A Non-Type-Theoretic Semantics for Type-Theoretic Language

Stuart Allen
Ph.D. Thesis

87-866
September 1987
(Revised December 1987)

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501
A NON-TYPE-THEORETIC SEMANTICS FOR
TYPE-THEORETIC LANGUAGE

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
Stuart Frazier Allen
August 1987
A Non-Type-Theoretic Semantics for Type-Theoretic Language

Stuart Frazier Allen, Ph.D.
Cornell University 1987

Since 1970 several methods have been proposed for using formal systems of constructive logic as programming languages. One prominent approach is based upon systems of computationally significant terms which either bear or are assigned types; these systems are essentially lambda calculi or combinatory logics in which either the terms are explicitly typed or else types are assigned to untyped terms in the manner of Curry. This thesis concerns two such systems, namely, Martin-Löf's intuitionistic type theory of 1979, and a variation of that theory upon which Nuprl is based. Nuprl is a system implemented at Cornell for developing functional programs and constructive proofs.

The expressive machinery of these theories can be given a rather natural non-type-theoretic semantics that is not inherently constructive and yet closely follows the semantical explanation of type theory. The principal content of this thesis is a careful development of such a semantic reinterpretation with the intention of making the bulk of type-theoretic practice, of the kind arising from the use of Nuprl and formalizations of Martin-Löf's theory, independent of its original type-theoretic and constructive basis. The reinterpretation opens the type-theoretic methodology of programming to nonconstructivists and others who may not subscribe to the intuitionistic theory of types, preserving the features of type-theoretic language that make it a suitable language for programming. Moreover, the natural structural similarity between the type-theoretic concepts and their reinterpretations yields an analytic tool which may serve type-theorists as well.

The body of this thesis has two phases. In the first, the semantic concepts of Martin-Löf's theory, including expressions, types, judgements of functionality, and universes, are reinterpreted. This phase culminates in a non-type-theoretic definition of the types explicitly defined in Martin-Löf's paper of 1979. The remainder of the thesis treats various topics of semantic significance, including the representation of propositions as types, the anticipation of new terms and types, certain "type-free" forms of inference, and a sort of "universe polymorphism." Finally, we shall reinterpret the semantics of Nuprl's judgements of functionality which differs radically from that of Martin-Löf's judgements in the use of assumptions.
Biographical Sketch

Stuart Allen received a B.S. in Computer Science from the University of New Orleans in 1978. After working for two years as a computer programmer, he enrolled at Cornell; in 1985 he received an M.S. in Computer Science.
Acknowledgements

Four people, I think, have most influenced my intellectual development, and I am grateful to them for it. My parents, Samuel Allen and Virginia Winfrey Allen, have always had a great sensitivity to the intellectual needs of their children and have always taken pains to have them satisfied. Fred Hosch, my undergraduate professor at the University of New Orleans, was a fine and stimulating teacher. He most impressed me with his inspiring blend of intelligence, humor, and seriousness. My graduate professor, Robert Constable, has provided me with the criticism, support, and encouragement that I needed in order to carry out the work of this thesis. It will be a while before I can assess his effect upon me, but, already I can appreciate his principled attitude toward his research, and I regard his work as a model. I may also say that I admire his taste in intellectual matters.

Joe Bates, with Constable, directed the Nuprl project. Working as a member of that project with about a dozen other people has been, for me, a profound experience in collaboration. Joe's friendship is, I believe, the most valuable side-effect of my participation in that project. Bob Harper, my friend and classmate, was the colleague with whom I worked most closely prior to his graduation. More recently, he read a draft of this thesis and provided me with many helpful remarks.

Doug Howe has spent a lot of time discussing with me nearly every topic of this thesis. Discussions with him are always enjoyable for me because he is so quick to understand even poorly presented situations. I am also grateful to Scott Smith for numerous discussions and for his comments on a draft.

