated types. Nevertheless the approach proves to be quite useful.

The annotation algorithm takes as arguments a basis B and an untyped term s and produces a typed term σ such that $\sigma^\circ \equiv s$ and $B \triangleright \sigma$. The decision method for $H \gg s \in T@i$ builds a basis B and an atom set $\mathcal{A}$ such that H justifies both, then annotates s and T in B, producing σ and τ, respectively. By the definition of annotation and the semantics of typed terms, this implies that

$$H \gg s \in (\uparrow\sigma)^\circ$$

and

$$H \gg T \in (\uparrow\tau)^{\circ}$$

are both valid judgements. So, if $\uparrow\tau$ matches U_i and $\uparrow\sigma$ matches τ, then

$$H \gg s \in T @ i$$

is valid. The definition of "matches" is quite flexible — plausible candidates are identity computational equivalence unifiability, or some combination of these.

Because of the flexibility inherent in type theory, some aspects of the annotation algorithm are heuristic. For instance, the process of building an initial basis B and atom set $\mathcal{A}$ from the antecedent H of a goal is based on certain assumptions about the user's intentions. For instance, if there is a binding of the form $x{:}\,\mathsf{eq}(t,t,T) @ i$ in H, then the atoms $T\uparrow\mathsf{U}_i$ and $t\uparrow T\uparrow\mathsf{U}_i$ are added to $\mathcal{A}$. This is certainly a *sound* step (H justifies this atom), but there is no reason why it has to be the *right* one for the situation at hand. For instance, t might be 0 and T might be some type other than nat; choosing to "read" 0 as of type $T\uparrow\mathsf{U}_i$ rather than nat may or may not be the best choice under the circumstances. But it is important to note that the annotation algorithm is not committed to this choice — *any* means of producing B and $\mathcal{A}$ from H is acceptable, provided that H justifies the results. The presentation below is intended to serve only as a first cut at building a proof assistant. The method developed below is designed as a flexible, tunable framework, not as a definitive solution. The criterion for success of the method lies in its ability to be adjusted to the practical setting of proof development in ν-PRL, not in an analytical result about the completeness of the algorithm.

This chapter is organized as follows. Section 4.1 is devoted to a generalization of the unification algorithm. This algorithm plays a crucial role in the construction of decision heuristics. In Section 4.2 a detailed account of annotation is presented. This section consists of two parts. The first is an outline of annotation, illustrating the elements of the method. Then a detailed account of several term constructors from type theory is given on a case–by–case basis. In Section 4.3 possible avenues for improvement of the basic annotation algorithm are explored. A decision heuristic for the membership relation of the ν-PRL logic is given in Section 4.4. The chapter closes with a brief discussion of methods for efficient implementation of our algorithms.

4.1 Constrained unification

The heart of the annotation method is a generalization of the unification computation [Robinson 65], called *constrained unification*. It is an extension in two separable, but related senses. In one dimension, the unifiability relation is extended to terms with binding operators. The class of variables which may be instantiated by a substitution is separate from the class of variables which may be bound by some binding construct. The letters α, β, and γ are used for instantiable variables. Unifiability is further extended by relativizing the instantiable variables to a decidable predicate on terms, called the *constraint condition* for that variable. The algorithm is not committed to any particular constraint condition, but for the purposes of this chapter it is the well-formedness relation on typed terms. Both of these extensions are conservative: if the constraint conditions associated with the instantiable variables are always true and if terms do not have binding operators, then two terms unify in the ordinary sense if and only if they unify in the extended sense.

The extension of unifiability to binding operators is quite straightforward — the term $u.\sigma$ unifies with $v.\tau$ if $\sigma[x/u]$ unifies with $\tau[x/v]$, where x is new. The idea is that the bodies of the abstractions must "match up" by the same structural conditions that are normally associated with unification. This extension unification to the scope binding operators requires that the notion of instantiation must be generalized to admit capture–incurring replacements of variables by terms with references to binding operators. For instance, the term $\lambda(x.o(\alpha))$ unifies with the term $\lambda(y.o(y))$ under the substitution $\alpha \mapsto v_1$, where v_1 is the standard name of the first bound variable.

The addition of an instantiation constraint is also quite simple. To each instantiable variable, α, is associated a decidable predicate on terms, called the *constraint condition* for α. A term τ may be bound to α only if τ satisfies the constraint associated with α. The constraint conditions must be defined with respect to a basis which provides a binding for each of the free v_i occurring in the term τ. In this chapter the constraint condition associated with type variable α is the relation $B(\alpha) \; !\!\triangleright \; \tau$.

The unification algorithm appears in Figure 4.1. The input to Unify consists of two basis–term pairs, $\langle B, \sigma \rangle$ and $\langle B', \sigma' \rangle$ such that

1. B and B' are orderly;
2. $B \equiv_\alpha B'$;
3. $B \triangleright \sigma$;
4. $B' \triangleright \sigma'$.

It is assumed that B and B' adhere to the standard naming convention, though for the sake of clarity, the algorithm does not explicitly preserve this property. Unify returns a substitution S such that $\langle SB, S\sigma \rangle \equiv_\alpha \langle SB', S\sigma' \rangle$. Notice that since B and B' have standard variable names, and $B \equiv_\alpha B'$, this is equivalent to $S\sigma \equiv_\alpha S\sigma'$ by the definition of $\equiv_\alpha$ on basis–term pairs. It follows from the semantics of typed terms schemes (Theorem 3.2, page 60) that $B \triangleright_S \sigma$ and $B' \triangleright_S \sigma'$.

The algorithm is defined by induction on the structure of σ and σ'. There are two base–caes, one for σ a type variable, and one for σ' a type variable. Otherwise, the definition of unifiability guarantees that σ and σ' have the same outer structure, and so the algorithm is called recursively on corresponding components to build a unifying substitution.

For the sake of the induction, Unify is defined in terms of Unify$'$ which takes a substitution, S, as an additional parameter. The conditions on the arguments are essentially the same as those for Unify, except that all of the type variables are instantiated by S. In particular, this means that the requirement that B and B' be orderly is replaced by the requirement that they be S-*orderly*, that is orderly under the $\triangleright_S$ relation rather than $\triangleright$. (Recall that $\triangleright_S$ is the same as $\triangleright$ except that all occurrences of type variables are instantiated according to S.) The conditions on the arguments to Unify$'$ are:

- B and B' are S-*orderly*;
- $SB \equiv_\alpha SB'$;
- $B \triangleright_S \sigma$;

- $B' \triangleright_S \sigma$.

The correctness of Unify follows from the following theorem:

Theorem 4.1 *Suppose that* Unify$'[S, \langle B,\sigma\rangle, \langle B',\sigma'\rangle] = S'$, *and that the above rstrictions on the arguments are met. Then S' is an extension of S such that $\langle S'B, S'\sigma\rangle \equiv_\alpha \langle S'B', S'\sigma'\rangle$.*

Proof By induction on the structure of σ; see Figure 4.1. Note that if σ is not a type variable, then the outermost structure of $S\sigma$ is the same as that of σ, as can be easily verified by inspecting the definition of substitution in Chapter 3.

- If $S\sigma \equiv \alpha$ and $S\sigma'$ is acceptable as a replacement for α in S, then by the test for acceptability, S' is a valid substitution. Furthermore, $S'B \equiv_\alpha S'B'$ since $\equiv_\alpha$ is preserved under substitution, and $S'\sigma \equiv S'\sigma'$ by construction of S'.

- The symmetric case is handled symmetrically.

- If both terms are the same variable x, then simply unify their type fields (in their respective left contexts) to obtain a unifier for the two terms. The recursive call satisifies the conditions of Unify$'$ because the definition of well–formedness ensures that $B \triangleright_S x{\uparrow}\tau$ only if $B_x \triangleright_S x{\uparrow}\tau$.

- The case of atoms is similar, with B_t playing the role of B_x.

- Let $\sigma \equiv x{:}\rho.\tau$ and $\sigma' \equiv x{:}\rho'.\tau'$. The conditions for the first recursive call are met by the definition of well–formedness of abstractions. Therefore, by the induction hypothesis

  $$\langle S_1B, S_1\rho\rangle \equiv_\alpha \langle S_1B', S_1\rho'\rangle.$$

  The fact that S extends S' implies that $B \triangleright_{S_1} \rho$, and that $B' \triangleright_{S_1} \rho'$, so the conditions for the second recursive call are met. Let S_2 be the result of the second call. Then by induction

  $$\langle (S_2B, x{:}\, S_2\rho), S_2\tau\rangle \equiv_\alpha \langle (S_2B', x{:}\, S_2\rho'), S_2\tau'\rangle.$$

  The identity

  $$\langle S_2B, S_2\rho\rangle \equiv_\alpha \langle S_2B', S_2\rho'\rangle$$

follows from the fact that S_2 extends S_1 and that $\equiv_\alpha$ is preserved under substitution. These facts together are sufficient to obtain

$$\langle S_2B, x{:}\, \rho.\tau\rangle \equiv_\alpha \langle S_2B', x{:}\, \rho'.\tau'\rangle,$$

as desired.

- Let $\sigma \equiv o(\sigma_1, \ldots, \sigma_{n-1}) \uparrow \sigma_n$, and $\sigma' \equiv o(\sigma'_1, \ldots, \sigma'_{n-1}) \uparrow \sigma'_n$. The recursive calls to Unify$'$ all satisfy the conditions on the arguments because of the definition and semantics of well-formedness. Let S_n be the result of the last call. Then by the fact that S_n extends S and each of the S_i, the following identities hold:

$$\langle S_nB, S_n\sigma_i\rangle \equiv_\alpha \langle S_nB', S_n\sigma'_i\rangle \quad (1 \leq i \leq n)$$

It follows that

$$\langle S_nB, S_no(\sigma_1, \ldots, \sigma_{n-1}) \uparrow \sigma_n\rangle \equiv_\alpha \langle S_nB', S_no(\sigma'_1; \ldots; \sigma'_{n-1}) \uparrow \sigma'_n\rangle.$$

by the definition of $\equiv_\alpha$ on compound terms. ∎

Note that the test for well-formedness of a candidate replacement for a type variable, $B(\alpha) \rhd_S \sigma'$, can be performed without a recursive analysis of σ'. It is known, by hypothesis, that $B \rhd_S \sigma'$, so it suffices to check that $SB(\alpha)$ covers $S\sigma'$. Also note that in Figure 4.1, no provision is made for ensuring that bases have standard names or that the names of the bound variables of two abstractions may differ. It is a trivial matter to incorporate the requisite substitutions into the algorithm.

Some examples will serve to illustrate Unify. Consider the terms

$$\sigma \equiv \Pi(x{:}\, \mathsf{nat}\uparrow\mathsf{U}_1.\alpha)\uparrow\mathsf{U}_1$$

and

$$\tau \equiv \Pi(x{:}\, \mathsf{nat}\uparrow\mathsf{U}_1.\mathsf{nat}\uparrow\mathsf{U}_1)\uparrow\mathsf{U}_1$$

where $B(\alpha) \equiv ()$, and $U(\alpha) \equiv \mathsf{U}_1$. The terms σ and τ are each well-formed in the empty basis. Unify$[\langle(), \sigma\rangle, \langle(), \tau\rangle]$ succeeds, returning the substitution $S{:}\, \alpha \mapsto \mathsf{nat}\uparrow\mathsf{U}_1$. It is easy to verify that $S\sigma \equiv_\alpha S\tau$. Now consider

$$\sigma \equiv \Pi(x{:}\, \mathsf{U}_1.x\uparrow\mathsf{U}_1)\uparrow\mathsf{U}_2$$

$$\text{Unify}[\langle B,\sigma\rangle,\langle B',\sigma'\rangle] = \text{Unify}'[\emptyset,\langle B,\sigma\rangle,\langle B',\sigma'\rangle]$$

$$
\begin{array}{l}
\text{Unify}'[S,\langle B,\sigma\rangle,\langle B',\sigma'\rangle] = \\
\quad \textbf{if } S\sigma \equiv \alpha \ \wedge\ \alpha \notin S\sigma' \ \wedge\ \uparrow S\sigma' \equiv U(\alpha) \ \wedge\ B(\alpha) \mathrel{!\triangleright_S} \sigma' \ \rightarrow\ S[\alpha \mapsto \sigma'] \\
\quad \square\ S\sigma' \equiv \alpha \ \wedge\ \alpha \notin S\sigma \ \wedge\ \uparrow S\sigma \equiv U(\alpha) \ \wedge\ B(\alpha) \mathrel{!\triangleright_S} \sigma \ \rightarrow\ S[\alpha \mapsto \sigma] \\
\quad \square\ \sigma \equiv x\uparrow\tau \ \wedge\ \sigma' \equiv x\uparrow\tau' \ \rightarrow\ \text{Unify}'[S,\langle B_x,\tau\rangle,\langle B'_x,\tau'\rangle] \\
\quad \square\ \sigma \equiv t\uparrow\tau \ \wedge\ \sigma' \equiv t\uparrow\tau' \ \rightarrow\ \text{Unify}'[S,\langle B_t,\tau\rangle,\langle B'_t,\tau'\rangle] \\
\quad \square\ \sigma \equiv x{:}\,\rho.\tau \ \wedge\ \sigma' \equiv x{:}\,\rho'.\tau' \ \rightarrow \\
\quad\quad \textbf{let } S_1 = \text{Unify}'[S,\langle B,\rho\rangle,\langle B',\rho'\rangle] \ ; \\
\quad\quad \textbf{in } \ \text{Unify}'[S_1,\langle (B,x{:}\,\rho),\tau\rangle,\langle (B',x{:}\,\rho'),\tau'\rangle] \\
\quad \square\ \sigma \equiv o(\sigma_1,\ldots,\sigma_{n-1})\uparrow\sigma_n \ \wedge\ \sigma' \equiv o(\sigma'_1,\ldots,\sigma'_{n-1})\uparrow\sigma'_n \ \rightarrow \\
\quad\quad \textbf{let } S_1 = \text{Unify}'[S,\langle B,\sigma_1\rangle,\langle B',\sigma'_1\rangle] \ ; \\
\quad\quad\quad\quad S_2 = \text{Unify}'[S_1,\langle B,\sigma_2\rangle,\langle B',\sigma'_2\rangle] \ ; \\
\quad\quad\quad\quad \vdots \\
\quad\quad\quad\quad S_{n-1} = \text{Unify}'[S_{n-2},\langle B,\sigma_{n-1}\rangle,\langle B',\sigma'_{n-1}\rangle] \\
\quad\quad \textbf{in } \ \text{Unify}'[S_{n-1},\langle B,\sigma_n\rangle,\langle B',\sigma'_n\rangle] \\
\quad \textbf{fi}
\end{array}
$$

Figure 4.1: Unification Algorithm

and

$$\tau \equiv \Pi(x{:}\, \mathsf{U}_1.\alpha)$$

with $B(\alpha)$ and $U(\alpha)$ as before. The call Unify$[\langle(),\sigma\rangle,\langle(),\tau\rangle]$ fails because the term $v_1 \uparrow \mathsf{U}_1$ is not well-formed in $B(\alpha)$. But if $B(\alpha)$ were $v_1{:}\, \mathsf{U}_1$, the call to Unify would succeed with $S{:}\, \alpha \mapsto v_1 \uparrow \mathsf{U}_1$, a unifier for σ and τ. For a more complex example, let

$$\sigma \equiv \Pi(x{:}\, \alpha.\Pi(y{:}\, \gamma.\alpha))$$

and

$$\tau \equiv \Pi(x{:}\, \mathsf{U}_1.\Pi(y{:}\, x \uparrow \mathsf{U}_1.\alpha))$$

where

$$B(\alpha) \equiv (), U(\alpha) \equiv \mathsf{U}_2$$

and

$$B(\gamma) \equiv v_1{:}\, \alpha, U(\gamma) \equiv \mathsf{U}_1.$$

Both σ and τ are well-formed in empty basis. Unify$[\langle(),\sigma\rangle,\langle(),\tau\rangle]$ succeeds with the substitution $S{:}\, \{\, \alpha \mapsto \mathsf{U}_1, \gamma \mapsto v_1 \uparrow \mathsf{U}_1 \,\}$. Notice that once α becomes bound to U_1 by the outer abstraction, $B(\gamma)$ becomes, in effect, $v_1{:}\, \mathsf{U}_1$; the term $v_1 \uparrow \mathsf{U}_1$ is well-formed in that basis.

4.2 Annotation

This section is devoted to the construction of an annotation algorithm, Ann. It is based on the system of typed terms of Chapter 3 and on the unification algorithm just presented. Ann takes as arguments a basis B and an untyped term t and produces a typed analog τ of t which is well-formed in B. The algorithm presented below is not intended as the definitive solution to the annotation problem, but as a starting point for the construction of a practical reasoning assistant for ν-PRL. There is considerable leeway in the choice of annotation methods; an implementation will surely incorporate heuristics that can be suggested only as a result of practical experience.

Annotation methods can be quite complex, and the algorithm can appear, at first glance, to be hopelessly obscure. In the interest of the reader's sanity, the algorithm is presented in stages, beginning with a rough outline

$$
\begin{array}{l}
\mathrm{Ann}[B,t] = \mathit{Ann}'[\emptyset, B, t] \\
\\
\mathrm{Ann}'[S,B,t] = \\
\quad \textbf{if } t \equiv x \wedge x{:}\sigma \textit{ occurs in } B \ \rightarrow\ \langle S, x{\upharpoonright}\sigma\rangle \\
\quad \square\ t{\upharpoonright}\sigma \in \mathcal{A} \textit{ for some } \sigma \ \rightarrow\ \langle S, t{\upharpoonright}\sigma\rangle \\
\quad \square\ t \equiv o(r; s) \ \rightarrow \\
\qquad \textbf{let } \langle S_1, \rho\rangle = \mathrm{Ann}'[S, B, r]\ ; \\
\qquad\qquad \langle S_2, \sigma\rangle = \mathrm{Ann}'[S_1, B, s]\ ; \\
\qquad\qquad \langle S_3, \tau\rangle \textit{ be such that } B \rhd_{S_3} \tau \\
\qquad \textbf{in } \langle S_3,\ o(\rho;\sigma){\upharpoonright}\tau\rangle \\
\quad \square\ t \equiv o(r; u.s) \ \rightarrow \\
\qquad \textbf{let } \langle S_1, \rho\rangle = \mathrm{Ann}'[S, B, r]\ ; \\
\qquad\qquad \langle S_2, \mu\rangle \textit{ be such that } B\ !\!\rhd_{S_2} \mu\ ; \\
\qquad\qquad \langle S_3, \sigma\rangle = \mathrm{Ann}'[S_2, (B, u{:}\,\rho), s]\ ; \\
\qquad\qquad \langle S_4, \tau\rangle \textit{ be such that } B \rhd_{S_3} \tau \\
\qquad \textbf{in } \langle S_4,\ o(\rho; u{:}\,\mu.\sigma){\upharpoonright}\tau\rangle \\
\quad \textbf{fi}
\end{array}
$$

Figure 4.2: Outline of the Annotation Algorithm

(see Figure 4.2). The outline is then fleshed out on a case–by–case basis, with a discussion of the techniques used in each case.

Ann takes an orderly basis B and an untyped term t as arguments. It is assumed that B provides a binding for each free variable in t and that B covers the atom set $\mathcal{A}$ of the underlying logic of typed terms.[1] The result of a call to Ann is a substitution S and a typed term τ such that $(S\tau)^\circ \equiv t$ and $B \rhd_S \tau$.