Finally, I am thankful to have had the friendship of Vivian Sewelson, Jim Kadin, Meryl Hassman, and Rob Coalson, whose association and support have been most important to me during my attendance at Cornell. Also, the indices for this thesis were made possible through Meryl's efforts.
# Table of Contents

1 Introduction .......................................................... 1  
   1.1 Martin-Löf’s Theory of Types ................................. 5  
   1.2 The Type Theory of NuPrl ...................................... 10  
   1.3 Overview of the Thesis ....................................... 17  
   1.4 Related Reinterpretations ................................... 21  
   1.5 Notation ...................................................... 24  

2 Type Assignment Systems .......................................... 27  
   2.1 An Example: a System of Finite Types ..................... 31  

3 Functionality and Sequents ...................................... 35  
   3.1 Fn Facts .................................................... 39  

4 Universes .................................................................. 44  
   4.1 Universes and Type System Hierarchies .................... 45  
   4.2 An Example: Martin-Löf’s Types ............................. 48  
   4.3 Universe Polymorphism ........................................ 55  

5 Respect for Relations ............................................. 63  
   5.1 Respected Relations and Functionality ..................... 63  
   5.2 Change of Bound Variables ................................... 66  
   5.3 Partial Identity of Value ..................................... 67  
   5.4 Closed Subterm Variation ..................................... 67  
   5.5 An Example: Induction ....................................... 70  

6 Open-endedness ...................................................... 73  
   6.1 Adding New Types ............................................. 74  
   6.2 Adding New Terms ............................................. 77  

7 Propositions and Types ........................................... 81  

8 Pointwise Functionality ........................................... 89  
   8.1 Fn@ Facts .................................................... 92
Chapter 1

Introduction

Proposals for using formal systems of intuitionistic logic as programming languages were made in [Constable 71] and [Bishop 70]. Development of this idea at Cornell produced two formal proof systems designed to allow executable programs to be extracted from proofs. The first of these, PL/CV (V for verifier) [Constable & O'Donnell 78, Constable, Johnson, & Eichenlaub 82], accomplished the extraction rather directly by textually embedding PL/C\(^1\) programs in proofs, interspersed with nonexecutable proof text. The proof-theoretic significance of the embedding lay in the interpretation of programs as constructive proofs.

The other system, PRL (for program refinement logic) [Bates 79], was designed to permit the development of programs by rigorous progressive refinement of program specifications as well as proofs of propositions. Bates understood his approach to programming to be formalizing the methodology advocated in various structured programming texts, e.g., [Wirth 73, Conway & Gries 75, Dijkstra 76]. Refinement in PRL proceeds by reducing the proof that a specification, the goal, can be satisfied to proving the satisfiability of zero or more other specifications, the subgoals. The rules of refinement were designed to enable the construction of a program satisfying the goal from programs satisfying the subgoals. Thus, from a complete proof that a specification could be met, a program satisfying it could be automatically extracted.

Further work at Cornell was based on the correspondence between intuitionistic propositions and types proposed by Scott [Scott 70] and elaborated by Martin-Löf [Martin-Löf 73, HAN], the formal aspect of which was discovered by Curry [Curry, Feys, & Craig 58] and Howard [Howard 80].\(^2\) At

\(^1\)PL/C is a sublanguage of PL/I.

\(^2\)Without special concern for intuitionistic semantics, de Bruijn independently developed a similar correspondence between propositions (as the contents of possible assertions) and syntactic types of AUTOMATH expressions. (See [de Bruijn 68a, de Bruijn 68b, de Bruijn 80].) The act of assertion is expressed in AUTOMATH as the presentation of a term
first, much of this work focused on the design and study of systems of typed combinators \cite{Constable82, Constable83a, Constable83b, Constable83c, ConstableZlatin84}. The type theory of PL/CV3 \cite{ConstableZlatin84} is, perhaps, the best example of this phase. Matters of special interest to the investigators were the intensional analysis of types and functions, the discovery of new and useful type constructors, and the application of type theory to computer science. Later efforts were directed at the implementation of Bates's refinement methodology using Martin-Löf's type theory.