Ann is defined in terms of Ann′ which takes a substitution S as additional argument. The arguments to Ann′ are assumed to satisfy the following conditions:

1. B binds each of the free variables in t;
2. B covers the atom set $\mathcal{A}$;

[1] See Section 3.3 for the definition of covering.

3. B is S-orderly;

The result is a substitution S', which extends S, and a typed term τ such that $(S'\tau)^{\circ} \equiv t$ and $B \rhd_{S'} \tau$.

Consider the outline of Ann$'$ in Figure 4.2. The base cases of Ann are t a variable or an atom. Suppose that t is some variable, x. There is, by assumption, a binding $x{:}\sigma$ in B. The typed analog of t is taken to be $x{\uparrow}\sigma$ since $B \rhd_S x{\uparrow}\sigma$ and $S[x{\uparrow}\sigma]^{\circ} \equiv x$. If t is not a variable, but there is an atom $t{\uparrow}\sigma$ in $\mathcal{A}$, then $t{\uparrow}\sigma$ is taken to be the typed analog of t. Since B covers $\mathcal{A}$, it follows that $B_t \rhd_S t{\uparrow}\sigma$, and the erasure of $S[t{\uparrow}\sigma]$ is t, as desired.

The next two cases are outlines of the annotation of compound terms, illustrating those aspects of the annotation process which all constructors have in common. Consider the case of a constructor o which does not bind any variables, and let $t \equiv o(r;s)$. The intention is to build a typed analog $o(\rho;\sigma){\uparrow}\tau$ of t. The first step is to annotate r. By the induction hypothesis, the result of the recursive call to Ann, $\langle S_1, \rho\rangle$, satisfies $(S_1\rho)^{\circ} \equiv r$ and $B \rhd_{S_1} \rho$. Next s is annotated, extending the substitution S_1. The result is $\langle S_2, \sigma\rangle$, where S_2 is an extension of S_1, $B \rhd_{S_2} \sigma$, and $(S_2\sigma)^{\circ} \equiv s$. Since S_2 extends S_1, it follows that $B \rhd_{S_2} \rho$ as well. Furthermore, since r has no type variables and $(S_1\rho)^{\circ} \equiv r$, it follows that $(S_2\rho)^{\circ} \equiv r$. The last step in the annotation of t is the construction of a term τ to serve as the type of $o(\rho,\sigma)$. In order to ensure that the resulting term is well-formed, τ must be built so as to conform to the introduction rule for the constructor o. The process of building τ will often involve further calls to Unify and Ann, and so this step is characterized as the construction of a substitution S_3 (extending S_2) and a typed term τ such that $B \rhd_{S_3} \tau$ and τ matches ρ and σ in accordance with o introduction. Since S_3 extends S_2, it follows that $B \rhd_{S_3} \rho$ and $B \rhd_{S_3} \sigma$. Erasures are preserved by the instantiation of type variables, so $(S_3\rho)^{\circ} \equiv r$ and $(S_3\sigma)^{\circ} \equiv s$. Putting all the pieces together, we have $B \rhd_{S_3} o(\rho;\sigma){\uparrow}\tau$ and $S_3[o(\rho;\sigma){\uparrow}\tau]^{\circ} \equiv o(r;s)$, as desired.

The annotation for a term constructor o which binds a variable is very similar to the previous case. The main difference is that the domain type, μ, of the abstraction $u.s$ must be built in such a way that $B \mathrel{!\rhd_{S_2}} \mu$. This ensures that the basis $B, u{:}\,\mu$ is S_2-orderly, as required for the second recursive call to Ann. As before, each substitution is built as an extension of the previous one, and the type field τ matches the component terms according to the introduction rule for o. The verification that the requirements for the result of Ann are met is very similar to that for non–binding constructors.

Notice that if μ is constructed by annotation of an untyped term, then μ will satisfy $B \triangleright_{S_2} \mu$, but B will not necessarily *cover* μ. The only way that B can fail to cover μ is if μ contains a type variable α such that $B(\alpha) \not\sqsubseteq B$. In order to ensure that this does not occur, we introduce the notion of "clipping." It follows from the fact that μ is well–formed in B that each $B(\alpha)$ which is not a prefix of B is an extension of B. So if we replace each such $B(\alpha)$ by B, then B covers μ, and so $B \mathrel{!\triangleright_{S_2}} \mu$, as desired. We refer to this operation as clipping of μ to basis B. In the presentation below, whenever we write $B, x{:}\tau$, it is assumed that τ has been clipped to B. The use of clipping here reflects the inability to defer the instantiation of μ any longer — some commitment to the structure of its instances must be made before annotation can proceed.

It is important to note that even at this early stage of its development, the annotation algorithm incorporates some heuristic decisions which are subject to modification as experience dictates. The principal heuristic is the choice of the logic of typed terms (see the discussion in Chapter 3). The structure of this logic determines the overall structure of the annotation algorithm and the criteria for its correctness. But even within the limited range of the logic of typed terms there is considerable room for maneuvering to enhance the performance of Ann. For example, the outline in Figure 4.2 treats t as an atom whenever possible. This is based on two principles: first, that the algorithm which builds $\mathcal{A}$ makes clever choices of atoms; second, that such cleverly contrived atoms are likely to play a significant role in the determination of the type of a term. If either of these assumptions is not true, then the "early" use of atoms can be detrimental. An alternative strategy is to use $\mathcal{A}$ only if the other means of annotating t fail.

A major source of informed guesswork in Ann is construction of the "synthesized" components of the typed analog of a term. The synthesized components of a typed term are those components which have no correlate in the erasure of that term. Examples are the type field of a term, the domain type of an abstraction, and the auxiliary field of terms like **decide**. The techniques that can be brought to bear in the construction of the synthesized components of a term are highly dependent on the particular constructor. We turn, then, to the elaboration of Figure 4.2 by considering each of several constructors from type theory.

We begin with the simplest case: t is some universe type U_i. Since universes are well-formed typed terms and are their own erasure, the annotation is trivial:

$$t \equiv \mathsf{U}_i \;\rightarrow\; \langle S, \mathsf{U}_i \rangle$$

Next consider **void**, the constructor for the empty type. It is a member of U_1, and is well-formed in any basis, and so is annotated as follows:

$$t \equiv \mathsf{void} \;\rightarrow\; \langle S, \mathsf{void}{\uparrow}\mathsf{U}_1 \rangle$$

There is a difficulty with this typing — **void** certainly is a level 1 type, but it is also a level i type for any i. The choice of U_1 is premature. The solution to this problem is straightforward, but tends to obscure the essence of the algorithm. Discussion of a solution is deferred to Section 4.3. For now, whenever there is a choice of universe type, U_1 will be used.

Associated with the type **void** is the elimination form $\mathsf{any}(s)$. This is an interesting constructor because it can be of any type whatsoever (hence its name), provided that the type of s is **void**. To determine whether or not this is so, annotate s, obtaining σ, and determine whether the type field of σ is $\mathsf{void}{\uparrow}\mathsf{U}_1$. It is overly restricitve to insist that the type of σ be *literally* $\mathsf{void}{\uparrow}\mathsf{U}_1$; instead we require only that the type of σ *unify* with $\mathsf{void}{\uparrow}\mathsf{U}_1$. This approach affords some flexibility; for instance, the type of σ might be some type variable α which is instantiated to $\mathsf{void}{\uparrow}\mathsf{U}_1$ by unification. In later extensions to the annotation algorithm, this use of Unify will also provide for considering equations between terms as well as finding common instances.

Similar tests arise often in Ann, and so we introduce the function $\mathrm{Check}[S, B, \sigma, \tau]$ (where $B \rhd_S \sigma$ and $B \rhd_S \tau$) which returns a substitution S' extending S such that $\langle S'B, S'\sigma \rangle \equiv_\alpha \langle S'B, S'\tau \rangle$.[2] Check is defined as follows:

$$\mathrm{Check}[S, B, \sigma, \tau] = \mathrm{Unify}'[S, \langle B, \sigma \rangle, \langle B, \tau \rangle]$$

Let us return to the annotation of $\mathsf{any}(s)$. Given that the type of σ is **void**, it is necessary to select a type for $\mathsf{any}(\sigma)$. In the absence of contextual information, there is no reasonable choice that Ann might make other than

[2] Since the same basis B is used for both terms, this is the same as saying that $S'\sigma \equiv_\alpha S'\tau$.

to not make a choice at all. This is accomplished by introducing a new type variable α to serve as the type of $\mathsf{any}(\sigma)$. Its type is then free to be determined by the requirements imposed by its surrounding context. Thus any is treated as follows:

$$\begin{aligned} t \equiv \mathsf{any}(s) \ \rightarrow\ & \mathbf{let}\ \langle S_1, \sigma\rangle = \mathrm{Ann}'[S, B, s]\ ;\\ & \qquad S_2 = \mathrm{Check}[S_1, B, \uparrow\sigma, \mathsf{void}\uparrow \mathsf{U}_1]\ ;\\ & \qquad B(\alpha) \equiv B\ ;\ U(\alpha) \equiv \mathsf{U}_1\\ & \mathbf{in}\ \ \langle S_2, \mathsf{any}(\sigma)\uparrow\alpha\rangle \end{aligned}$$

(Two programming conventions about type variables are used here. The first is that whenever Ann binds $B(\alpha)$ and $U(\alpha)$, it is implicit that α is chosen as a new type variable. The second is that the basis and universe associated with a type variable persist beyond the scope of the **let**. These bindings are best viewed as assignments to the two global variables B and U.)

The proof of correctness of this clause of Ann is tedious, but not difficult. After the first recursive call, we have $B \rhd_{S_1} \sigma$ and $S_1[\sigma]^\circ \equiv s$. After the call to Check,

$$S_2[\uparrow\sigma] \equiv \uparrow S_2[\sigma] \equiv \mathsf{void}\uparrow\mathsf{U}_1.$$

Since S_2 extends S_1, $B \rhd_{S_2} \sigma$ and $S_2[\sigma]^\circ \equiv s$. Now,

$$B \rhd_{S_2} \mathsf{any}(\sigma)\uparrow\alpha$$

follows from $B \rhd_{S_2} \sigma$ and $B \rhd_{S_2} \alpha$, by the definition of the well-formedness relation. The former holds by the induction hypothesis and the latter is true if

$$S_2 B(\alpha) \sqsubseteq S_2 B$$

which is follows from the fact that $B(\alpha) \equiv B$. To see that S_2 produces a typed analog of t, observe that

$$\begin{aligned} S_2[\mathsf{any}(\sigma)\uparrow\alpha]^\circ &\equiv (\mathsf{any}(S_2\sigma)\uparrow S_2[\alpha])^\circ\\ &\equiv \mathsf{any}(S_2[\sigma]^\circ)\\ &\equiv \mathsf{any}(s) \end{aligned}$$

as desired. This argument is typical of that for all of the constructors, and so in the sequel a detailed correctness proof will not be presented.

Consider the annotation of $\mathsf{any}(x)$ in the basis $B, x{:}\,\alpha$, where $B(\alpha) \equiv B$. First, the argument to any is annotated with result $x \uparrow \alpha$. Then α is instantiated to $\mathsf{void} \uparrow \mathsf{U}_1$ by the call to Check, and a new type variable β (with $B(\beta) \equiv B, x{:}\,\alpha$) is introduced. The annotated form is $\mathsf{any}(x \uparrow \mathsf{void} \uparrow \mathsf{U}_1) \uparrow \beta$, which is well-formed in $B, x{:}\,\alpha$, as desired.

Turning to the dependent function type, suppose that $t \equiv \Pi(x{:}\,r.s)$. Both r and s must be types; this is enforced by insisting that the annotated forms have universes as types. The annotation method for Π is:

$$
\begin{array}{lll}
t \equiv \Pi(x{:}\,r.s) & \rightarrow \textbf{let} & \langle S_1, \rho \rangle = \mathrm{Ann}'[S, B, r] \ ; \\
& & S_2 = \mathrm{Check}[S_1, B, \uparrow\rho, \mathsf{U}_1] \ ; \\
& & \langle S_3, \sigma \rangle = \mathrm{Ann}'[S_2, (B, x{:}\,\rho), s] \ ; \\
& & S_4 = \mathrm{Check}[S_3, (B, x{:}\,\rho), \uparrow\sigma, \mathsf{U}_1] \\
& \ \ \textbf{in} & \langle S_4, \Pi(x{:}\,\rho.\sigma) \uparrow \mathsf{U}_1 \rangle
\end{array}
$$

NOTE: By the convention adopted above, ρ is implicitly clipped to basis B in the second recursive call to Ann$'$!

The methods used for λ and apply are reminiscent of those employed by Milner in his type inference algorithm for ML, but generalized to the case of a dependent function space. To annotate $\lambda x.s$, the type of x is allowed to be determined by constraints imposed by the annotation of s, rather than by an *a priori* commitment to a domain type. This is accomplished by setting the type of x to be a type variable α, as follows:

$$
\begin{array}{lll}
t \equiv \lambda(x.s) & \rightarrow \textbf{let} & B(\alpha) \equiv B \ ; \ U(\alpha) \equiv \mathsf{U}_1 \ ; \\
& & \langle S_1, \sigma \rangle = \mathrm{Ann}'[S, (B, x{:}\,\alpha), s] \ ; \\
& & S_2 = \mathrm{Check}[S_1, (B, x{:}\,\alpha), \uparrow\uparrow\sigma, \mathsf{U}_1] \\
& \ \ \textbf{in} & \langle S_1, \lambda(x{:}\,\alpha.\sigma) \uparrow \Pi(x{:}\,\alpha.\uparrow\sigma) \uparrow \mathsf{U}_1
\end{array}
$$

Note that $B \mathrel{!\triangleright_{S_1}} \alpha$, so the basis $B, x{:}\,\alpha$ is orderly.

By way of example consider the annotation of the term $\lambda(x.\mathsf{any}(x))$ in basis B. A new type variable, α is introduced with basis $B(\alpha) \equiv B$. Then $\mathsf{any}(x)$ is annotated in basis $B, x{:}\,\alpha$. By the previous example, the result is $\mathsf{any}(x \uparrow \mathsf{void} \uparrow \mathsf{U}_1) \uparrow \beta$ where $B(\beta) \equiv B, x{:}\,\alpha$. Notice that B covers α, so the update $B, x{:}\,\alpha$ entails no clipping. Note also that the instantiation of α by $\mathsf{void} \uparrow \mathsf{U}_1$ also instantiates $B(\beta)$ to $B, x{:}\,\mathsf{void} \uparrow \mathsf{U}_1$. The annotation is completed by constructing the type of the term, with result

$$\lambda(x{:}\,\mathsf{void} \uparrow \mathsf{U}_1.\,\mathsf{any}(x \uparrow \mathsf{void} \uparrow \mathsf{U}_1) \uparrow \beta) \uparrow \Pi(x{:}\,\mathsf{void} \uparrow \mathsf{U}_1.\beta) \uparrow \mathsf{U}_1$$

which is well–formed in basis B.

For applications, the type components are typed separately, then their types are unified so that the domain of the function matches the type of the object to which it is applied, *viz*:

$$
\begin{array}{lll}
t \equiv \mathsf{apply}(r; s) & \rightarrow \textbf{let} & \langle S_1, \rho\rangle = \mathrm{Ann}'[S, B, r] \ ; \\
& & \langle S_2, \sigma\rangle = \mathrm{Ann}'[S_1, B, s] \ ; \\
& & S_3 = \mathrm{Check}[S_2, B, \uparrow\uparrow\sigma, \mathsf{U}_1] \ ; \\
& & B(\alpha) \equiv B, x{:} \uparrow\sigma \ ; \ U(\alpha) \equiv \mathsf{U}_1 \ ; \\
& & S_4 = \mathrm{Check}[S_3, B, \uparrow\rho, \Pi(x{:} \uparrow\sigma.\alpha)\uparrow\mathsf{U}_1] \ ; \\
& & \tau \equiv (S_4\alpha)[\sigma/x] \\
& \textbf{in} & \langle S_4, \mathsf{apply}(\rho;\sigma)\uparrow\tau\rangle
\end{array}
$$

The second call of Check ensures that there is a Π type whose domain is an instance of both the domain of the type of ρ and of the type of σ. The corresponding instance of the range type is bound to α in the subsitution S_3. Notice that $B(\alpha)$ is defined so as to account for the fact that α occurs within the scope of $x{:}\uparrow\sigma$ in the Π type. Since Π is a dependent function space, $S_3\alpha$ may contain free occurrences of x, which are replaced by σ in accordance with the introduction rule for **apply**. Note that $\uparrow\sigma$ must be clipped to basis B before building $B(\alpha)$.

There is a subtle difficulty with the substitution of σ for x in $S_4\alpha$ — what does it mean to replace free occurrences of x in a type variable? A type variable can be later instantiated so as to introduce occurrences of x which must be replaced by σ. The problem is that we have not extended the capture–avoiding substitution function (hereafter called *replacement* to avoid confusion with the other notion of substitution) to terms with type variables. The obvious remedy is to extend replacment to type variables by marking the type variable in such a way that the replacement is performed on all future instantiations of that variable. More precisely, the operation $\alpha[\tau/x]$ is defined to be a new type variable α' which is related to α by the following two constraints:

1. $B(\alpha')$ is $B(\alpha)$ with all occurrences of x replaced by τ (the binding for x is deleted);

2. For any substitution S, it must be that $S\alpha' \equiv (S\alpha)[\tau/x]$.

However, if this solution is adopted, then the definition of unification, among other functions, must be modified to ensure that the above conditions are true for all such pairs of type variables. The condition for α' to unify with σ is that there must be some term σ' such that $\sigma'[\tau/x] \equiv \sigma$. The trouble is that σ' is not uniquely determined because replacement is not one–to–one. Since σ' may itself involve type variables, unification must be recursively applied to any such σ', and so the situation recurs.

Another solution, motivated by the apparent intractibility of the above method, is to use clipping — the type variables β in $S_4\sigma$ have their bases restricted to B, thereby preventing any further instantiation by terms with free occurrences of x in them. Thus σ is clipped to basis B before applying the substitution to $S_4\alpha$. This is somewhat restrictive, but it accords with the principle underlying the annotation method that decisions about the annotation of a term are made locally, immediately after annotating the components of that term. Type variables are used to defer decisions about the types of terms as long as possible; this use of clipping represents the inability to defer a choice any longer.

To illustrate the annotation of apply, consider the term $\mathsf{apply}(\lambda(z.z); x)$ in basis $B, x{:}\,\alpha, y{:}\,\beta$ where $B(\alpha) \equiv B$ and $B(\beta) \equiv B, x{:}\,\alpha$. The first argument to apply is annotated, yielding $\lambda(z{:}\,\gamma.z{\uparrow}\gamma){\uparrow}\Pi(z{:}\,\gamma.\gamma){\uparrow}\mathsf{U}_1$, where γ is a new type variable with basis $B(\gamma) \equiv B, x{:}\,\alpha, y{:}\,\beta$. The annotation of the second argument, x, yields $x{\uparrow}\alpha$. Then $\Pi(z{:}\,\gamma.\gamma)$, the type of the first argument, is unfified with $\Pi(z{:}\,\alpha.\delta)$ where δ is new and $B(\delta) \equiv B, x{:}\,\alpha, y{:}\,\beta, z{:}\,\alpha$. Unify maps γ and δ to α, a valid substitution so that the result is

$$\mathsf{apply}(\lambda(z{:}\,\alpha.z{\uparrow}\alpha){\uparrow}\Pi(x{:}\,\alpha.\alpha){\uparrow}\mathsf{U}_1; x{\uparrow}\alpha){\uparrow}\alpha$$

Notice that no clipping of α need occur since $B(\alpha) \sqsubseteq B, x{:}\,\alpha, y{:}\,\beta$. If both x and y are discharged by introducing typed λ's, the type of the entire term would be

$$\Pi(x{:}\,\mathsf{void}{\uparrow}\mathsf{U}_1.\Pi(y{:}\,\beta.\alpha))$$

which is well-formed in B.

The last type that we shall consider is the union type. The case of $t \equiv r + s$ is straightforward:

$$\begin{aligned} t \equiv r + s \;\rightarrow\; \textbf{let}\ \langle S_1, \rho\rangle &= \mathrm{Ann}'[S, B, r]\ ; \\ S_2 &= \mathrm{Check}[S_1, B, \rho, \mathsf{U}_1]\ ; \end{aligned}$$

$$
\begin{aligned}
&\qquad \langle S_3, \sigma\rangle = \mathrm{Ann}'[S, B, s] \ ; \\
&\qquad S_4 = \mathrm{Check}[S_3, B, \sigma, \mathsf{U}_1] \\
&\mathbf{in}\ \ \langle S_4, (\rho + \sigma)\!\uparrow\!\mathsf{U}_1\rangle
\end{aligned}
$$

The treatment of inl is similar to that of any: the right half of the union type is taken to be a type variable which can be instantiated in accordance with the surrounding context:

$$
\begin{aligned}
t \equiv \mathsf{inl}(s)\ \ \longrightarrow\ &\mathbf{let}\ \langle S_1, \sigma\rangle = \mathrm{Ann}'[S, B, s] \ ; \\
&\qquad B(\alpha) \equiv B \ ; \ U(\alpha) \equiv \mathsf{U}_1 \ ; \\
&\qquad S_2 = \mathrm{Check}[S_1, B, \uparrow\!\sigma, \mathsf{U}_1] \ ; \\
&\qquad \tau \equiv (\uparrow\!\sigma + \alpha)\!\uparrow\!\mathsf{U}_1 \\
&\mathbf{in}\ \ \langle S_2, \mathsf{inl}(\sigma)\!\uparrow\!\tau\rangle
\end{aligned}
$$

The decide constructor illustrates heuristic type determination and the use of "auxiliary fields" in typed terms. Recall that the typed analog of a decide term has an additional component encoding the range type $x.T$ (see the introduction rule for decide in Appendix B). It is necessary for the annotation algorithm to "guess" this type based on the types of the other components. Leaving this aside for the moment, the annotation method is:

$$
\begin{aligned}
&t \equiv \mathsf{decide}(r; u.s_1; v.s_2)\ \ \longrightarrow \\
&\quad \mathbf{let}\ \langle S_1, \rho\rangle = \mathrm{Ann}'[S, B, r] \ ; \\
&\qquad B(\alpha) \equiv B \ ; \ U(\alpha) \equiv \mathsf{U}_1 \ ; \\
&\qquad B(\beta) \equiv B \ ; \ U(\beta) \equiv \mathsf{U}_1 \ ; \\
&\qquad S_2 = \mathrm{Check}[S_1, B, \uparrow\!\rho, (\alpha + \beta)\!\uparrow\!\mathsf{U}_1] \ ; \\
&\qquad \langle S_3, \sigma_1\rangle = \mathrm{Ann}'[S_2, (B, u{:}\,\alpha), s_1] \ ; \\
&\qquad \langle S_4, \sigma_2\rangle = \mathrm{Ann}'[S_3, (B, v{:}\,\beta), s_2] \ ; \\
&\qquad \sigma \equiv \ldots \ ; \\
&\qquad \tau \equiv \sigma[\rho/z] \\
&\quad \mathbf{in}\ \ \langle S_4, \mathsf{decide}(\rho; u{:}\,\alpha.\sigma_1; v{:}\,\beta.\sigma_2; z{:}\,\uparrow\!\rho.\sigma)\!\uparrow\!\tau\rangle
\end{aligned}
$$

Notice that $\uparrow\!\rho$ must be clipped to basis B in order for its two components, α and β to be eligible for addition to the basis in the second and third recursive calls to Ann$'$. Just as with apply, the range type σ must be clipped to basis B when the substitution of ρ for z is applied in order to ensure that no future instantiations of σ can introduce a z.

First, the principal argument r of the decide form is annotated, yielding ρ. The type of ρ must be a union type, say $\alpha + \beta$. The substitution

S_2 instantiates α and β to be the left and right components of the type of ρ, *i.e.*, $\uparrow S_2\rho \equiv S_2(\alpha + \beta)$. Then the two abstractions are annotated. The domain of the first is the left half of the type of ρ, and the domain of the second is the right half. In order for **decide** to be well-formed the types of σ_1, σ_2, and the type of the whole term, τ, must be appropriate instances of some term σ, the range type of the **decide** form. The definition of "appropriate instance" appears on page 49 in Chapter 3. The upshot is that we must build a term $\boldsymbol{z}\colon \uparrow\rho.\sigma$ to serve as the auxiliary field of the annotated form of a **decide**. This term must satisfy the these constraints (leaving the substitutions implicit for the sake of clarity):

1. $\boldsymbol{B} \rhd \boldsymbol{z}\colon \uparrow\rho.\sigma$;
2. $\uparrow\sigma_1 \equiv \sigma[\mathsf{inl}(\boldsymbol{u})/\boldsymbol{z}]$;
3. $\uparrow\sigma_2 \equiv \sigma[\mathsf{inr}(\boldsymbol{v})/\boldsymbol{z}]$;
4. $\tau \equiv \sigma[\rho/\boldsymbol{z}]$.

As discussed in Chapter 3, the use of term identity can be relaxed to weaker notions of equality.

Any method of building such a σ may be used in the annotation method defined above. A simple method is to superimpose σ_1 and σ_2, replacing corresponding occurrences of $\mathsf{inl}(\boldsymbol{u})$ and $\mathsf{inl}(\boldsymbol{v})$ by a new variable $\boldsymbol{z}$ (with type $\uparrow\rho$), then checking that the result is well-formed. It is important to note that if this method is used, then terms $\uparrow\sigma_1$ and $\uparrow\sigma_2$ must be clipped to $\boldsymbol{B}$ in order to prevent later instantiation of these variables from introducing an occurrence of $\mathsf{inl}(\boldsymbol{u})$ or $\mathsf{inr}(\boldsymbol{v})$. Since the auxiliary field for the term must be built at the time that it is annotated, it is not possible to defer the selection of the values of these type variables any longer; some restrictions must be introduced.

Another heuristic can be brought to bear for the choice of the auxiliary field and type field of **decide** once we have a theory of equality of typed terms in place. Consider the term $\mathsf{decide}(\boldsymbol{e}; \boldsymbol{u}.3; \boldsymbol{v}.\mathsf{true})$; it is crucial that the range type itself be a **decide**, for there is no single type which has both 3 and **true** as members. A pattern-matching heuristic such as that outlined above will fail to find a suitable range type for this term. But we can always take the type of a **decide** to be another **decide** whenever we cannot find a suitable range type. There is no infinite regress here because the type of the second decide will be some universe U_i since this is the type of both arms. Returning to the annotation algorithm, this means that the auxiliary field is

$$z\colon \mathrm{TF}[\rho].\,\mathsf{decide}(z\!\uparrow \mathrm{TF}[\rho];\, u\colon \alpha.\!\uparrow\!\sigma_1;\, v\colon \beta.\!\uparrow\!\sigma_2)$$

unless $\uparrow\sigma_1 \equiv \uparrow\sigma_2$, in which case it is $z\colon \mathrm{TF}[\rho].\,\mathrm{TF}[\sigma_1]$. [3] There is a subtlety here: since we have "shifted" the binding structure of σ_1 and σ_2 by introducing the binding $z\colon \uparrow\rho$, both terms must be clipped to basis B in order for the result to be well–formed. Notice that the relationship between the range type and the types of the arms and of the entire term is based on the assumption that there is a suitable theory of equality of typed terms which incorporates the computation rules.

As an example, consider the term

$$t \equiv \lambda(x.\,\mathsf{decide}(\mathsf{any}(x);\, u.0;\, v.1)).$$

The "arms" of the decide are deliberately chosen to be simple so that we can concentrate on the annotation of the first component. The annotation of t in the empty basis begins by taking $x\colon \alpha$ where $B(\alpha)$ is empty and $U(\alpha)$ is U_1, and the body is annotated. This entails annotating $\mathsf{any}(x)$ in the basis $x\colon \alpha$. Since **any** insists that the type of its argument be **void**, α is instantiated to be $\mathsf{void}\!\uparrow\!\mathsf{U}_1$. The type of **any** is taken to be β (with $B(\beta) \equiv x\colon \mathsf{void}\!\uparrow\!\mathsf{U}_1$ and $U(\beta) \equiv \mathsf{U}_1$) so that it can be determined by context. Once $\mathsf{any}(x)$ has been annotated the decide annotation method forces its type to be some union type, say $(\beta_1 + \beta_2)\!\uparrow\!\mathsf{U}_1$. The two arms of the **decide** are annotated to yield $u\colon \beta_1.0\!\uparrow\!\mathsf{nat}\!\uparrow\!\mathsf{U}_1$ and $v\colon \beta_2.1\!\uparrow\!\mathsf{nat}\!\uparrow\!\mathsf{U}_1$, respectively. Since the types of both arms are identical, the range type is taken to be $z\colon \beta.\mathsf{nat}\!\uparrow\!\mathsf{U}_1$. The type of the entire term is then

$$\Pi(x\colon (\beta_1 + \beta_2)\!\uparrow\!\mathsf{U}_1.\mathsf{nat}\!\uparrow\!\mathsf{U}_1)\!\uparrow\!\mathsf{U}_1$$

in accordance with the annotation method for λ.

Suppose that the arms of the above term t were $u.\mathsf{apply}(u, 0)$ and $v.\mathsf{apply}(v, \mathsf{true})$ (where **true** inhabits the type **bool** which has not been explicitly treated here). Then the annotated forms of these terms would be

$$u\colon \beta_1.\,\mathsf{apply}(u\!\uparrow\!\Pi(w\colon \mathsf{nat}\!\uparrow\!\mathsf{U}_1.\gamma_1)\!\uparrow\!\mathsf{U}_1)\!\uparrow\!\gamma_1$$

[3] Recall that $\mathrm{TF}[\sigma]$ is the type field of σ.

and

$$v{:}\,\beta_2.\,\mathsf{apply}(v\uparrow\Pi(w{:}\,\mathsf{bool}\uparrow\mathsf{U}_1.\gamma_2)\uparrow\mathsf{U}_1)\uparrow\gamma_1$$

where $B(\gamma_1)$ is $x{:}\,\mathsf{void}\uparrow\mathsf{U}_1, u{:}\,\beta_1$ and $B(\gamma_2)$ is $x{:}\,\mathsf{void}\uparrow\mathsf{U}_1, v{:}\,\beta_2$. Notice that the variable $w{:}\,\mathsf{nat}\uparrow\mathsf{U}_1$ is clipped from $B(\gamma_1)$ and $B(\gamma_2)$ by the annotation of **apply**. A reasonable heuristic might be to take $\gamma_1 \equiv \gamma_2$, so that the auxiliary field of the **decide** would be $x{:}\,(\beta_1+\beta_2)\uparrow\mathsf{U}_1.\gamma_1$, and the result type would then also be γ_1. Notice that this entails clipping $B(\gamma_1)$ and $B(\gamma_2)$ to B. An alternative is to take the range type to be

$$z{:}\,(\beta_1+\beta_2)\uparrow\mathsf{U}_1.\,\mathsf{decide}(z; u{:}\,\beta_1.\gamma_1; v{:}\,\beta_2.\gamma_2; z{:}\,\mathsf{U}_1)\uparrow\mathsf{U}_1$$

so that the choice of range type is **decide**'d in each case.

4.3 Enhancements to Ann

In this section we discuss some directions for the extension of Ann. We begin with the matter of universe levels mentioned above and suggest some other improvements to Ann. We noted in Section 4.2 that the annotation method makes an untimely commitment to the universe level of many types. For instance, the annotation algorithm for **any** insists that the type of the argument be $\mathsf{void}\uparrow\mathsf{U}_1$. It would be desirable to defer this choice so that it can be made in accordance with the surrounding context. This is achieved by extending the class of universe indices to include a set of variables called *index variables* which range over numerals standing for the positive integers. The syntactic variables $\boldsymbol{i}$ and $\boldsymbol{j}$ range over this extended class of indices. Substitutions are extended to admit index variables into the domain and indices into the range. Unify is extended to allow U_κ (where κ is an index variable) to match U_i by binding κ to $\boldsymbol{i}$. Uses of U_1 in Ann can now be replaced by U_κ where κ is a new index variable. For instance, the annotated form of **void** becomes $\mathsf{void}\uparrow\mathsf{U}_\kappa$. Similarly, the annotation of **any** can require that σ, the argument, unify with U_κ rather than U_1. This solves the problem of premature commitment to universe levels.

A further liberalization of the treatment of universe levels by Ann is also possible. The annotation of $r + s$, for example, requires that both components be members of U_1. By using index variables, this means that both components must be of some fixed level $\boldsymbol{i}$. But this is unnecessarily restrictive. The universes of type theory are cumulative — any member of U_i is also a member of U_j for $j \geq i$. Therefore, rather than insist that r and s be of the *same* level, we can instead take the type to be U_i, where i is the larger of the indices of r and s. For instance, if the type of r is of level 2 and the type of s is of level 1, then the type of the entire term is U_2.

However, there is a difficulty with this approach: if the index of either of the types of the components is an index variable, then it is not possible to determine i. The solution is to introduce a set C of inequalities between index letters and insist that C always be satisfiable. For instance, let the indices of r and s be κ_1 and κ_2, respectively. Then the type assigned to $r+s$ is U_κ, and the inequalities $\kappa \geq \kappa_1$ and $\kappa \geq \kappa_1$ are added to C. Substitutions must respect C: κ may not be bound to i unless the replacement of κ by i in C leaves the result satisifiable. Notice that the decision problem for satisfiability of C is decidable by, for instance, the ARITH decision procedure of PRL.

Another enhancement to Ann which may prove useful in the construction of a decision method is the extension of type variables to arbitrary terms. There is no reason to insist that the type of a type variable be a universe. The universe $U(\alpha)$ associated with α can be replaced by a typed term $T(\alpha)$ which satisfies $B(\alpha)$!▷ $T(\alpha)$. Then α may be bound to τ only if, in addition to the other requirements, $\uparrow\tau \equiv T(\alpha)$.

Other means of enhancing Ann are certainly conceivable. One deficiency of the version presented in Secion 4.2 is that it is unable to take advantage of user-supplied type information. This information can be crucial to the success of Ann in a practical setting since it can potentially bear no relation whatsoever to the type (if any) which would be inferred for that term. For instance, if the the range type of a decide instance is supplied, then no guesswork need be applied to determine it. The chances for successful annotation are greatly improved. Similarly, if the type of a λ term is known, then so is the type of the domain of the function. Knowing the domain type will, in general, improve the effectiveness of Ann on the body of the λ term since this information may not be inferrable.

A closely related point is the observation that Ann very often applies Check to a type expression which it has just produced. This suggests a third argument, σ, be given to Ann so that if Ann returns S and τ, then $\langle SB, S\tau\rangle \equiv_\alpha \langle SB, S\sigma\rangle$. That is, σ serves as a pattern against which the inferred type must match. This requirement is easily met: add a call to Unify as the last step of annotation. But this solution misses the point because it does not use the information in σ to its greatest advantage. A better solution is to integrate σ into the annotation method so that, for instance, the annotation of a λ would take σ into account when determining the domain type. This seems to be a promising line of development.

A much more open-ended direction for enhancement is in the area of heuristic type determination. The version of Ann presented above incorporates several such methods, for instance, in the determination of the range type of decide. The case of ind is particularly interesting because the determination of the range type is the problem of "guessing" an induction hypothesis. Much work has been done in developing heuristics for this problem [Boyer & Moore 79]; it remains to see if they can be applied in Ann.

Finally, a much more difficult problem is discovering a solution to constrained substitution problem discussed above. If two type variables α and α' must satisfy some constraint (such as $S\alpha' \equiv (S\alpha)[\tau/x]$, which arose in connection with replacement), the definition of unification breaks down if the constraint condition is not invertible. In the case just mentioned, the variable α' unifies with σ only if there is some term σ' such that $\sigma'[\tau/x] \equiv \sigma$; the binding of α is taken to be some such σ'. There may be many (or no) choices for σ'. But among the choices which work "locally", there may be some which are better choices than others, depending on all the occurrences of α. This suggests that a unification algorithm which works under these constraints must employ backtracking to try various choices for σ'.

4.4 A decision method for membership

In this section we show how to use the tools developed above to build a decision heuristic for the membership relation of the PRL type theory. The problem is to decide the truth of the judgement

$$H \gg t \in A@i$$

under the supposition that H is well-formed. The outline of the algorithm is as follows:

1. Build an atom set $\mathcal{A}$ from H such that H justifies $\mathcal{A}$;
2. Build an orderly basis B from H such that H justifies B;
3. Let $\langle S_1, \tau\rangle = \mathrm{Ann}[B, t]$;
4. Let $\langle S_2, \sigma\rangle = \mathrm{Ann}'[S_1, B, A]$;
5. Let $S_3 = \mathrm{Unify}'[S_2, \langle B, \uparrow\tau\rangle, \langle B, \sigma\rangle]$;
6. Let $S_4 = \mathrm{Unify}'[S_3, \langle B, \uparrow\sigma\rangle, \langle B, \mathsf{U}_i\rangle]$.

If any of the steps fail, then the decision procedure fails.

The correctness of this algorithm is based on the semantics of typed terms (see Theorems 3.2 and 3.1) and on the correctness of Unify and Ann. Since H justifies the atom set $\mathcal{A}$ and the basis B, both

$$H \gg t \in (\uparrow S_4\tau)^\circ$$

and

$$H \gg A \in (\uparrow S_4\sigma)^\circ$$

are valid. But by the choice of S_4,

$$(\uparrow S_4\tau)^\circ \equiv (S_4\sigma)^\circ \equiv S$$

and

$$(\uparrow S_4\sigma)^\circ \equiv \mathsf{U}_i$$

Therefore

$$H \gg t \in S$$

and

$$H \gg S \in \mathsf{U}_i$$

are both valid judgements, as desired.

Returning to the example on page 80, we see that the above method will decide that

$$H \gg \lambda(x.\lambda(y.\mathsf{apply}(\lambda(z.z); x))) \in \Pi(x{:}\, U_1.\Pi(y{:}\, x.x)) \mathbin{@} 1$$

because the type field of the annotated form of the λ-term,

$$\Pi(x{:}\,\alpha.\Pi(y{:}\,\beta.\alpha)\uparrow\mathsf{U}_1)\uparrow\mathsf{U}_1$$

unifies with the annotated form of the Π-type,

$$\Pi(x\colon \mathsf{U}_1.\Pi(y\colon x{\uparrow}\mathsf{U}_1.x{\uparrow}\mathsf{U}_1){\uparrow}\mathsf{U}_1){\uparrow}\mathsf{U}_1$$

under the substitution $\alpha \mapsto \mathsf{U}_1$ and $\beta \mapsto x{\uparrow}\mathsf{U}_1$. Notice that $B(\alpha)$!▷ U_1 and $B(\beta)$!▷ $x{\uparrow}\mathsf{U}_1$, so this is a valid substitution.