In \cite{HAN}, Martin-Löf presented an intuitionistic theory of types (also see \cite{Padova}) and recommended it as a basis for computer programming, defining types that made this recommendation plausible. In a manner formally similar to \cite{CurryFeysCraig58}, types are assigned to untyped expressions. The theory is based on untyped expressions whose computational use is stipulated when forms of expression are introduced. A type is \textit{defined} by specifying which (untyped) expressions are its members and which pairs of its members are \textit{equal}; thus, types are similar to Bishop's sets \cite{Bishop67}. Formalizations of the theory which use the type constructors defined in \cite{HAN} and the (natural deduction) inference rules given there, have been implemented by the group at Göteborg \cite{Petersson82, NordstromPeterssonSmith86}.

Bates and Constable adopted the identification of program specifications with types (as did Zlatin and Constable for PL/CV3), and designed a program refinement system, Nuprl \cite{BatesConstable81, BatesConstable83, Constableetal86}, which is an advanced development system for proofs of intuitionistic type theory. The complete system is explained in \cite{Constableetal86}. What is of concern here is that the formal system upon which Nuprl is built is based on Martin-Löf's theory of types. There are significant differences, especially between the forms of judgement, but the similarities are far greater.

The expressive machinery of intuitionistic type theory can be given a rather natural non-type-theoretic semantics that is not inherently constructive and yet closely follows the semantical explanation of type theory. The principal content of this thesis is a careful development of such a semantic reinterpretation with the intention of making the bulk of type-theoretic practice, of the kind arising from the use of Nuprl and formalizations of \cite{HAN}, independent of its original type-theoretic and constructive basis. The reinterpretation opens the type-theoretic methodology of programming to non-constructivists and others who may not subscribe to the intuitionistic theory of types, preserving the features of type-theoretic language that make it a suitable language for programming.

Moreover, in the unfamiliar domain of intuitionistic type theory, the reinterpretation can serve as a staff made of familiar mathematical material.
Questions of consistency and sensibleness may be treated using conventional, well understood means. For example, the inference rules of [HAN] can be shown to be valid under reinterpretation. Since the reinterpretation is non-trivial, the collection of rules is consistent, and since only terms that have types have values under the reinterpretation, any term typed using the rules of [HAN] has a value. Further, although one must not infer type-theoretic validity from validity under reinterpretation, or vice versa, the natural structural similarity between the type-theoretic concepts and their reinterpretations yields an analytic tool even for type-theorists. This author has seen a number of type-theoretically incorrect inference rules proposed by researchers, at Cornell and elsewhere, the defects of which are easily exposed when one attempts to show them valid under reinterpretation. Of course, the defects also become apparent if the type-theoretic arguments are carried out in sufficient detail; the value of reinterpretation in this context, at our current stage of familiarity with type-theoretic thought, seems to derive from our skill at gauging the adequacy of ordinary mathematical arguments.

We shall reserve the term type theory for genuine theories of logical type, such as Russell’s paradigmatic theory [Russell 08], Stenlund’s impredicative theory of species [Stenlund 72], and Martin-Löf’s intuitionistic theory of types, which are theories of propositional form based on the notion of domains of functions. Usually, we shall use the term to refer to Martin-Löf’s theory, especially as presented in [HAN]. Opposed to theories of logical type are systems in which terms and types are ordinary mathematical objects, and in which the connection between types and the terms of those types consists of an ordinary relation between objects. We shall refer to these as type assignment systems or simply type systems. Some examples of type assignment systems are systems that formalize theories of logical type and the numerous combinatory systems in which, by various methods, terms are assigned types. Usually, we shall use the term type system more narrowly to refer to a certain kind of type system which we will define later for use in our non-type-theoretic semantics.

Martin-Löf’s type theory was intended to serve as a theoretical basis for (intuitionistic) mathematics. The central features that permit it to serve its foundational purposes are these:

- A type is the range of significance for a function’s argument, as in Russell’s ramified theory, and this notion is elemental.