It remains to specify how the atom set $\mathcal{A}$ and initial basis B are built. The basis B must bind each of the variables in H and must be orderly. The atom set $\mathcal{A}$ is built according to some criterion for determining likely base–cases of annotation. Any method at all can be used, subject only to the requirement that H justify $\mathcal{A}$. The following is a simple method for building $\mathcal{A}$ and B. First, ensure that the variables in H are named according to the standard convention. Then, for each binding $x\colon T\,@\,i$ in H, add $T{\uparrow}\mathsf{U}_i$ to $\mathcal{A}$ and add $x\colon T{\uparrow}\mathsf{U}_i$ to B. In addition, if $T \equiv \mathsf{eq}(a, b, A)$, add $T{\uparrow}\mathsf{U}_i$, $a{\uparrow}(T{\uparrow}\mathsf{U}_i)$, and $b{\uparrow}(T{\uparrow}\mathsf{U}_i)$ to $\mathcal{A}$. The motivation for this step is that if a hypothesized typing occurs in H, then it is a reasonable guess that it plays an important role in determining the truth of a membership judgement under the premises in H. It is easy to see that B is orderly, B covers $\mathcal{A}$, and H justifies both B and $\mathcal{A}$.

There is room for improvement of this aspect of the algorithm. For instance, it is likely that the numerals will be atoms with type nat (recall that there is no inherent reason why 0 must be of type nat). Therefore it is likely that $\mathcal{A}$ will contain $0{\uparrow}\mathsf{nat}{\uparrow}\mathsf{U}_1$. Another improvement would be to attempt to annotate each of the terms T that occur as bindings $x\colon T\,@\,i$ in H. Since H is presumed sound, it is known that T is a type of level i. This information can be used to strengthen the initialization procedure for Ann. This is best illustrated by example. Suppose that $x\colon A \rightarrow B\,@\,i$ occurs in H. It is possible to do better than to take $A \rightarrow B$ to be an atom with type U_i. The type structure rules of the ν-PRL type theory license the inference that A and B are level i types from the fact that $A \rightarrow B$ is. Therefore, if the annotation of A fails, the atom $A{\uparrow}\mathsf{U}_i$ can be added to $\mathcal{A}$, similarly for B (and, recursively, their components). If both A and B are unannotatable, the result will be $(A{\uparrow}\mathsf{U}_i \rightarrow B{\uparrow}\mathsf{U}_i){\uparrow}\mathsf{U}_i$, which is a considerable improvement: the type of x is at least known to be a function whereas before it was unanalyzable.

4.5 Implementation notes

We close this chapter with a discussion of implementation issues. It may seem, at first glance, that the algorithms for substitution, unification, and annotation are awkward to implement. The algorithms are presented so as to facilitate the presentation and the proofs, with efficiency playing a secondary role. This section consists of some observations which should be of help to the implementer.

Note that the relation $B \triangleright_S \tau$ is defined in such a way that one can implement the substitution S as a global table of bindings for each type variable α which are applied whenever α is accessed. Under this convention $B \triangleright_S \tau$ is just $B \triangleright \tau$; type variables in B, τ, and the $B(\alpha)$ are automatically replaced according to S.

Note also that Unify$'$ and Ann$'$ are defined in such a way that a parameter substitution S is an *update* — the result substitution is an extension of the argument substitution. It is a trivial matter to derive an imperative implementation of these two algorithms which takes S to be a global variable, updating the list of bindings as necessary. Each call to Unify$'$ or Ann$'$ will, in general, extend this table.

Simple inspection of the definition of $S^*[B,\sigma]$ in Chapter 3 reveals that the basis argument, B, is unnecessary provided that B adheres to the standard naming convention and that σ has no free variables which do not appear in B. Since variable occurrences have their types associated directly with them, and since free variables are named v_k for some k, the parameter B can be replaced by the length k of B, updating as necessary. Recursive calls with B_x as parameter are replaced by recursive calls with $i-1$ as parameter, where x is the ith variable in B.

The arguments to Unify can be limited to just two terms and a natural number, rather than two basis–term pairs. If there are two bases B and B' satisfying the conditions for Unify (see Section 4.1 above), then the arguments can be simply σ, σ', and k, where k is the length of B (and hence B' since $B \equiv_\alpha B'$). This brings a considerable savings in the overhead of implementing Unify.

This concludes the presentation of a decision method for the membership relation. The algorithm presented is a very flexible basis for the construction of useful decision methods. With experience, more heuristics will suggest themselves and will be incorporated into the PRL system. The foregoing development provides both an algorithmic basis for these enhancements and a theoretical framework with respect to which the correctness of any extensions can be judged. The effectiveness, however, can only be determined by using the implemented system. It remains to be seen whether or not the tools we have developed will prove useful in practice.

Chapter 5

A Decision Method for Equality

The subject of this chapter is the construction of a heuristic decision method for the equality relation of type theory. The method is based on the theory of typed terms developed in Chapter 3 and on the annotation algorithm of Chapter 4. The idea is that the truth of a judgement of the form $H \gg t = t' \in T@i$ can be tested by producing typed analogs τ of t, τ' of t', and σ of T, then testing whether or not $\tau = \tau'$ and $\uparrow\tau = \sigma$ in a theory of equality for typed terms. The equality relation one typed terms properly includes both the identity relation (modulo α-conversion) and the equivalence relation generated by the computation rules, and is therefore undecidable. A decidable fragment for use by the equality heuristic is easily defined. The equality heuristic is based on annotation; its effectiveness is therefore strongly affected by the effectiveness of Ann.

By defining an equality relation on typed terms, the restrictions that must be met by the annotation methods can be relaxed from identities to equalities (see, for example, the discussion of the annotation of **decide** on page 49). Furthermore, a decidable equality relation can be built into the unification algorithm so that equal terms are unifiable. The relation tested by Unify properly includes unifiability in the sense of Chapter 4 and the decidable fragment of the equality relation. Since unification lies at the heart of the annotation algorithm, this extension strengthens Ann considerably. Thus the strength of the equality heuristic depends on the strength of Ann and the strength of Ann is improved by the presence of a decidable equality relation on typed terms.

This chapter is organized as follows. Section 5.1 is devoted to the definition of a theory of equality of typed terms. This theory is parameterized by an arbitrary set of equations between typed terms and, in addition, includes all instances of the computation rules of type theory. A semantics of equality is given in terms of the underlying type theory. In Section 5.2 some issues in the construction of a decision method for the theory of equality of typed terms are discussed. The equality relation on typed terms is undecidable. But by limiting the sorts of derivations which can be considered and by limiting the applicability of the computation rules, a decidable fragment can be obtained. In Section 5.3 we show how to incorporate a decision method for equality into the unification algorithm of Chapter 4. This algorithm is then used as the basis for an equality heuristic for the ν-PRL system.

5.1 A theory of equality of typed terms

There are three key aspects to the equality relation of type theory that motivate the definition of equality for typed terms. The first is that there is a striking regularity among the equality rules for each of the terms. Each term constructor respects the equality relation in each of its argument positions. When an argument position is an abstraction, $x.t$, this means that the constructor respects equality in this position for any value of x in the domain of the abstraction. For instance, the equality rule for **decide** is:

$$\begin{array}{ll} H \gg \mathsf{decide}(e; u.s; v.t) = \mathsf{decide}(e'; u.s'; v.t') \in T(e)_z & \text{by intro} \\ \quad H \gg e = e' \in A + B & \\ \quad H, u{:}\, A \gg s = s' \in T(\mathsf{inl}(u)) & \\ \quad H, v{:}\, B \gg t = t' \in T(\mathsf{inr}(v)) & \end{array}$$

The second and third subgoals illustrate the equality principal for abstractions: $u.s$ "=" $u.s'$ if $s = s'$ for all values of u (in a type determined from context).

The second aspect of equality in type theory that motivates the definition below is that the membership and equality relations respect equality in the type position. That is, if $a = b \in A$ and $A = B \in \mathsf{U}_i$, then $a = b \in B$. In the system of typed terms the type of a term appears as a component of that term, called the type field. The fact that type membership respects equality allows the type field to be treated like the other components of the term — equals may be substituted for equals.

The third aspect of equality that we exploit is the fact that the membership rules are really just special cases of the equality rules. An introduction rule is just the reflexive instance of an equality rule. For instance, the union introduction rule,

$$\begin{array}{l} H \gg \mathsf{inl}(a) \in A + B \quad \text{by intro} \\ \quad H \gg a \in A \end{array}$$

is just the **inl** equality rule,

$$\begin{array}{l} H \gg \mathsf{inl}(a) = \mathsf{inl}(a') \in A + B \quad \text{by intro} \\ \quad H \gg a = a' \in A \end{array}$$

with $a \equiv a'$. Thus a derivation of type membership by repeated uses of the introduction rules has the same structure as a derivation of equality by repeated use of the corresponding equality rules. Recall that typed terms encode a derivation of membership with this structure. The theory of equality is defined only on well-formed typed terms. In effect, it retraces the derivation of well-formedness for each term, ensuring that the two terms are component-wise equal. For example, let

$$\sigma \equiv \lambda(x{:}\,\rho.\tau)\uparrow\Pi(x{:}\,\rho.\uparrow\tau)\uparrow\mathsf{U}_1$$

and

$$\sigma' \equiv \lambda(x{:}\,\rho'.\tau')\uparrow\Pi(x{:}\,\rho'.\uparrow\tau')\uparrow\mathsf{U}_1$$

and supppose that they are both well-formed in the empty basis. Then an equality derivation consists of a "walk" through the basis-term pairs as follows: $\langle(),\sigma = \tau\rangle$ if $\langle(),\rho = \rho'\rangle$ and $\langle x{:}\,\sigma,\tau\rangle = \langle x{:}\,\sigma'.\tau'\rangle$. Notice that each term remains well-formed in its basis and that the two bases are equal throughout the derivation, provided that they are equal to begin with.

These observations motivate the following definition of equality of typed terms. Let B and B' be orderly bases, and let σ and σ' satisfy $B \rhd \sigma$ and $B' \rhd \sigma'$. Then $\langle B,\sigma\rangle = \langle B',\sigma'\rangle$ if:

- $\langle B,\sigma\rangle \equiv_\alpha \langle B',\sigma'\rangle$;
- $\langle B_x,\tau\rangle = \langle B'_x,\tau'\rangle$, where $\sigma \equiv x{\uparrow}\tau$ and $\sigma' \equiv x{\uparrow}\tau'$;
- $\langle B,\sigma\rangle = \langle B'',\sigma''\rangle$ and $\langle B',\sigma'\rangle = \langle B'',\sigma''\rangle$, provided that $B'' \triangleright \sigma''$;
- $\langle B,\sigma_1\rangle = \langle B',\sigma'_1\rangle$ and $\langle (B,x{:}\sigma_1),\sigma_2\rangle = \langle (B',x{:}\sigma'_1),\sigma'_2\rangle$ where $\sigma \equiv x{:}\,\sigma_1.\sigma_2$ and $\sigma' \equiv x{:}\,\sigma'_1.\sigma'_2$.
- $\langle B,\sigma_i\rangle = \langle B',\sigma'_i\rangle$ for each $1 \le i \le n$, where $\sigma \equiv o(\sigma_1,\ldots,\sigma_{n-1}){\uparrow}\sigma_n$ and $\sigma' \equiv o(\sigma'_1,\ldots,\sigma'_{n-1}){\uparrow}\sigma'_n$.

Two bases B and B' are equal, $B = B'$ if they are the same length and are componentwise equal: for each $x{:}\,\sigma$ in B, and corresponding $x{:}\,\sigma'$ in B', the equation $\langle B_x,\sigma\rangle = \langle B'_x,\sigma'\rangle$ holds. Later on it will be convenient to assume that $B = B'$ so that the premise of the equality rule for variables can be dropped, and the condition for the applicability of the symmetry–transitivity rule is strengthened by adding $B = B'' = B'$. There is no equality rule for atoms (beyond $\equiv_\alpha$) since any two occurrences of an atom have the same type field. Note that the combined symmetry–transitivity rule is eliminable from above theory since the relation it defines is already symmetric and transitive.

The above theory is uninteresting because it lacks any base–cases other than identity. However, the relation it defines is symmetric and transitive, even without the symmetry–transitivity rule. This observation, in combination with the fact that equality is defined by induction on the structure of the two terms, is the key to seeing that the relation defined is decidable: a decision method consists of a componentwise inspection of the two terms.

The theory can be strengthened while retaining decidability by adding a decidable set $\mathcal{E}$ of *hypothesized equations* as base–cases. The phrase "hypothesized equations" is used because $\mathcal{E}$ will, in the method to be outlined below, be obtained from a ν-PRL hypothesis list. The equations in $\mathcal{E}$ are between typed terms with syntactically identical type fields, *i.e.*, if $\sigma = \tau$ occurs in $\mathcal{E}$, then ${\uparrow}\sigma \equiv {\uparrow}\tau$. The terms which participate in equations in $\mathcal{E}$ must, of course, be well–formed. Therefore, $\mathcal{E}$ is defined with respect to a basis $B_\mathcal{E}$ such that each term τ appearing in $\mathcal{E}$ satisfies $B_\mathcal{E} \triangleright \tau$ and $B_\mathcal{E}$ covers τ. The following rule is then added to the above rules for equality:

- $\langle B,\sigma\rangle = \langle B',\sigma'\rangle$ if $\sigma = \sigma'$ appears in $\mathcal{E}$.

In order to ensure that $B \triangleright \sigma$ and $B \triangleright \sigma'$ whenever they are equal, it is assumed that $B_{\mathcal{E}}$ is a prefix of both B and B'.

The computation rules of type theory are also base–cases of this theory. This is accomplished by adding the rule

- $\langle B, \sigma\rangle = \langle B', \sigma'\rangle$ if $\sigma \simeq \sigma'$, where $\simeq$ denotes the equivalence relation generated by the reduction rules of type theory.

For instance, the equation $\langle B, \mathsf{apply}(\lambda(x{:}\,\rho.\sigma); \tau)\rangle = \langle B, \sigma[\tau/x]\rangle$ is derivable because $\mathsf{apply}(\lambda(x{:}\,\rho.\sigma); \tau) \simeq \sigma[\tau/x]$, in accordance with the computation rule for **apply** (otherwise known as β conversion).

This addition makes the theory undecidable (since it includes the conversion relation of the untyped λ-calculus). However, it is useful for theoretical reasons to include this relation, primarily because some theory of equality for typed terms is need so that the relationships between the components of a term and its type can be liberalized. For instance, the annotation method for **decide** can use the technique discussed on page 83 since it is only necessary that $\uparrow\sigma_1 = \tau[\mathsf{inl}(u)/z]$ (as opposed to insisting on identity).

The equality relation is extended to typed term schemes as follows:

- $\langle B, \alpha\rangle = \langle B', \alpha\rangle$.

The meaning of this extension if given by a theorem similar in spirit to Theorem 3.2.

Theorem 5.1 *Let $B \triangleright \sigma$ and $B' \triangleright \sigma'$. Then $\langle B, \sigma\rangle = \langle B', \sigma'\rangle$ if and only if for every substitution S, $\langle SB, S\sigma\rangle = \langle SB', S\sigma'\rangle$.*

Proof (Note: in order for $SB \triangleright S\sigma$ we must consider each $B(\alpha)$ to be replaced by $SB(\alpha)$; alternatively, we could introduce a notion of equality under a substitution analagous to the relation $\triangleright_S$.) For the "if" part, take S to be the empty substitution. For the "only if", proceed by induction on the size of the derivation.

- $(\equiv_\alpha)$ Follows immediately from the fact that identity is preserved under subsitution.

- $(\mathcal{E})$ Since hypothesized equations have no type variables, this case is trivial.

- ($\sigma \simeq \sigma'$) Follows from the observation that instantiation of type variables cannot eliminate a redex.

- ($\sigma \equiv \alpha \equiv \sigma'$) Follows directly from the fact that both terms are the same type variable α, so $S\sigma \equiv S\sigma'$.

- (Symmetry–transitivity) Follows directly by induction.

- ($\sigma \equiv x{\uparrow}\tau$, $\sigma' \equiv x{\uparrow}\tau'$) By the induction hypothesis, $\langle SB_x, S\tau\rangle = \langle SB'_x, S\tau'\rangle$. But $SB_x \equiv (SB)_x$, and $Sx{\uparrow}\tau \equiv x{\uparrow}S\tau$, and so the result follows by the equality rule for variables.

- ($\sigma \equiv x{:}\,\sigma_1.\sigma_2$, $\sigma' \equiv x{:}\,\sigma'_1.\sigma'_2$) By the induction hypothesis we have

$$\langle SB, S\sigma_1\rangle = \langle SB', S\sigma'_1\rangle$$

  and

$$\langle S(B, x{:}\,\sigma_1), S\sigma_2\rangle = \langle S(B', x{:}\,\sigma'_1), S\sigma'_2\rangle$$

  But $S(B, x{:}\,\sigma_1) \equiv SB, x{:}\,S\sigma_1$ by the definition of the application of a substitution to a basis, so the result follows by an application of the equality rule for abstractions.

- ($\sigma \equiv o(\sigma_1, \ldots, \sigma_{n-1}){\uparrow}\sigma_n$, $\sigma' \equiv o(\sigma'_1, \ldots, \sigma'_{n-1}){\uparrow}\sigma'_n$) By the induction hypothesis,

$$\langle SB, S\sigma_i\rangle = \langle SB', S\sigma'_i\rangle$$

  for each $1 \leq i \leq n$. The result follows by an application of the equality rule for compound terms.

This completes the proof. ▮

In particular, if a substitution replaces each type variable with a typed term, then the resulting basis–term pairs are equal as (ground) typed terms.

The semantics of the equality relation is given along the same lines as the semantics of typed terms given in Chapter 3. That is, we show that a derivation of equality between typed terms determines a derivation of equality in the underlying type theory. The basic fact about type theory that we exploit is that equal types are semantically interchangeable. Another way to say the same thing is to note that equality expresses codesignation — $a = b \in A$ means that a and b designate the same abstract object in type A. If A is a universe, this means that a and b designate the same type. The semantics of types guarantees that two equal types have the same membership and equality relation (though the converse does not hold for PRL).

To precisely state the meaning of an equation between typed terms, we must first redefine the notion of a hypothesis list H justifying a basis B. This is essentially the same definition as that given in Chapter 3, except that H can contain binding for a type *equal*, as opposed to *identical*, to σ°.

Definition 5.1 *Hypothesis list H* justifies *basis B if (1) for each $x{:}\sigma$ in B there is a binding $x{:}s$ in H such that $H \gg s = \sigma^\circ$ is true; and (2) H_x justifies B_x.*

We must also define the notion of a hypothesis list justifying a set of hypothesized equations.

Definition 5.2 *Hypothesis list H* justifies *an equation set $\mathcal{E}$ if $H \gg \sigma^\circ = \tau^\circ \in \uparrow\sigma$ is a true judgement for each equation $\sigma = \tau$ in $\mathcal{E}$.*

We are now in a position to state the semantics of the equality relation on typed terms.