- A proposition may be construed as a type whose members are the “proofs” of that proposition. “Proof” is to be understood here in a sense that permits intuitionists to say that a proof of if $A$ then $B$ is a function that maps any proof of $A$ to a proof of $B$. Such a function requires that the proofs of $A$ form a type. Naturally, proposition-valued
functions are construed as type-valued functions.

- Functions are in some sense computable.

- There is a supply of methods for constructing types and functions that is rich enough to permit expression of a broad body of mathematics.

Using the formal language of Martin-Löf's theory of types as a programming language does not require interpreting it type-theoretically. We may instead, in a way developed in the body of this thesis, use the language as a formalization of a type assignment system. Stripped of its logical role, the essential use of a type is as an index to the collection of terms that are its members. Construing a proposition as the type of its proofs is supplanted by merely representing the proposition as a type that is inhabited just if the proposition is true.

Martin-Löf's type theory makes possible a unified and direct treatment of many concepts pertinent to programming.

- Since the evaluation of terms is computable, we may use the terms as programs and data values. Such a programming notation may be higher order in virtue of the terms being closed under lambda abstraction and application.

- A rich collection of data type constructors has been defined in the theory [HAN, Aczel 82, Constable et al. 86, Constable & Mendler 85, Mendler 87, Mendler, Panangaden, & Constable 86]. Much of the effort in this area at Cornell has been aimed at supplementing the types of [HAN], which satisfy many mathematical needs, with new methods of type construction necessary for programming. Types themselves (or indices) can be used as data and can be assigned types, one effect being that certain kinds of polymorphic functions can be specified, and another being that abstract data types are easily expressed [Constable 83b, Mitchell & Plotkin 85, MacQueen 86].

- So-called total and partial correctness of programs can be specified. The use of types of total functions is well understood, while the specification of partial functions in type theory is a topic of ongoing research [Constable 83c, Constable 83b, Constable & Mendler 85, Constable & Smith 87] (see especially [Constable & Smith 87]).

- Propositions of a ramified quantificational logic are easily expressed.

This unified framework need not be based on constructivist principles or on the theory of logical types. But, when one abandons the type-theoretic
account of this framework one may no longer claim a certain kind of comprehensiveness. Type theory is supposed to be a general account of mathematics and, perhaps, computation; it is not supposed that all of mathematics and computer science follow from it, but rather, that it is a proper theoretical framework in which to elaborate these things. Martin-Löf has even suggested that such comprehensiveness is a requirement for the logical adequacy of high level programming languages.\footnote{[HAN] closes with the statement, "In fact, I do not think that the search for logically ever more satisfactory high level programming languages can stop short of anything but a language in which (constructive) mathematics can be adequately expressed."}

We shall proceed to review the type theories of [HAN] and Nuprl, and then introduce the main work of this thesis, namely, the development of a certain non-type-theoretic reinterpretation of type-theoretic language.

## 1.1 Martin-Löf’s Theory of Types

In this thesis, we shall mean by Martin-Löf’s type theory the one presented in his Hannover paper of 1979, *Constructive Mathematics and Computer Programming* [HAN]. The reader is assumed to be familiar with that paper.\footnote{The theory of [HAN] differs substantially from its ancestor presented in [Martin-Löf 73].} A more thorough presentation is to be found in notes of Martin-Löf’s Padova lectures of 1980 [Padova] and in the forthcoming introductory text [Nordstrom, Petersson, & Smith] from the Göteborg group. At the end of this section there is a brief discussion about a variation of type theory which is suggested in [Padova].

The possibility of introducing forms of expression and defining types lies at the core of the theory. Introducing a form of expression does not depend on the definition of types. Part of introducing a form of expression consists in specifying how its subexpressions are indicated and specifying which variables become bound in which subexpressions.\footnote{Martin-Löf has formulated a system of expressions and definitions, his theory of arities, which bears on this aspect of introducing forms of expression, but we shall not use it here. It is explained in [Nordstrom, Petersson, & Smith].} Also, when one introduces a form of expression, one must explain how to evaluate expressions of that form. An expression can have at most one value, which must be an expression that has itself as value. Expressions that are values are called canonical. One interesting aspect of the expressions introduced in [HAN] is that whether an expression is canonical depends only on the outer form; canonical expressions may have noncanonical subexpressions. (A fascinating and detailed earlier account of the syntax of expressions may be found in notes on lectures given by Martin-Löf at Oxford in 1975 [Martin-Löf 75].)

A type is defined by choosing a canonical term, the type, then specify-
ing which (canonical) expressions are to be its canonical members, and then specifying which pairs of canonical members are to be equal. The equality between canonical members of a type may be any equivalence relation between them. Usually, type definition is effected by defining deterministic procedures for defining types given various parameters. We shall call these procedures type constructors; for the purpose of generalization, we may consider types to be 0-place type constructors. The canonical type constructors that are defined in [HAN] are conveniently divisible into three groups. The constituent types of a type are those upon whose definition the type construction depends.

- \( N_n \) \( \subseteq \text{I}(A, a, b) \rightarrow A + B \). Each of these defines a type from finitely many, explicitly given, constituent types.

- \( (\Sigma x \in A)B \rightarrow (\Pi x \in A)B \rightarrow (W x \in A)B \). Each of these defines a type from an explicitly given family of constituent types that is indexed by an explicitly given constituent type. We say \( B_x (x \in A) \) is a family of types if \( A \) is a type and, whenever \( a \) and \( e \) are equal members of \( A \), \( B_a \) and \( B_e \) are extensionally equal types.\(^6\)

- \( U_n \). The equal members of universe \( U_n \) are the extensionally equal types that are definable from the type constructors other than \( U_{n+1} \). Although we might have included universes in the first group as having no constituent types, it seems clearer to conceive of them as resulting from a second-order type constructor that builds a universe from a level number and a collection of ordinary first order type constructors.

The type constructors defined in [HAN] are sufficient for carrying out the well-known intuitionistic correspondence between types and propositions. We shall not review this correspondence here, but perhaps it should be noted that the quantificational logic is ramified in the following way. We may say that a member of \( U_n \) is a type of order-\( n \) entities. (We attach the order to the type.) A type of individuals is a member of \( U_0 \). Identifying propositions with types, the type of order-\( n + 1 \) propositions is \( U_n \) (only types of individuals are types of order-0 entities). A quantified proposition of order-\( n + 1 \) must have as the domain of quantification a type of order-\( n \) entities. For example, if \( (\Pi x \in A)B_x \) is a proposition of order-\( n + 1 \), i.e., \( (\Pi x \in A)B_x \in U_n \), then \( A \) is a type of order-\( n \) entities, i.e., \( A \in U_n \).

It should be especially noted that the introduction of new forms of expression and the definition of new types are permitted. When one introduces a form of expression, one must explain how expressions of that form are to

\(^6\)Types are extensionally equal when they have the same membership and the same equality between members.
be evaluated even when subexpressions are to be used that have forms not yet introduced. Similarly, the membership of a type must be specified in a way that anticipates the introduction of new forms of expression. For example, if a new atomic expression \( t \) is introduced with the value 0, then \( \text{suc}(t) \) must be a member of \( \mathbb{N} \) even though \( \mathbb{N} \) was defined before \( t \) was introduced.\(^7\) As another example, in order to admit \( B_z \) as a family of types over a type \( A \), the values of \( B_z \) for the possible future members of \( A \) must be considered. Similarly, when \( (\Pi x \in A)B_z \) is defined, one must anticipate applying its members to future members of \( A \). Since new types may be defined, type constructors must be defined in a way that makes them applicable to types not yet defined. It may not be possible to accomplish this kind of profound open-endedness by means of the non-type-theoretic reinterpretation that is the subject of this thesis. The reader might well keep this in mind since this is probably the principal limitation of the approach to be pursued here.