Theorem 5.2 *Let B_1 and B_2 be orderly bases extending $B_{\mathcal{E}}$ such that $B_1 = B_2$. Let σ_1 and σ_2 be typed terms satisfying $B_1 \triangleright \sigma_1$, and $B_2 \triangleright \sigma_2$. Let H be a hypothesis list justifying the basis B_1, the atom set $\mathcal{A}$, and the equation set $\mathcal{E}$. If $\langle B_1, \sigma_1\rangle = \langle B_2, \sigma_2\rangle$, then*

$$H \gg (\uparrow\sigma_1)^\circ = (\uparrow\sigma_2)^\circ \in \uparrow\uparrow\sigma_1$$

and

$$H \gg \sigma_1^\circ = \sigma_2 \in (\uparrow\sigma_1)^\circ$$

are both valid judgements.

Proof Strictly speaking, this proposition can be proved only for a specific instance of the system of typed terms. What follows is a proof for the cases that are independent of the particular constructors and an example of the proof for the λ and Π constructors. We proceed by induction on derivations, regarding the implicit derivation of equality of B_1 and B_2 as a sequence of subderivations. It follows by induction, then, that H justifies B_2 as well.

- $(\equiv_\alpha)$ This is just Theorem 3.1, the semantics theorem for typed terms.
- $(\mathcal{E})$ By the fact that H justifies $\mathcal{E}$ and the assumption that $\uparrow\sigma_1 \equiv \uparrow\sigma_2$.
- $(\simeq)$ Computational equivalence is included in the equality relation of type theory.
- $(\sigma_1 \equiv x\uparrow\tau_1,\ \sigma_2 \equiv x\uparrow\tau_2)$ By the induction hypothesis,

  $$H \gg (\uparrow\tau_1)^\circ = (\uparrow\tau_2)^\circ \in \uparrow\uparrow\tau_1$$

  and

  $$H \gg \tau_1^\circ = \tau_2^\circ \in \uparrow\tau_1$$

  are both valid judgements. Therefore we obtain

  $$\begin{array}{l} H \gg x = x \in \tau_1^\circ \quad \text{by equality} \\ \quad H \gg x = x \in \sigma^\circ \\ \quad H \gg \sigma^\circ = \tau_1^\circ \in (\uparrow\sigma)^\circ \end{array}$$

  where $x{:}\,\sigma^\circ$ appears in H. The second subgoal is provable since H_x justifies $(B_1)_x$, which follows from the assumption that H justifies B_1.
- (Symmetry–transitivity) By the induction hypothesis and the fact that H covers B'' since $B'' = B$.

We use the λ constructor to illustrate the proof for compound terms. Consider the terms

$$\sigma_1 \equiv \lambda(x{:}\,\rho_1.\tau_1)\uparrow\Pi(x{:}\,\rho_1.\uparrow\tau_1)\uparrow\mathsf{U}_1$$

and

$$\sigma_2 \equiv \lambda(x{:}\,\rho_2.\tau_2)\uparrow\Pi(x{:}\,\rho_2.\uparrow\tau_2)\uparrow\mathsf{U}_1.$$

A derivation of $\langle B_1, \sigma_1\rangle = \langle B_2, \sigma_2\rangle$ consists of three subderivations:

1. $\langle B_1, \rho_1\rangle = \langle B_2, \rho_2\rangle$;
2. $\langle (B_1, x{:}\,\rho_1), \tau_1\rangle = \langle (B_2, x{:}\,\rho_2), \tau_2\rangle$;
3. $\langle B_1, \Pi(x{:}\,\rho_1.\uparrow\tau_1)\rangle = \langle B_2, \Pi(x{:}\,\rho_2.\uparrow\tau_2)\rangle$.

By the induction hypothesis we have

1. $H \gg (\uparrow\rho_1)^\circ = (\uparrow\rho_2)^\circ \in \uparrow\uparrow\rho_1$;

2. $H \gg \rho_1^\circ = \rho_2^\circ \in (\uparrow\rho_1)^\circ$;

3. $H, x{:}\,\rho_1^\circ \gg (\uparrow\tau_1)^\circ = (\uparrow\tau_2)^\circ \in \uparrow\uparrow\tau_1$;

4. $H, x{:}\,\rho_1^\circ \gg \tau_1^\circ = \tau_2^\circ \in (\uparrow\tau_1)^\circ$ ($H, x{:}\,\rho_1^\circ$ justifies $B, x{:}\,\rho_1$ and $B, x{:}\,\rho_2$ since H justifies B_1 which equals B_2).

These are sufficient for desired result, namely:

1. $H \gg \Pi(x{:}\,\rho_1^\circ.(\uparrow\tau_1)^\circ) = \Pi(x{:}\,\rho_2^\circ.(\uparrow\tau_2)^\circ) \in \mathsf{U}_1$;

2. $H \gg \lambda(x.\tau_1^\circ) = \lambda(x.\tau_2^\circ) \in \Pi(x{:}\,\rho_1^\circ.(\uparrow\tau_1)^\circ)$.

by application of the equality rules for Π and λ, respectively. ∎

5.2 Discussion of the equality relation

A key tool for building a heuristic proof assistant for the equality relation of ν-PRL is a decision method for the equality relation on typed terms. It was noted that the symmetry–transitivity rule is redundant because the relation defined by those rules is already symmetric and transitive. In order to maintain this property for a non-empty set $\mathcal{E}$ of equations, we assume that $\mathcal{E}$ is closed under the symmetry–transitivity rule. Then, removing this rule from the definition of the equality relation, we obtain a decision procedure for equality of typed terms. The algorithm so obtained is, metaphorically speaking, a "two-fingered" algorithm — the test for equality of two terms consists of running two "fingers" through both terms, component-by-component, recursively testing for equality. If each test succeeds, then the terms are equal. Notice that the bases play no active role in this algorithm; they may be safely replaced by a natural number k recording the length of the basis so that standard names can be picked when descending into the body of an abstraction.

It is well-known that the theory of interconvertibility of λ-terms is undecidable (see, for instance, [Stenlund 72]). It is therefore impossible to include all of the instances of the computation rules in the equality test. A new base case can be added to the equality test to determine whether or not two terms form a redex–contractum pair. Methods for detecting redices are well-known; see, for example,[Hoffmann & O'Donnell 84].

It is interesting to note that if symmetry and transitivity are included, then any algorithm for equality must explicitly use bases, whereas the algorithm defined above does not need to do so. In the early stages of this research, the author intended to implement the equality relation on typed terms by a modification to the congruence closure algorithm (see [Johnson 80,Nelson & Oppen 80]). The equality rule for abstractions is nearly identical to the equality rule for an ordinary binary operator o, except that a substitution must be performed in order to cancel the effect of the names of the bound variables. The idea behind the reduction to ordinary congruence closure is that a representation for abstractions can be defined so that that substitution is unnecessary, and so the equality rule for abstractions reduces to that of the other compound terms. This is best illustrated by example. Consider the terms $x{:}\,\sigma.o(x\uparrow\sigma)\uparrow\tau$ and $y{:}\,\sigma'.o(y\uparrow\sigma')\uparrow\tau'$, taking both with respect to some fixed basis B. For the sake of simplicity, we assume that neither σ nor τ contain any abstractions. Using the "name-free" representation of [deBruijn 72], these terms become $\mathbf{dot}(\sigma; o(\mathrm{BV}_1\uparrow\sigma)\uparrow\tau)$, and $\mathbf{dot}(\sigma'; o(\mathrm{BV}_1\uparrow\sigma')\uparrow\tau')$, respectively. The binary constructor **dot** represents the abstraction, and the nullary constructor BV_i represents the bound variable whose binding occurrence is the ith **dot** toward the root of the expression tree (extending into the basis as necessary). Since **dot** is just a binary term constructor, its equality rule is the ordinary pointwise-equality: $\mathbf{dot}(\sigma;\tau) = \mathbf{dot}(\sigma';\tau')$ only if $\sigma = \sigma'$ and $\tau = \tau'$. But since all abstractions are replaced in the name-free encoding by **dot**'s, we have eliminated the need for a special rule for binding operators.

It appears, then, that we have reduced the equality theory to the ordinary theory of congruence of equality, and thus can base a decision method on the well-known congruence closure algorithm. However, there is a snag. An application of transitivity cannot guarantee, in the absence of an explicit test, that the two bases implicitly associated with each term are equal. For instance, consider the terms

$$\sigma \equiv \lambda(\lambda(\lambda(o(\mathrm{BV}_1\uparrow\sigma_1; \mathrm{BV}_3\uparrow\sigma_3))))$$

and

$$\tau \equiv \lambda(\lambda(\lambda(o(\mathrm{BV}_1\uparrow\tau_1; \mathrm{BV}_3\uparrow\tau_3))))$$

These two terms are the name–free encodings of

$$\sigma' \equiv \lambda(x\colon \sigma_1'.\lambda(y\colon \sigma_2'.\lambda(z\colon \sigma_3'.o(x\uparrow\sigma_1'; z\uparrow\sigma_3'))))$$

and

$$\tau' \equiv \lambda(x\colon \tau_1'.\lambda(y\colon \tau_2'.\lambda(z\colon \tau_3'.o(x\uparrow\tau_1'; z\uparrow\tau_3'))))$$

respectively. It is entirely possible that the bodies of these two terms are equal (if $\sigma_1 = \tau_1$ and $\sigma_3 = \tau_3$), but this does not guarantee that σ and τ are equal! For instance, the basis associated with the first term has σ_2 as its second component, whereas the second has τ_2. If $\sigma_2 \neq \tau_2$, then the two terms are not equal. But under these assumptions the name–free forms of these terms *are* equal, a violation of soundness. The difficulty is that in the absence of bases it is not possible to prevent two terms from being equal when their bases of occurrence are not pointwise equal. Therefore, there can be no hypothesis list H which justifies both bases, and so no proof of equality can be obtained in type theory. Thus we see that it is not possible to implement a decision method for equality with summetry and transitivity without also taking into account the bases associated with each term.

5.3 A heuristic for equality

We now proceed to the construction of a decision heuristic for equality. The first step is to incorporate the equality relation for typed terms into the unification algorithm of Chapter 4. This idea is not new; see [Siekmann & Szabó 82] for an extensive overview of this topic. The addition is quite trivial: simply add

$$\langle SB, S\sigma\rangle = \langle SB', S\sigma'\rangle \;\rightarrow\; S$$

as the first clause of the conditional in Unify′ (see Figure 4.1 on page 71). Then, given $\langle B,\sigma\rangle$ and $\langle B',\sigma'\rangle$ such that $B = B'$, $B \triangleright \sigma$, and $B' \triangleright \sigma'$, the extended version of Unify returns a substitution S such that $\langle SB, S\sigma\rangle = \langle SB', S\sigma'\rangle$.

The incorporation of equality into unification is a conservative extension of both the unifiability relation (as defined in Chapter 4) and the equality relation (as implemented by the decision procedure). Thus, two terms which are unifiable in the sense of Chapter 4 are unifiable in the extended sense. In addition, the extended algorithm will succeed (without changing the substitution S) if the two terms are (detectably) equal. The equality decision procedure outlined above includes the set $\mathcal{E}$ of hypothesized equations and some instances of the computation rules, and so Unify will succeed for at least these cases. For example, suppose that the equation $\sigma = \tau$ appears in $\mathcal{E}$, let B be an orderly basis extending $B_{\mathcal{E}}$, let $B(\alpha)$ be B, and consider

$$\text{Unify}[\langle B, \lambda(x{:}\,\rho.\sigma)\rangle, \langle B, \lambda(x{:}\,\alpha.\tau)\rangle]$$

(The type field of the λ's is omitted for the sake of clarity). The unifying substitution $\alpha \mapsto \rho$ is obtained as follows: first α is mapped ρ by the unification of the domains of the abstraction, then the bodies are unified. Since the bodies are equal (by construction of $\mathcal{E}$), the unification succeeds without updating the substitution. Of course, σ and τ need not have been equal for such trivial reasons — the process of recursively calling Unify and checking for equality can be iterated, allowing for more complex patterns of interaction. It is entirely possible that instantiation of a type variable can lead to the discovery of an equality. For instance, consider the terms $\lambda(x{:}\,\alpha.\alpha)$ and $\lambda(x{:}\,\sigma.\tau)$ with the bases and $\mathcal{E}$ as above. Unify instantiates α to σ, allowing the bodies to unify since $\sigma = \tau$.

The extended unification algorithm can be directly incorporated into the annotation algorithm. This has the effect of relaxing the "matching" constraints enforced by Ann to equality (from strict identity). This change affords considerable flexibility in the discovery of typings for terms. For instance, when annotating $\mathsf{any}(s)$, the type of s does not have to be *identical* to $\mathsf{void}\!\uparrow\!\mathsf{U}_1$, just *equal* to it. Since this restriction is enforced by a call to Unify (via the function Check), this relaxation is automatically incorporated into Ann — if Unify can determine that the type of s is equal to $\mathsf{void}\!\uparrow\!\mathsf{U}_1$, then it will succeed. Thus the definition of a decidable sub–relation of equality provides for both the construction of an equality heuristic and the enhancement of the effectivenes of the membership heuristic.

Given these tools, we are now in a position to describe a decision heuristic for equality in the ν-PRL logic. Given the goal $H \gg r = s \in T@i$, proceed as follows:

- Build a initial basis $\mathcal{B}$ and atom set $\mathcal{A}$ as in Chapter 4;

- Build an equation set by adding $a \uparrow A \uparrow \mathsf{U}_i = b \uparrow A \uparrow \mathsf{U}_1$ to $\mathcal{E}$ for each $x{:}\,\mathsf{eq}(a, b, A) \,@\, i$ in H; the basis $B_{\mathcal{E}}$ is defined to be B;
- Obtain typed analogs ρ of r, σ of s, and τ of t in basis B;
- Let $S = \text{Unify}[\langle B, \rho\rangle, \langle B, \sigma\rangle]$;
- Let $S' = \text{Unify}'[S, \langle B, \uparrow\rho\rangle, \langle B, \tau\rangle]$;
- Check that $\uparrow S'\tau \equiv \mathsf{U}_i$.

Notice that the equations in $\mathcal{E}$ are built from atoms that are put into $\mathcal{A}$ by the annotation algorithm of Chapter 4.

By the semantics of unification with equality and the fact that S' extends S, it follows that

$$\langle S'B, S'\rho\rangle = \langle S'B, S'\sigma\rangle$$

and

$$\langle S'B, S'\uparrow\rho\rangle = \langle S'B, S'\tau\rangle.$$

Now H justifies B, $\mathcal{A}$, and $\mathcal{E}$ by construction; therefore H justifies $S'B$ as well since $\sigma^\circ \equiv (S\sigma)^\circ$ for any S and σ. Applying the semantics of typed terms, we obtain

- $H \gg (\uparrow S'\rho)^\circ = (\uparrow S'\sigma)^\circ \in \uparrow\uparrow S'\rho$;
- $H \gg (S'\rho)^\circ = (S'\sigma)^\circ \in (\uparrow S'\rho)^\circ$;
- $H \gg (\uparrow S'\rho)^\circ = (S'\tau)^\circ \in (\uparrow S'\tau)^\circ$.

It follows from the correctness of the Ann that $(S'\rho)^\circ \equiv \rho^\circ \equiv r$, and similarly for σ and τ, so we have

$$H \gg r = s \in T$$

and

$$H \gg T \in \mathsf{U}_i$$

as desired.

Consider again the example of the membership heuristic on page 87. There we observed that

$$H \gg \lambda(x.\lambda(y.\,\mathsf{apply}(\lambda(z.z); x))) \in \Pi(x{:}\,U_1.\Pi(y{:}\,x.x)) \,@\, 1$$

would be obtained by the heuristic because the type field of the annotated form of the λ-term,

$$\Pi(x{:}\,\alpha.\Pi(y{:}\,\beta.\alpha)\uparrow\mathsf{U}_1)\uparrow\mathsf{U}_1$$

unifies with the annotated form of the Π-type,

$$\Pi(x{:}\,\mathsf{U}_1.\Pi(y{:}\,x\uparrow\mathsf{U}_1.x\uparrow\mathsf{U}_1)\uparrow\mathsf{U}_1)\uparrow\mathsf{U}_1$$

under the substitution $\alpha \mapsto \mathsf{U}_1$ and $\beta \mapsto x\uparrow\mathsf{U}_1$. Notice that $B(\alpha)$!$\triangleright$ U_1 and $B(\beta)$!$\triangleright$ $x\uparrow\mathsf{U}_1$, so this is a valid substitution. Furthermore, we obtain the equation

$$H \gg \lambda(x.\lambda(y.\,\mathsf{apply}(\lambda(z.z);x))) = \lambda(x.\lambda(y.x)) \in \Pi(x{:}\,U_1.\Pi(y{:}\,x.x)) \,@\, 1$$

as follows. The annotated form of the right-hand side of the equation is

$$\lambda(x\uparrow\delta.\lambda(y\uparrow\mu.x\uparrow\delta))\uparrow\Pi(x{:}\,\delta.\Pi(y{:}\,\mu.\delta))\uparrow\mathsf{U}_1$$

This unifies with the annotated form of the left-hand side under the substitution $\{\,\delta \mapsto \alpha, \mu \mapsto \beta\,\}$, since the bodies of the innermost λ's are computationally equal. The types of the two terms are unifiable (in the extended sense), and they are in turn unifiable with the type of equation as before.

There are, of course, ways in which this algorithm might be improved. Perhaps a stronger, yet still tractable, test for equality can be found. Any improvement to this algorithm yields corresponding gains in both annotation and tests for equalty. The construction of $\mathcal{E}$ can also be improved — any number of ways of culling equations from H are acceptable (the arithmetic assistant of PRL is a likely candidate for such equations). As with the membership heuristic, we can hope only to provide an adequate, tunable framework for the construction of decision methods, relying on experience with it to suggest improvements.

Chapter 6

Conclusion

6.1 What has been accomplished?

This thesis concerns itself with problem of implementing a rich constructive logic in the framework of the PRL proof development environment. Constructing such a system is an immense task; I have attempted to solve only a few of the problems encountered in such an effort. The common theme among the problems that I have chosen is that each plays a key role in making ν-PRL viable. The theory of types is a very rich language for expressing problems in the domain of constructive mathematics (programming problems being only one such area). But the price of expressiveness is high: trivial syntactic well-formedness conditions in the world of predicate calculus expand into potentially non-trivial proofs of well-formedness in the world of type theory; types of terms can no longer be easily determined, nor is there even a notion of *the* type of a term; treatment of equality becomes considerably more complex when its domain includes more than integers and lists. These complications (and many others that we have not addressed) tend to limit the ease with which type theory can be used as a PRL base logic. Our mission has been to remedy the deficiencies of type theory and to provide a framework for the construction of mechanical proof assistants that can help to alleviate the burdens associated with the increase in generality.

Chapter 2 was devoted to organizing a set of inference rules for type theory that fit into the PRL refinement paradigm. The central concern was to formulate a logic which supports refinement and at the same time minimizes the burden of demonstrating the well-formedness of the theorems that the user sets out to refine. In type theory it is possible to designate a formula in very complex ways so that it is not at all obvious that a given term in fact is a formula. As a result it is not possible to give a decidable (let alone context-free) well-formedness condition for formulas, and so the user is forced to demonstrate that this is the case. Fortunately, we were able to structure the logic in such a way that well-formedness can be, for the most part, demonstrated simultaneously with truth, thereby alleviating much of the burden. However, there are still situations in which the user is forced to demonstrate well-formedness. Examples are the "other half" of a union type when doing a union introduction and the range type associated with the elimination rules. In practice it has turned out that it is often possible to decide these subgoals by mechanical means. This led to the decision to consider the construction of mechanical assistants for the type membership relation (of which well-formedness is a special case).

It often happens in the course of proving a theorem that it is necessary to derive trivial equality assertions which are then used to substitute equals for equals in some other term. Experience with PL/CV2 and λ-PRL shows that the equality decision procedure plays a crucial role in the proof development process. Without it, proof construction would be far too tedious to be seriously considered as a means of program development. With type theory equality has a wider spectrum than in either of these systems. The traditional distinction between individuals and formulas (equivalently, between objects and types) does not exist in type theory. One consequence of this is that there is a rich theory of equality on formulas as well as on "objects." This theory of equality includes the computation rules for type theory, thereby allowing for programs which compute formulas (types). Thus we were led to consider the problem of equality decision methods for type theory.

The equality relation for type theory is undecidable, and so we were led to consider heuristic methods which, it is hoped, will prove useful in practice. But even if attention is restricted to a decidable subrelation of equality, the problem remains quite difficult. The λ-PRL equality decision method depends critically on the ability to determine the types of terms by inspection. For instance, all numerals are of type integer, and all terms built with "." are of type list. Furthermore, the type of all functions is known in advance: either they are built-in or they are defined by primitive recursion or extraction before they are used. Therefore, the applicability of an equation is unlimited — if $f(3) = 7$, then *all* occurrences of $f(3)$ and 7 can be "merged" (treated as equal), and all equations can be propagated throughout the terms in which they occur.

This is not the case for type theory. From the point of view of the semantics of type theory, there is no reason whatever why any term is inherently a member of some particular type (or is a member of a class of types which are at all related). For instance, one could consistently extend the type theory with a type consisting of, just to be perverse, the numerals and the pairs. A given occurrence of a numeral might then be a member of the naturals or of this new type; one cannot tell just by looking at it. From the point of view of a fixed formalization of type theory, the situation is somewhat different. In the ν-PRL type theory there is in fact no such type — the numerals inhabit the type of natural numbers and no other. Therefore, a good heuristic is to suppose that a numeral is of type **nat**, and proceed from there. Furthermore, the type of a term is always constrained by the context of its occurrence. If a numeral occurs as the first argument of an **ind** form, then it must be a natural number (otherwise, the entire term cannot be well-formed according to the ν-PRL rules). Therefore, such contextual information can be exploited when trying to determine the type of an occurrence of a term.

Thus the problem of constructing a mechanical assistant for the equality relation depends on a mechanism for determining the types of occurrences of a term in order to know whether or not an equation applies. Since there is no such thing as *the* type of a term, we instead seek to find *a* type for the term which is correct both according to the semantics of type theory and according to the intentions of the user. Of course, the user's intentions are unfathomable in general, but one can hope to build an assistant which is reasonably good at making such guesses. The best that we can do is to provide a tunable framework for building such assistants and rely on experience to dictate the sorts of heuristics that might be brought to bear on the problem.

Since an equality heuristic depends on type determination, and since there is independent interest in a membership assistant, we are led to consider the problem of attaching types to occurrences of terms. In Chapter 3 we defined a logic of typed terms to address this problem. This formalization of type theory provides a formal account of the informal idea of attaching types to term occurrences. One may also view the logic of typed terms as a means of encoding derivations of type membership directly in a term. This view is the basis for the construction of a decision method for the equality relation. We also introduced typed term schemes in order to provide a semantic basis for a type determination algorithm based on satisfying contextual constraints.

In Chapter MembChap we addressed the problem of finding consistent type assignments for term occurrences. The process of attaching types to terms, annotation, is based on the considerations outlined above: there is no right answer, but any answer is constrained to be semantically correct and contextually consistent. For instance, it is semantically correct to take the type of a numeral to be **nat**; it is contextually inconsistent to take the type of a numeral to be **nat** if it is applied to a function whose domain is some other type. Building on the methods of Hindley and Milner, we constructed an annotation algorithm based on unification. Unification takes place with respect to a class of type variables, scheme variables which may be replaced by terms denoting types. Since the language of types is neither separable from the rest of the language of terms, nor is it a decidable subclass, we were led to a generalization of the unification computation that takes this into account: a type variable may be replaced by a term only if it can be determined to denote a type. Unification is defined on typed terms, and so it is simply a matter of inspection whether or not a typed term denotes a type. However, typed terms are built by annotation, so the likelihood of this discovering whether or not a term denotes a type is directly related to the effectiveness of the annotation algorithm.

The annotation and unification algorithms were then combined to build a simple, tunable proof assistant for the membership relation of type theory. By the semantics of typed terms, the annotation process amounts to the search for certain sorts of derivations of membership. Therefore, we can determine whether or not $s \in T$ by annotating s and T, then determining whether or not the type of s matches T and whether or not T denotes a type. When T is a universe, the annotation of T is trivial. Therefore checking the well-formedness of s (membership in some U_i) reduces to annotating of s and checking whether or not its type matches U_i.

Having formalized the notion of attaching types to term occurrences and having derived a means of determining types to attach to terms, we were then able to return to the problem of building a mechanical assistant for the equality relation. By exploiting the fact that all term constructors respect equality and that equality of types is codesignation, we defined a method of checking whether or not two typed terms are equal: check whether or not they are component–by–component equal. The equality relation is based on an arbitrary set of equations (which can be provided by the user) and on the computation rules for the various term constructors. This equality test can be built into the unification algorithm, thereby enhancing the annotation algorithm, the membership heuristic, and the equality heuristic by allowing a more liberal notion of matching to be used. Finally, a PRL proof assistant for equality can be built as follows: to test $r = s \in T$, annotate r, s, and T, and determine whether or not the annotated forms of r and s are equal, and whether or not their types are equal to the annotated form of T.

6.2 Directions for future research

There is no shortage of research problems generated by the PRL project, nor are the solutions to those problems presented in this thesis to be considered definitive. For those cases in which my ideas for future research are well–defined, they have been included in the discussion in the chapter in which the ideas are developed. In this section I will present ideas which are much less well–defined, and so should be regarded as speculative.

A very interesting problem which grows out of the work in Chapter 2 is the design of a more flexible proof editor. RED, the λ-PRL proof editor, is deficient in many respects. RED addresses the problem of systematic proof construction, but does not address the problem of proof modification. It is crucial to the success of a PRL–style development environment that derivation trees be much more plastic than they are at present. It seems desirable to provide editing operations that allow a proof tree to be inconsistent (and so is not a proof tree at all, but let us overlook this terminological problem). A useful model of proof editing appears to be one which allows for a wide range of structural modifications coupled with a collection of invariant restoration methods. The structural modifications would include ordinary text and tree manipulations (without regard to the interpretation of either by the system). The interesting part is the design of a suitable collection of invariant restoration methods. The motivation for this model is that a proof tree is defined as a tree of goals satisfying some collection of predicates, the proof tree invariants, which specify the conditions under which a tree is a valid proof tree. The task of RED, in conjunction with the user, is to coerce an arbitrary tree into a proof tree by the judicious use of invariant restoration methods.

Inference rules are simple cases of proof tree invariants. The subgoals of a given goal must match up with that goal in accordance with the rule whose name is attached to the goal. As it stands, the subgoals are generated so as to satisfy this invariant; it is not possible to present RED with a sequence of trees and ask whether or not these trees are (or can be made to) instantiate such–and–such a rule applied to such–and–such a goal. But if internal modifications to a proof tree are allowed, this is exactly the sort of question that RED must be able to answer. Furthermore, the user must be able, in general, to coax a tree into satisfying such an invariant. It is highly desirable that there be available a library of standard methods for doing this; otherwise, the process can be exceedingly tiresome. Furthermore, restoration of an invariant at one point of a tree may cause disruption of other invariants at other points in the tree. Some means of controlling this propagation must be devised.

It is interesting to note that extraction can be regarded as one such invariant–restoring operation. Let each node have associated with it a field which contains the extracted code for that node. Then extraction (which at present can only be applied to a sufficiently complete proof tree) is the process of restoring the invariants that must hold between the extracted code field of a node and those of its immediate descendants. If the extracted code invariant is updated after each major proof step (for some suitable definition of "major"), then extraction occurs dynamically as the proof tree is created. This allows for direct manipulation of extracted code without recourse to the use of **eq** types as employed in R.

The view of proof construction as invariant–restoration also provides for greater flexibility in the order in which proofs are developed. It seems to me that strict adherence to the refinement mode is undesirable, particularly in the PRL framework. My feeling is that the word "refinement" is more properly applied at the level of the problem statement rather than at the level of proof construction. There are simply too many details involved in building a valid proof tree to support the principle that refinement at the rule level corresponds to refinement at the problem level. For instance, the existential (dependent product) introduction rule requires that the witness object be built before the proof of the body can be conducted. This represents a forced sequencing that is inappropriate at the problem level. It is entirely possible, even likely, that the selectio of a witness cannot be done until the proof of the body has been at least partially conducted.

For instance, proofs of complexity bounds of algorithms are often of this form. To show that $T(n) = O(f(n))$, one must show that there is a c such that $T(n) \leq cf(n)$ for sufficiently large n. Typically, one does not pick the constant c, then prove the bound; rather, one starts to build the proof of the bound, selecting c as late as possible so that the proof can be completed. Since PRL is supposed to address the proof ***development*** process, it certainly must provide for this pattern of proof construction. A proof editor built on the model of restoration of invariants can easily provide this facility — the proof of the bound contains an expression e which, in a complete proof tree, is the value of c, the witness of the existential.

Another direction for research is the application of type theory to the study of type structure in other programming languages. Recent research on data types, modules, and abstraction mechanisms [MacQueen 84,Hook 84,Mitchell & Plotkin 85,Burstall & Lampson 84] tends to confirm my opinion that type theory is the right framework for discussing these issues. MacQueen's modules proposal for Standard ML has a very natural semantics in terms of the dependent function and product types of type theory. The universe structure combined with intensional operations on functions and types seems to be a promising framework for discussing the problem of providing an eval function for ML in ML. Recently Constable has addressed some of these issues [Constable 82a,Constable 84].

Another direction for research is in applying the PRL view of programs and types to a programming language such as ML. I think that it would be worthwhile to explore a stronger type structure for ML which lies somewhere between its current type system and what might be called a "full" specification. The ML type structure is very interesting because it provides a useful, but not annoying, correctness constraint on programs. I think that it would be interesting to build a program development environment for ML which addresses the problem of building type-correct programs under an extended notion of type. My experience with PRL indicates that such a system could be built, and that it would be possible to use it to build programs. The tools developed in Chapter 4 would replace the current notion of type inference for ML, functioning as an automated assistant, rather than an authoritative correctness check.

The extension of the type structure of ML could include the possibility of manipulating correct ML programs in ML via the intensional mechanisms of the type theory. This allows ML to be both the metalanguage and object language, and provides for the creation of provably-correct tactics for manipulating correctness proofs of ML programs (these issues will be explored in the forthcoming thesis of Knoblock [Knoblock 85]). This sort of reflective capability allows for the creation of a closed system of correct programs, much as LISP is a closed system of programs (sans correctness proofs). One of the virtues of LISP is that it is a self-contained, regular world. A similar environment for a semantically clean language like ML is an exciting prospect.

In the long run I see PRL as the beginning of a new paradigm for the use of interactive computer systems. Programming is a process of explanation, explaining why there is a solution to a certain problem, explaining how it is that this solution is organized. Ideally, PRL will serve as the infinitely-patient colleague, always willing to listen to one's half-baked ideas, criticizing the weak points, and suggesting ways to deal with the trivia. The more that one can concentrate on explanations, and the less on the bitter details, the better. PRL is the first step in the construction of such systems. I do not think that it was a false one.

Appendix A

The M Logic

This appendix consists of the rules for the **M** logic. The rules are organized into sections, one for each type constructor, and one section for the general rules.

VOID

Formation

$H \gg \mathsf{void} = \mathsf{void} \in \mathsf{U}_k$ by void form

Elimination

$H \gg \mathsf{any}(e) = \mathsf{any}(e') \in T[e/z]$ by void elim
  $H \gg e = e' \in \mathsf{void}$
  $H, z{:}\,void \gg T \in \mathsf{U}_k$

NATURAL NUMBERS

Formation

$H \gg \mathsf{nat} = \mathsf{nat} \in \mathsf{U}_k$ by nat form

Introduction

$H \gg 0 = 0 \in \mathsf{nat}$ by nat intro

$H \gg \sigma e = \sigma e' \in \mathsf{nat}$ by nat intro
  $H \gg e = e' \in \mathsf{nat}$

Elimination

$H \gg \mathsf{ind}(e; t_0; x.y.t_1) = \mathsf{ind}(e'; t_0'; x.y.t_1') \in T[e/z]$ by nat elim
  $H \gg e = e' \in \mathsf{nat}$
  $H \gg t_0 = t_0' \in T[0/z]$
  $H, x{:}\, \mathsf{nat}, y{:}\, T[x/z] \gg t_1 = t_1' \in T[\sigma x/z]$

Computation

$H \gg \mathsf{ind}(0; t_0; x.y.t_1) = t_0 \in T[0/z]$ by ind red
  $H \gg 0 \in \mathsf{nat}$
  $H \gg t_0 \in T[0/z]$
  $H, x{:}\, \mathsf{nat}, y{:}\, T[x/z] \gg t_1 \in T[\sigma x/z]$

$H \gg \mathsf{ind}(\sigma e; t_0; x.y.t_1) = t_1[e/x] \in T[\sigma e/z]$ by ind red
  $H \gg e \in \mathsf{nat}$
  $H \gg t_0 \in T[0/z]$
  $H, x{:}\, \mathsf{nat}, y{:}\, T[x/z] \gg t_1 \in T[\sigma x/z]$

UNION

Formation

$H \gg A + B = A' + B' \in \mathsf{U}_k$ by union form
$H \gg A = A' \in \mathsf{U}_k$
$H \gg B = B' \in \mathsf{U}_k$

Introduction

$H \gg \mathsf{inl}(a) = \mathsf{inl}(a') \in A + B$ by union intro
$H \gg a = a' \in A$
$H \gg B \in \mathsf{U}_k$

$H \gg \mathsf{inr}(b) = \mathsf{inr}(b') \in A + B$ by union intro
$H \gg b = b' \in B$
$H \gg A \in \mathsf{U}_k$

Elimination

$H \gg \mathsf{decide}(e; x.t_1; y.t_2) = \mathsf{decide}(e'; x.t_1'; y.t_2') \in T[e/z]$ by union elim
$H \gg e = e' \in A + B$
$H, x{:}\, A \gg t_1 = t_1' \in T[\mathsf{inl}(x)/z]$
$H, y{:}\, B \gg t_2 = t_2' \in T[\mathsf{inr}(y)/z]$
$H, z{:}\, A + B \gg T \in \mathsf{U}_k$

Computation

$H \gg \mathsf{decide}(\mathsf{inl}(a); x.t_1; y.t_2) = t_1[a/x] \in T[\mathsf{inl}(a)/z]$ by decide red
$H \gg \mathsf{inl}(a) \in A + B$
$H, x{:}\, A \gg t_1 \in T[\mathsf{inl}(x)/z]$
$H, y{:}\, B \gg t_2 \in T[\mathsf{inr}(y)/z]$
$H, z{:}\, A + B \gg T \in \mathsf{U}_k$

$H \gg \mathsf{decide}(\mathsf{inr}(b); x.t_1; y.t_2) = t_2[b/y] \in T[\mathsf{inr}(b)/z]$ by decide red
$H \gg \mathsf{inr}(b) \in A + B$
$H, x{:}\, A \gg t_1 \in T[\mathsf{inl}(x)/z]$
$H, y{:}\, B \gg t_2 \in T[\mathsf{inr}(y)/z]$
$H, z{:}\, A + B \gg T \in \mathsf{U}_k$

Type Structure

$H \gg A = A' \in \mathsf{U}_k$ by union structure
$H \gg A + B = A' + B' \in \mathsf{U}_k$

$H \gg B = B' \in \mathsf{U}_k$ by union structure
$H \gg A + B = A' + B' \in \mathsf{U}_k$

PRODUCT

Formation

$H \gg x{:}A \times B = x{:}A' \times B' \in \mathsf{U}_k$ by product form
$\quad H \gg A = A' \in \mathsf{U}_k$
$\quad H, x{:}\, A \gg B = B' \in \mathsf{U}_k$

Introduction

$H \gg \langle a\ b \rangle = \langle a'\ b' \rangle \in x{:}A \times B$ by product intro
$\quad H \gg a = a' \in A$
$\quad H \gg b = b' \in B[a/x]$
$\quad H, x{:}\, A \gg B \in \mathsf{U}_k$

Elimination

$H \gg \mathsf{spread}(e; x.y.t) = \mathsf{spread}(e'; x.y.t') \in T[e/z]$ by product elim
$\quad H \gg e = e' \in x{:}A \times B$
$\quad H, x{:}\, A, y{:}\, B \gg t = t' \in T[\langle x\ y \rangle / z]$
$\quad H, z{:}\, x{:}A \times B \gg T \in \mathsf{U}_k$

Computation

$H \gg \mathsf{spread}(\langle a\ b \rangle; x.y.t) = t[a, b/x, y] \in T[\langle a\ b \rangle / z]$ by spread red
$\quad H \gg \langle a\ b \rangle \in x{:}A \times B$
$\quad H, x{:}\, A, y{:}\, B \gg t \in T[\langle x\ y \rangle / z]$
$\quad H, z{:}\, x{:}A \times B \gg T \in \mathsf{U}_k$

Type Structure

$H \gg A = A' \in \mathsf{U}_k$ by product structure
$\quad H \gg x{:}A \times B = x{:}A' \times B' \in \mathsf{U}_k$

$H, x{:}\, A \gg B = B' \in \mathsf{U}_k$ by product structure
$\quad H \gg x{:}A \times B = x{:}A' \times B' \in \mathsf{U}_k$

FUNCTION

Formation

$H \gg x{:}A \to B = x{:}A' \to B' \in \mathsf{U}_k$ by arrow form
$\quad H \gg A = A' \in \mathsf{U}_k$
$\quad H, x{:}\, A \gg B = B' \in \mathsf{U}_k$

Introduction

$H \gg \lambda x.b = \lambda x.b' \in x{:}A \to B$ by function intro
$\quad H \gg A \in \mathsf{U}_k$
$\quad H, x{:}\, A \gg b = b' \in B$

Elimination

$H \gg \mathsf{apply}(f; a) = \mathsf{apply}(f'; a') \in B[a/x]$ by function elim
$\quad H \gg f = f' \in x{:}A \to B$
$\quad H \gg a = a' \in A$

Computation

$H \gg \mathsf{apply}(\lambda x.b; a) = b[a/x] \in B[a/x]$ by app red
$\quad H \gg a \in A$
$\quad H, x{:}\, A \gg b \in B$

Type Structure

$H \gg A = A' \in \mathsf{U}_k$ by function structure
$\quad H \gg x{:}A \to B = x{:}A' \to B' \in \mathsf{U}_k$

$H, x{:}\, A \gg B = B' \in \mathsf{U}_k$ by function structure
$\quad H \gg x{:}A \to B = x{:}A' \to B' \in \mathsf{U}_k$

SET

Formation

$H \gg \{x: A \mid B\} = \{x: A' \mid B'\} \in \mathsf{U}_k$ by set form
  $H \gg A = A' \in \mathsf{U}_k$
  $H, x: A \gg B = B' \in \mathsf{U}_k$

Introduction

$H \gg a = a' \in \{x: A \mid B\}$ by set intro
  $H \gg a = a' \in A$
  $H \gg e \in B[a/x]$
  $H, x: A \gg B \in \mathsf{U}_k$

Elimination

$H \gg t[a/z] = t'[a'/z] \in T[a/z]$ by set elim
  $H \gg a = a' \in \{x: A \mid B\}$
  $H, x: A, y: B \gg t[x/z] = t'[x/z] \in T[x/z]$ [*No y in t or t'.*]
  $H, z: \{x: A \mid B\} \gg T \in \mathsf{U}_k ext$

Type Structure

$H \gg A = A' \in \mathsf{U}_k$ by set structure
  $H \gg \{x: A \mid B\} = \{x: A' \mid B'\} \in \mathsf{U}_k$

$H, x: A \gg B = B' \in \mathsf{U}_k$ by set structure
  $H \gg \{x: A \mid B\} = \{x: A' \mid B'\} \in \mathsf{U}_k$

EQUALITY TYPE

Formation

$H \gg \mathsf{eq}(a;b;A) = \mathsf{eq}(a';b';A') \in \mathsf{U}_k$ by equality type form
$H \gg a = a' \in A$
$H \gg b = b' \in A$
$\dagger H \gg A = A' \in \mathsf{U}_k$

† *Reflexive instances of this subgoal may be omitted.*

Introduction

$H \gg \mathsf{axiom} \in \mathsf{eq}(a;b;A)$ by eq intro
$H \gg a = b \in A$

Elimination

$H \gg a = b \in A$ by eq elim
$H \gg e \in \mathsf{eq}(a;b;A)$

Type Structure

$H \gg a = b \in A$ by eq structure
$H \gg \mathsf{eq}(a;b;A) \in \mathsf{U}_k$

$H \gg A \in \mathsf{U}_k$ by eq structure
$H \gg \mathsf{eq}(a;b;A) \in \mathsf{U}_k$

GENERAL RULES

$H_1, x{:}\, A, H_2 \gg x \in A$ by hypothesis
  $H_1 \gg A \in \mathsf{U}_i$

$H \gg b[a/x] = b'[a'/x] \in B[a/x]$ by substitution
  $H \gg a = a' \in A$
  $H, x{:}\, A \gg b = b' \in B$

$H \gg a = b \in A$ by equality of types
  $H \gg A = B \in \mathsf{U}_i$
  $H \gg a = b \in B$

$H \gg \mathsf{U}_i = \mathsf{U}_i \in \mathsf{U}_j$ by universe form [$i < j$]

$H \gg A = A' \in \mathsf{U}_k$ by cumulativity [$j < k$]
  $H \gg A = A' \in \mathsf{U}_j$

Appendix B

The J logic

This appendix consists of the rules for the J logic. The rules are organized into sections, one for each type constructor, and one section for the general rules.

VOID

Formation

$H \gg \mathsf{void} = \mathsf{void} \in \mathsf{U}_k \mathbin{@} l$ by void form $[k < l]$

Elimination

$H \gg \mathsf{any}(e) = \mathsf{any}(e') \in T[e/z] \mathbin{@} k$ by void elim
$\quad H \gg e = e' \in \mathsf{void} \mathbin{@} i$
$\quad H, z{:}\, void \mathbin{@} i \gg T \in \mathsf{U}_k \mathbin{@} l$

NATURAL NUMBERS

Formation

$H \gg \mathsf{nat} = \mathsf{nat} \in \mathsf{U}_k$ @ l by nat form $[k < l]$

Introduction

$H \gg 0 = 0 \in \mathsf{nat}$ @ k by nat intro

$H \gg \sigma e = \sigma e' \in \mathsf{nat}$ @ k by nat intro
  $H \gg e = e' \in \mathsf{nat}$ @ k

Elimination

$H \gg \mathsf{ind}(e; t_0; x.y.t_1) = \mathsf{ind}(e'; t_0'; x.y.t_1') \in T[e/z]$ @ k by nat elim
  $H \gg e = e' \in \mathsf{nat}$ @ i
  $H \gg t_0 = t_0' \in T[0/z]$ @ k
  $H, x\colon \mathsf{nat}$ @ $i, y\colon T[x/z]$ @ $k \gg t_1 = t_1' \in T[\sigma x/z]$ @ k
  $H, z\colon \mathsf{nat}$ @ $i \gg T \in U_k$ @ l

Computation

$H \gg \mathsf{ind}(0; t_0; x.y.t_1) = t_0 \in T[0/z]$ @ k by ind red
  $H \gg 0 \in \mathsf{nat}$ @ i
  $H \gg t_0 \in T[0/z]$ @ k
  $H, x\colon \mathsf{nat}$ @ $i, y\colon T[x/z]$ @ $k \gg t_1 \in T[\sigma x/z]$ @ k
  $H, z\colon \mathsf{nat}$ @ $i \gg T \in U_k$ @ l

$H \gg \mathsf{ind}(\sigma e; t_0; x.y.t_1) = t_1[e/x] \in T[\sigma e/z]$ @ k by ind red
  $H \gg e \in \mathsf{nat}$ @ i
  $H \gg t_0 \in T[0/z]$ @ k
  $H, x\colon \mathsf{nat}$ @ $i, y\colon T[x/z]$ @ $k \gg t_1 \in T[\sigma x/z]$ @ k
  $H, z\colon \mathsf{nat}$ @ $i \gg T \in U_k$ @ l

UNION

Formation

$H \gg A+B = A'+B' \in \mathsf{U}_k \text{ @ } l$ by union form $[k<l]$
$\quad H \gg A = A' \in \mathsf{U}_k \text{ @ } l$
$\quad H \gg B = B' \in \mathsf{U}_k \text{ @ } l$

Introduction

$H \gg \mathsf{inl}(a) = \mathsf{inl}(a') \in A+B \text{ @ } k$ by union intro
$\quad H \gg a = a' \in A \text{ @ } k$
$\quad H \gg B \in \mathsf{U}_k \text{ @ } l$

$H \gg \mathsf{inr}(b) = \mathsf{inr}(b') \in A+B \text{ @ } k$ by union intro
$\quad H \gg b = b' \in B \text{ @ } k$
$\quad H \gg A \in \mathsf{U}_k \text{ @ } l$

Elimination

$H \gg \mathsf{decide}(e; x.t_1; y.t_2) = \mathsf{decide}(e'; x.t_1'; y.t_2') \in T[e/z] \text{ @ } k$ by union elim
$\quad H \gg e = e' \in A+B \text{ @ } i$
$\quad H, x{:}\, A \text{ @ } i \gg t_1 = t_1' \in T[\mathsf{inl}(x)/z] \text{ @ } k$
$\quad H, y{:}\, B \text{ @ } i \gg t_2 = t_2' \in T[\mathsf{inr}(y)/z] \text{ @ } k$
$\quad H, z{:}\, A+B \text{ @ } i \gg T \in \mathsf{U}_k \text{ @ } l$

Computation

$H \gg \mathsf{decide}(\mathsf{inl}(a); x.t_1; y.t_2) = t_1[a/x] \in T[\mathsf{inl}(a)/z] \text{ @ } k$ by decide red
$\quad H \gg \mathsf{inl}(a) \in A+B \text{ @ } i$
$\quad H, x{:}\, A \text{ @ } i \gg t_1 \in T[\mathsf{inl}(x)/z] \text{ @ } k$
$\quad H, y{:}\, B \text{ @ } i \gg t_2 \in T[\mathsf{inr}(y)/z] \text{ @ } k$
$\quad H, z{:}\, A+B \text{ @ } i \gg T \in \mathsf{U}_k \text{ @ } l$

$H \gg \mathsf{decide}(\mathsf{inr}(b); x.t_1; y.t_2) = t_2[b/y] \in T[\mathsf{inr}(b)/z] \text{ @ } k$ by decide red
$\quad H \gg \mathsf{inr}(b) \in A+B \text{ @ } i$
$\quad H, x{:}\, A \text{ @ } i \gg t_1 \in T[\mathsf{inl}(x)/z] \text{ @ } k$
$\quad H, y{:}\, B \text{ @ } i \gg t_2 \in T[\mathsf{inr}(y)/z] \text{ @ } k$
$\quad H, z{:}\, A+B \text{ @ } i \gg T \in \mathsf{U}_k \text{ @ } l$

Type Structure

$$\frac{H \gg A = A' \in \mathsf{U}_k \ @\ l \quad \text{by union structure}}{H \gg A + B = A' + B' \in \mathsf{U}_k \ @\ l}$$

$$\frac{H \gg B = B' \in \mathsf{U}_k \ @\ l \quad \text{by union structure}}{H \gg A + B = A' + B' \in \mathsf{U}_k \ @\ l}$$

PRODUCT

Formation

$H \gg x{:}A \times B = x{:}A' \times B' \in \mathsf{U}_k \text{ @ } l$ by product form $[k < l]$
  $H \gg A = A' \in \mathsf{U}_k \text{ @ } l$
  $H, x{:}\, A \text{ @ } k \gg B = B' \in \mathsf{U}_k \text{ @ } l$

Introduction

$H \gg \langle a\ b\rangle = \langle a'\ b'\rangle \in x{:}A \times B \text{ @ } k$ by product intro
  $H \gg a = a' \in A \text{ @ } k$
  $H \gg b = b' \in B[a/x] \text{ @ } k$
  $H, x{:}\, A \text{ @ } k \gg B \in \mathsf{U}_k \text{ @ } l$

Elimination

$H \gg \mathsf{spread}(e; x.y.t) = \mathsf{spread}(e'; x.y.t') \in T[e/z] \text{ @ } k$ by product elim
  $H \gg e = e' \in x{:}A \times B \text{ @ } i$
  $H, x{:}\, A \text{ @ } i, y{:}\, B \text{ @ } i \gg t = t' \in T[\langle x\ y\rangle/z] \text{ @ } k$
  $H, z{:}\, x{:}A \times B \text{ @ } i \gg T \in \mathsf{U}_k \text{ @ } l$

Computation

$H \gg \mathsf{spread}(\langle a\ b\rangle; x.y.t) = t[a, b/x, y] \in T[\langle a\ b\rangle/z] \text{ @ } k$ by spread red
  $H \gg \langle a\ b\rangle \in x{:}A \times B \text{ @ } i$
  $H, x{:}\, A \text{ @ } i, y{:}\, B \text{ @ } i \gg t \in T[\langle x\ y\rangle/z] \text{ @ } k$
  $H, z{:}\, x{:}A \times B \text{ @ } i \gg T \in \mathsf{U}_k \text{ @ } l$

Type Structure

$H \gg A = A' \in \mathsf{U}_k \text{ @ } l$ by product structure
  $H \gg x{:}A \times B = x{:}A' \times B' \in \mathsf{U}_k \text{ @ } l$

$H, x{:}\, A \text{ @ } k \gg B = B' \in \mathsf{U}_k \text{ @ } l$ by product structure
  $H \gg x{:}A \times B = x{:}A' \times B' \in \mathsf{U}_k \text{ @ } l$

FUNCTION

Formation

$H \gg x{:}A \to B = x{:}A' \to B' \in \mathsf{U}_k \text{ @ } l$ by arrow form $[k < l]$
$\quad H \gg A = A' \in \mathsf{U}_k \text{ @ } l$
$\quad H, x{:}\, A \text{ @ } k \gg B = B' \in \mathsf{U}_k \text{ @ } l$

Introduction

$H \gg \lambda x.b = \lambda x.b' \in x{:}A \to B \text{ @ } k$ by function intro
$\quad H \gg A \in \mathsf{U}_k \text{ @ } l$
$\quad H, x{:}\, A \text{ @ } k \gg b = b' \in B \text{ @ } k$

Elimination

$H \gg \mathsf{apply}(f; a) = \mathsf{apply}(f'; a') \in B[a/x] \text{ @ } k$ by function elim
$\quad H \gg f = f' \in x{:}A \to B \text{ @ } k$
$\quad H \gg a = a' \in A \text{ @ } k$

Computation

$H \gg \mathsf{apply}(\lambda x.b; a) = b[a/x] \in B[a/x] \text{ @ } k$ by app red
$\quad H \gg a \in A \text{ @ } k$
$\quad H, x{:}\, A \text{ @ } k \gg b \in B \text{ @ } k$

Type Structure

$H \gg A = A' \in \mathsf{U}_k \text{ @ } l$ by function structure
$\quad H \gg x{:}A \to B = x{:}A' \to B' \in \mathsf{U}_k \text{ @ } l$

$H, x{:}\, A \text{ @ } k \gg B = B' \in \mathsf{U}_k \text{ @ } l$ by function structure
$\quad H \gg x{:}A \to B = x{:}A' \to B' \in \mathsf{U}_k \text{ @ } l$

SET

Formation

$H \gg \{x: A \mid B\} = \{x: A' \mid B'\} \in \mathsf{U}_k \mathbin{@} l$ by set form $[k < l]$
  $H \gg A = A' \in \mathsf{U}_k \mathbin{@} l$
  $H, x: A \mathbin{@} k \gg B = B' \in \mathsf{U}_k \mathbin{@} l$

Introduction

$H \gg a = a' \in \{x: A \mid B\} \mathbin{@} k$ by set intro
  $H \gg a = a' \in A \mathbin{@} k$
  $H \gg e \in B[a/x] \mathbin{@} k$
  $H, x: A \mathbin{@} k \gg B \in \mathsf{U}_k \mathbin{@} l$

Elimination

$H \gg t[a/z] = t'[a'/z] \in T[a/z] \mathbin{@} k$ by set elim
  $H \gg a = a' \in \{x: A \mid B\} \mathbin{@} i$
  $H, x: A \mathbin{@} i, y: B \mathbin{@} i \gg t[x/z] = t'[x/z] \in T[x/z] \mathbin{@} k$ [*No* y *in* t *or* t'.]
  $H, z: \{x: A \mid B\} \mathbin{@} i \gg T \in \mathsf{U}_k \mathbin{@} lext$

Type Structure

$H \gg A = A' \in \mathsf{U}_k \mathbin{@} l$ by set structure
  $H \gg \{x: A \mid B\} = \{x: A' \mid B'\} \in \mathsf{U}_k \mathbin{@} l$

$H, x: A \mathbin{@} i \gg B = B' \in \mathsf{U}_k \mathbin{@} l$ by set structure
  $H \gg \{x: A \mid B\} = \{x: A' \mid B'\} \in \mathsf{U}_k \mathbin{@} l$

EQUALITY TYPE

Formation

$H \gg \mathsf{eq}(a; b; A) = \mathsf{eq}(a'; b'; A') \in \mathsf{U}_k \circledcirc l$ by equality type form $[k < l]$
$\quad H \gg a = a' \in A \circledcirc k$
$\quad H \gg b = b' \in A \circledcirc k$
$\quad \dagger H \gg A = A' \in \mathsf{U}_k \circledcirc l$

† *Reflexive instances of this subgoal may be omitted.*

Introduction

$H \gg \mathsf{axiom} \in \mathsf{eq}(a; b; A) \circledcirc k$ by eq intro
$\quad H \gg a = b \in A \circledcirc k$

Elimination

$H \gg a = b \in A \circledcirc k$ by eq elim
$\quad H \gg e \in \mathsf{eq}(a; b; A) \circledcirc k$

Type Structure

$H \gg a = b \in A \circledcirc k$ by eq structure
$\quad H \gg \mathsf{eq}(a; b; A) \in \mathsf{U}_k \circledcirc l$

$H \gg A \in \mathsf{U}_k \circledcirc l$ by eq structure
$\quad H \gg \mathsf{eq}(a; b; A) \in \mathsf{U}_k \circledcirc l$

GENERAL RULES

$H_1, x{:}\, A \,@\, i, H_2 \gg x \in A \,@\, i$ by hypothesis

$H \gg b[a/x] = b'[a'/x] \in B[a/x] \,@\, k$ by substitution
  $H \gg a = a' \in A \,@\, i$
  $H, x{:}\, A \,@\, i \gg b = b' \in B \,@\, k$

$H \gg a = b \in A \,@\, i$ by equality of types
  $H \gg A = B \in \mathsf{U}_i \,@\, k$
  $H \gg a = b \in B \,@\, k$

$H \gg \mathsf{U}_i = \mathsf{U}_i \in \mathsf{U}_j \,@\, k$ by universe form $[i < j < k]$

$H \gg A = A' \in \mathsf{U}_k \,@\, l$ by cumulativity $[j < k < l]$
  $H \gg A = A' \in \mathsf{U}_j \,@\, i$

$H \gg A \in \mathsf{U}_i \,@\, k$ by inhabitation $[i < k]$
  $H \gg a \in A \,@\, i$

Appendix C

The R logic

This appendix consists of the rules for the R logic. The rules are organized into sections, one for each type constructor, and one section for the general rules.

VOID

Formation

$H \gg \mathsf{U}_k \mathbin{@} l$ *ext* void by void formation[$k < l$]

Elimination

$H_1, z{:}\,\mathsf{void} \mathbin{@} i, H_2 \gg T \mathbin{@} k$ *ext* any(z) by elim on z
$H_1, z{:}\,\mathsf{void} \mathbin{@} i, H_2 \gg T \text{ in } \mathsf{U}_k \mathbin{@} l$

NATURAL NUMBERS

Formation

$H \gg \mathsf{U}_k$ @ l *ext* nat by void formation$[k < l]$

Introduction

$H \gg$ nat @ k *ext* 0 by intro 0

$H \gg$ nat @ k *ext* σe by intro σe
$H \gg e$ in nat @ k

Elimination

H, z: nat @ $i \gg T$ @ k *ext* $\mathsf{ind}(z; t_0; x.y.t_1)$ by elim on z
H, z: nat @ $i \gg T[0/z]$ @ k *ext* t_0
H, z: nat @ i, x: nat @ i, y: $T[x/z]$ @ $k \gg T[\sigma x/z]$ @ k *ext* t_1
H, z: nat @ $i \gg T$ in U_k @ l

UNION

Formation

$H \gg \mathsf{U}_k \ @\ l \quad ext\ A + B \quad$ by union formation$[k < l]$
$\quad H \gg \mathsf{U}_k \ @\ l \quad ext\ A$
$\quad H \gg \mathsf{U}_k \ @\ l \quad ext\ B$

Introduction

$H \gg A + B \ @\ k \quad ext\ \mathsf{inl}(a) \quad$ by intro left
$\quad H \gg A \ @\ k \quad ext\ a$
$\quad H \gg B \text{ in } \mathsf{U}_k \ @\ l$

$H \gg A + B \ @\ k \quad ext\ \mathsf{inr}(b) \quad$ by intro right
$\quad H \gg B \ @\ k \quad ext\ b$
$\quad H \gg A \text{ in } \mathsf{U}_k \ @\ l$

Elimination

$H_1, z{:}\ A + B \ @\ i, H_2 \gg T \ @\ k \quad ext\ \mathsf{decide}(z; x.t_1; y.t_2) \quad$ by elim on z
$\quad H_1, z{:}\ A + B \ @\ i, H_2, x{:}\ A \ @\ i \gg T[\mathsf{inl}(x)/z] \ @\ k \quad ext\ t_1$
$\quad H_1, z{:}\ A + B \ @\ i, H_2, y{:}\ B \ @\ i \gg T[\mathsf{inr}(y)/z] \ @\ k \quad ext\ t_2$
$\quad H_1, z{:}\ A + B \ @\ i, H_2, z{:}\ A + B \ @\ i \gg T \text{ in } \mathsf{U}_k \ @\ l$

PRODUCT

Formation

$H \gg \mathsf{U}_k \mathbin{@} l \quad ext\ x{:}A \times B$ by product formation $A[k < l]$
$\quad H \gg A \text{ in } \mathsf{U}_k \mathbin{@} l$
$\quad H, x{:}\, A \mathbin{@} k \gg \mathsf{U}_k \mathbin{@} l \quad ext\ B$

Introduction

$H \gg x{:}A \times B \mathbin{@} k \quad ext\ \langle a\ b\rangle$ by intro a
$\quad H \gg a \text{ in } A \mathbin{@} k$
$\quad H \gg B[a/x] \mathbin{@} k \quad ext\ b$
$\quad H, x{:}\, A \mathbin{@} k \gg B \text{ in } \mathsf{U}_k \mathbin{@} l$

Elimination

$H_1, z{:}\, (x{:}A \times B) \mathbin{@} i, H_2 \gg T \mathbin{@} k \quad ext\ \mathsf{spread}(z; x.y.t)$ by elim on z
$\quad H_1, z{:}\, (x{:}A \times B) \mathbin{@} i, H_2, x{:}\, A \mathbin{@} i, y{:}\, B \mathbin{@} i \gg T[\langle x\ y\rangle/z] \mathbin{@} k \quad ext\ t$
$\quad H_1, z{:}\, (x{:}A \times B) \mathbin{@} i, H_2, z{:}\, (x{:}A \times B) \gg T \text{ in } U_k \mathbin{@} l$

FUNCTION

Formation

$H \gg \mathsf{U}_k \circledcirc l \quad ext\ x{:}A \to B \quad$ by function formation $A[k < l]$
$\quad H \gg A \circledcirc k$ in $\mathsf{U}_k \circledcirc l$
$\quad H, x{:}\, A \circledcirc k \gg \mathsf{U}_k \circledcirc l \quad ext\ B$

Introduction

$H \gg x{:}A \to B \circledcirc k \quad ext\ \lambda x.b \quad$ by intro
$\quad H \gg A$ in $\mathsf{U}_k \circledcirc l$
$\quad H, x{:}\, A \circledcirc k \gg B \circledcirc k \quad ext\ b$

Elimination

$H_1, f{:}\,(x{:}A \to B) \circledcirc k, H_2 \gg B[a/x] \circledcirc k \quad ext$ apply$(f; a)$ by elim on f applied to a
$\quad H_1, f{:}\,(x{:}A \to B) \circledcirc k, H_2 \gg a$ in $A \circledcirc k$

SET

Formation

$H \gg \mathsf{U}_k \mathbin{@} l \quad ext\ \{x{:}\, A \mid B\}$ by set formation $A[k < l]$
$\quad H \gg A \text{ in } \mathsf{U}_k \mathbin{@} l$
$\quad H, x{:}\, A \mathbin{@} k \gg \mathsf{U}_k \mathbin{@} l \quad ext\ B$

Introduction

$H \gg \{x{:}\, A \mid B\} \mathbin{@} k \quad ext\ a$ by intro a
$\quad H \gg a \text{ in } A \mathbin{@} k$
$\quad H \gg B[a/x] \mathbin{@} k$
$\quad H, x{:}\, A \mathbin{@} k \gg B \text{ in } \mathsf{U}_k \mathbin{@} l$

Elimination

$H_1, z{:}\, \{x{:}\, A \mid B\} \mathbin{@} i, H_2 \gg T \mathbin{@} k \quad ext\ t[z/x]$ by elim on z
$\quad H_1, z{:}\, \{x{:}\, A \mid B\} \mathbin{@} i, H_2, x{:}\, A \mathbin{@} i, y{:}\, B \mathbin{@} i \gg T[x/z] \mathbin{@} k \quad ext\ t \quad [No\ y\ in\ t]$
$\quad H_1, z{:}\, \{x{:}\, A \mid B\} \mathbin{@} i, H_2, z{:}\, \{x{:}\, A \mid B\} \mathbin{@} i \gg T \text{ in } U_k \mathbin{@} l$

EQUALITY TYPE

Formation

$H \gg (a = b \text{ in } A) = (a' = b' \text{ in } A') \text{ in } \mathsf{U}_k \text{ @ } l$ by formation[$k < l$]
  $H \gg a = a' \text{ in } A \text{ @ } k$
  $H \gg b = b' \text{ in } A \text{ @ } k$† $H \gg A = A' \text{ in } \mathsf{U}_k \text{ @ } l$

† *Reflexive instances of this subgoal may be omitted.*

Introduction

$H \gg \text{axiom in } (a = a' \text{ in } A) \text{ @ } k$ by intro
  $H \gg a = a' \text{ in } A \text{ @ } k$

Elimination

$H \gg a = a' \text{ in } A \text{ @ } k$ by elim
  $H \gg e \text{ in } (a = a' \text{ in } A) \text{ @ } k$

GENERAL RULES

$H_1, x{:}\, A @ i, H_2 \gg A @ i$ *ext* x by hypothesis

$H \gg B[a, b/x, y] @ k$ *ext* $b[a, b/x, y]$ by substitution $a = b$ in A over $B_{x,y}$
$H \gg a = b$ in $A @ i$
$H, x{:}\, A @ i, y{:}\, A @ i, u{:}\, \mathrm{eq}(x, y, A) @ i \gg B$ *ext* b

$H \gg A @ i$ *ext* a by equal types B
$H \gg A = B$ in $\mathrm{U}_i @ k$
$H \gg B @ i$ *ext* a

$H \gg \mathrm{U}_j @ k$ *ext* U_i by universe i form $[i < j < k]$

$H \gg \mathrm{U}_k @ l$ *ext* A by cumulativity from j $[j < k < l]$
$H \gg \mathrm{U}_j @ i$ *ext* A

$H \gg A @ k$ *ext* a by cumulativity from j$[j < k]$
$H \gg A @ j$ *ext* a

$H \gg A @ k$ *ext* a by explicit introduction a
$H \gg a$ in $A @ k$

$H \gg A$ in $\mathrm{U}_i @ k$ by inhabitation$[i < k]$
$H \gg A @ i$

Bibliography

[Allen 85]
Stuart F. Allen.
The Logical Foundations of Type Theory.
Ph.D. Thesis (forthcoming), Cornell University, 1985.

[Barendregt 81]
Henk P. Barendregt.
The λ-Calculus: Its Syntax and Semantics.
North-Holldand, 1981.

[Bates 79]
Joseph L. Bates.
A Logic for Correct Program Development.
Ph.D. Thesis, Cornell University, 1979.

[Bates & Constable 81]
Joseph L. Bates and Robert L. Constable.
The Definition of μ-PRL.
Dept. of Computer Science Technical Report TR 82–492.
Cornell University, Ithaca, NY 1981.

[Burstall & Lampson 84]
Butler Lampson and Rod Burstall.
A Kernel Language for Abstract Data Types and Modules.
In *Semantics of Data Types*, G. Kahn, D.B. MacQueen, and G. Plotkin, eds., pp. 1–50.
Lecture Notes in Computer Science vol. 173., Springer–Verlag, 1984.

[Bates & Constable 83]
Joseph L. Bates and Robert L. Constable.

The Nearly-Ultimate PRL.
Dept. of Computer Science Technical Report TR 83–551.
Cornell University, Ithaca, NY 1983.

[Bates & Constable 85]
Joseph L. Bates and Robert L. Constable.
Proofs as Programs.
ACM Trans. on Prog. Lang. and Sys., v.7, n.1 (January 1985), pp. 113–136.

[Bell & Machover 77]
A Course in Mathematical Logic.
North-Holland, 1977.

[Bishop 67]
Errett Bishop.
Foundations of Constructive Analysis.
McGraw-Hill, 1967.

[Bishop 70]
Errett Bishop.
Mathematics as a Numerical Language.
In *Intuitionism and Proof Theory*, J. Myhill, *et. al.*, eds., pp. 53–71.
North-Holland, 1970.

[Boyer & Moore 79]
A Computational Logic.
Academic Press, 1979.

[Brouwer 23]
L. E. J. Brouwer.
On the significance of the principle of the excluded middle in mathematics, especially in function theory.
In [vanHeijenoort 67], pp. 334–345.

[deBruijn 72]
deBruijn, N. G.
Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem.

Koninluke Nederlandse Akademie van Wetenschappen,
Proc. Series A, v.75, n.5 (1972), pp. 381–392.

[deBruijn 78]
N. G. deBruijn.
Lambda Calculus with Namefree Formulas Involving Symbols that Represent Reference Transforming Mappings.
Koninluke Nederlandse Akademie van Wetenschappen,
Proc. Series A, v.81, n.3 (1978), pp. 348–356.

[deBruijn 80]
A Survey of the Project AUTOMATH.
In *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley, eds., pp. 589–606.
Academic Press, 1980.

[Cartwright 76]
Robert Cartwright.
User-defined Data Types as an Aid to Verifying LISP Programs.
Stanford University A. I. Memo AIM–296, 1976.

[Constable 71]
Rober L. Constable.
Constructive Mathematics and Automatic Programs Writers.
In *Proc. of IFIP Congress*, pp. 229–233.
Ljubljana, 1971.

[Constable 80]
Robert L. Constable.
Programs and Types.
In *Proc. 21st IEEE Symp. on the Foundations of Computer Science*, pp. 118–128, 1980.

[Constable 82a]
Robert L. Constable.
Intensional Analysis of Functions and Types.
Dept. of Computer Science Technical Report CSR 118–82.
University of Edinburgh, 1982.

[Constable 82b]
Robert L. Constable.
Formalizing Metamathematics in Type Theory.
Dept. of Computer Science, University of Edinburgh, 1982.

[Constable 82c]
Robert L. Constable.
Programs as Proofs.
Information Processing Letters, v.16, n.3 (April 15, 1982), pp. 105–112.

[Constable 83a]
Robert L. Constable.
Constructive Mathematics as a Programming Logic I: Some Principles of Theory.
Annals of Discrete Mathematics, v.24 (1985), pp. 21–38.

[Constable 83b]
Robert L. Constable.
Mathematics as Programming.
In *Proc. of Workshop on Logics of Programs*, pp. 116–128.
Lecture Notes in Computer Science vol. 164, Springer–Verlag, 1983.

[Constable 84]
Robert L. Constable.
Universally Closed Classes of Total Computable Functions.
Dept. of Computer Science Technical Report TR 84–640.
Cornell University, Ithaca, NY Spetember 1984.

[Constable, Johnson, & Eichenlaub 82]
Robert L. Constable, Scott D. Johnson, and Carl D. Eichenlaub.
Introduction to the PLC/V2 Programming Logic.
Lecture Notes in Computer Science vol. 135, Springer–Verlag, 1982.

[Constable, Knoblock, & Bates 84]
Robert L. Constable, Todd Knoblock, and Joseph L. Bates.
On Writing Programs to Find Proofs.
Dept. of Computer Science Technical Report TR 84–645.
Cornell University, Ithaca, NY 1984.
(to appear in *Journal of Automated Reasoning*)

[Constable & Zlatin 84]
Robert L. Constable and Daniel R. Zlatin.
The Type Theory of PL/CV3.
ACM Trans. on Prog. Lang. and Sys., v.6, n.1 (January 1984), pp. 97–117.

[Curry 1958]
H. B. Curry, R. Feys, and W. Craig.
Combinatory Logic, Volume 1.
North–Holland, 1958.

[Curry 72]
H. B. Curry, J. R. Hindley, and J. P. Seldin.
Combinatory Logic, Volume 2.
North–Holland, 1972.

[Damas & Milner 82]
Luis Damas and Robin Milner.
Principal Type Schemes for Functional Programs.
In *Proc. 9th ACM Symp. on Princ. of Prog. Lang.*, pp. 207–212, 1982.

[Dummet 77]
Michael Dummett.
Elements of Intuitionism.
Oxford Logic Series, Clarendon Press, 1977.

[Frege 1879]
Gottlob Frege.
Begriffsschrift, A Formula Language, Modeled Upon that for Arithmetic, for Pure Thought.
In [vanHeijenoort 67], pp. 1–82.

[Gentzen 35]
Gerhard Gentzen.
Investigations Into Logical Deduction.
In *The Collected Papers of Gerhard Gentzen*, M. .E. Szabo, ed.
North–Holland, Amsterdam, 1969.

[ADJ 78]
J. A. Goguen, J. W. Thatcher, and E. G. Wagner.

An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types.
In *Current Trends in Programming Methodology, vol. 4*, R. Yeh, ed.
Prentice–Hall, 1978.

[Gordon, Milner, & Wadsworth 79]
Michael Gordon, Arthur Milner, and Christopher Wadsworth.
Edinburgh LCF: A Mechanized Logic of Computation.
Lecture Notes in Computer Science vol. 78, Springer–Verlag, 1979.

[Gries 81]
David Gries.
The Science of Programming.
Springer–Verlag, 1981.

[vanHeijenoort 67]
J. van Heijenoort.
From Frege to Gödel: A Sourcebook in Mathematical Logic.
Harvard University Press, Cambridge, 1967.

[Heyting 56]
Arend Heyting.
Intuitionism: An Introduction
North–Holland, 1956.

[Hindley 69]
J. R. Hindley.
The Principle Type Scheme of an Object in Combinatory Logic.
Trans. of the American Mathematical Society n.146 (1969), pp. 29–40.

[Hoffmann & O'Donnell 84]
Christoph M. Hoffman and Michael J. O'Donnell.
Implementation of an Interpreter for Abstract Equations.
In *Proc. 11th ACM Symp. on Princ. of Prog. Lang.*, 1984, pp. 111–121.

[Hook 84]
James G. Hook.
Understanding Russell: A First Attempt.
In *Semantics of Data Types*, G. Kahn, *et. al.*, eds., pp. 69–85.
Lecture Notes in Computer Science vol. 173., Springer–Verlag, 1984.

[Howard 80]
W. Howard.
The Formulas–as–Types Notion of Construction.
In *To H. B. Curry: Essays on Combinatory Logic, Lambda–Calculus, and Formalism*, J. P. Seldin and J. R. Hindley, eds., pp. 479–490. Academic Press, 1980.

[Huet 80]
Confluent reductions: abstract properties and applications to term rewriting systems.
J. ACM v.27, n.4 (October 1979), pp. 797–821.

[Johnson 80]
Scott Johnson.
A Computer System for Checking Proofs.
Ph.D. Thesis, Cornell University, 1980.

[Kleene 52]
Stephen Kleene.
Introduction to Metamathematics.
D. van Nostrand, Princeton, 1952.

[Knoblock 85]
Todd Knoblock.
A Unified Metalanguage and Object Language for Constructive Type Theory.
Ph.D. Thesis (forthcoming), Cornell University, 1985.

[Krafft 81]
Dean B. Krafft.
AVID: A System for the Interactive Development of Verifiably Correct Programs.
Ph.D. Thesis, Cornell University, 1981.

[Leivant 83]
Daniel Leivant.
Polymorphic Type Inference.
In *Proc. 10th ACM Symp. on Princ. of Prog. Lang.*, pp. 88–98, 1983.

[MacQueen 84]
David MacQueen.
Modules for Standard ML.
Unpublished manuscript, December, 1984.

[Martin-Löf 73]
Per Martin-Löf.
An Intuitionistic Theory of Types: Predicative Part.
In *Logic Colloquium '73*, H. E. Rose and J. C. Shepherdson, eds., pp. 73–118.
North-Holland, 1973.

[Martin-Löf 82]
Per Martin-Löf.
Constructive Mathematics and Computer Programming.
In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pp. 153–175.
North-Holland, 1982.

[Milner 78]
Robin Milner.
A Theory of Type Polymorphism in Programming Languages.
J. Computer and System Sciences v.17 (1978), pp. 348–375.

[Mitchell & Plotkin 85]
Abstract types have existential type.
In *Proc. 12th ACM Symp. on Princ. of Prog. Lang.*, pp. 37–51, 1985.

[Nelson 81]
Greg Nelson.
Techniques for Program Verification.
Technical Report CSL–81–10,
Xerox Palo Alto Research Center, 1981.

[Nelson & Oppen 79]
Greg Nelson and Derek C. Oppen.
Simplification by Cooperating Decision Procedures.
ACM Trans. on Prog. Lang. and Sys. v.1, n.2 (October 1979), pp. 245–257.

[Nelson & Oppen 80]
Greg Nelson and Derek C. Oppen.
Fast Decision Procedures Based on Congruence Closure.
J. ACM v.27, n.2 (April 1980), pp. 356–364.

[Nordstrom 81]
Bengt Nordstrom.
Programming in Constructive Set Theory: Some Examples.
In *Proc. of the 1981 Conference on Functional Programming Languages and Computer Architecture.*
Portsmouth, 1981.

[Petersson 82]
Kent Petersson.
A Programming System for Type Theory.
Dept. of Computer Science LPM Memo 21.
University of Göteborg, Sweden, 1982.

[PRL Staff 83]
The PRL Staff.
PRL: Proof Refinement Logic Programmer's Manual (Lambda PRL, VAX Version).
Computer Science Department, Cornell University, 1983.

[PRL Staff 85]
The PRL Staff.
Implementing Mathematics with the Nuprl Proof Development System.
Computer Science Department, Cornell University, 1985.

[Pratt 1980]
On Specifying Verifiers.
In *Proc. 7th ACM Symp. on Princ. of Prog. Lang.*, pp. 106–116, 1980.

[Prawitz 65]
Dag Prawitz.
Natural Deduction: A Proof-Theoretical Study.
Almquist and Wiksell, Stockholm, 1965.

[Reps & Alpern 84]
Interactive Proof Checking.
In *Proc. 11th ACM Symp. on Princ. of Prog. Lang.*, 1984, pp. 36–45.

[Reynolds 74]
Towards a Theory of Type Structure.
In *Proc. Colloque sur la Programmation.*, pp. 408–423.
Lecture Notes in Computer Science, vol. 19., Springer–Verlag, 1974.

[Robinson 65]
A Machine–Oriented Logic Based on the Resolution Principle.
J. ACM v.12, pp. 23–41 (1965).

[Rose 61]
H. E. Rose.
On the Consistency and Undecidability of Recursive Arithmetic.
Zeitschr. für Math. Logik und Grundlagen d. Math, v.7 (1961), pp. 124–135.

[Russell 08]
Bertrand Russell.
Mathematical Logic as Based on a Theory of Types.
American Journal of Mathematics v.30 (1908), pp. 222–262.

[Sambin 80]
Giovanni Sambin.
Notes by Giovanni Sambin of a series of lectures given in Padova, Italy, June, 1980.
Padova, 1980.

[Sasaki 84]
James T. Sasaki.
The Extraction and Optimization of Programs from Constructive Proofs.
Ph.D. Thesis (forthcoming), Cornell University, 1984.

[Scott 70]
Dana Scott.
Constructive Validity.
In *Symposium on Automatic Demonstration.* pp. 237–275
Lecture Notes in Mathematics vol. 125, Springer–Verlag, 1970.