Let us review the four forms of judgement defined in [HAN]\(^8\):

\[
T \text{ type } (x_1 \in A_1 \ldots x_n \in A_n) \\
T = S \ (x_1 \in A_1 \ldots x_n \in A_n) \\
t \in T \ (x_1 \in A_1 \ldots x_n \in A_n) \\
t = s \in T \ (x_1 \in A_1 \ldots x_n \in A_n).
\]

The expressions of form \( x \in A \) in parentheses are called the assumptions of the judgement. The meanings are defined inductively in the number of assumptions. The meaning of a judgement without assumptions is independent of the meanings of judgements with assumptions. The meanings of the judgements with assumptions

\[
(x_1 \in A_n \ldots x_n \in A_n \ x \in A)
\]

are given in terms of judgements without assumptions and the judgement

\[
A \text{ type } (x_1 \in A_n \ldots x_n \in A_n).
\]

The judgements of forms other than “\( T \) type” presuppose other judgements.\(^9\) A judgement with assumptions

\[
(x_1 \in A_1 \ldots x_n \in A_n)
\]

\(^7\)We use \( \text{suc}(t) \) instead of Martin-Löf's \( t' \).

\(^8\)A more general account of Martin-Löf's conception of judgement is given as part of a series of lectures on logic [Martin-Löf 83].

\(^9\)A judgement is assigned a meaning only when the judgements it presupposes are known to hold.
presupposes that
\[
\text{for } i < n, \quad A_{i+1} \text{ type } (x_1 \in A_1 \ldots x_i \in A_i).^{10}
\]

A judgement
\[
T = S (x_1 \in A_1 \ldots x_n \in A_n)
\]
presupposes that
\[
T \text{ type } (x_1 \in A_1 \ldots x_n \in A_n) \text{ and } S \text{ type } (x_1 \in A_1 \ldots x_n \in A_n).
\]

A judgement
\[
t \in T (x_1 \in A_1 \ldots x_n \in A_n)
\]
presupposes that
\[
T \text{ type } (x_1 \in A_1 \ldots x_n \in A_n).
\]

A judgement
\[
t = s \in T (x_1 \in A_1 \ldots x_n \in A_n)
\]
presupposes that
\[
t \in T (x_1 \in A_1 \ldots x_n \in A_n) \text{ and } s \in T (x_1 \in A_1 \ldots x_n \in A_n).
\]

When there are no assumptions, \(T\) type means that \(T\) evaluates to a defined canonical type, and \(T = S\) means that \(T\) and \(S\) evaluate to canonical types that have the same canonical objects and the same pairs of equal canonical objects. Such equality between types is called \textit{extensional}. A judgement \(t \in T\) means that \(t\) evaluates to a canonical object of the canonical type to which \(T\) evaluates, and \(t = s \in T\) means that the values of \(t\) and \(s\) are equal in that type.

Judgements that use assumptions are judgements of \textit{functionality}. A judgement \(T\) type \((x_1 \in A_1 \ldots x_n \in A_n)\) means that
\[
T[a_1 \ldots a_n/x_1 \ldots x_n] = T[b_1 \ldots b_n/x_1 \ldots x_n]
\]
preserved that
\[
\text{for } i < n, \quad a_{i+1} = b_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i].
\]

A judgement \(T = S (x_1 \in A_1 \ldots x_n \in A_n)\) means that
\[
T[a_1 \ldots a_n/x_1 \ldots x_n] = S[a_1 \ldots a_n/x_1 \ldots x_n]
\]
preserved that
\[
\text{for } i < n, \quad a_{i+1} = b_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i].
\]

\(^{10}\)So, for example, no meaning is given to a judgement \(z \in A (x \in A)\) unless \(A\) is known to be a type.
A judgement \( t \in T \ (x_1 \in A_1 \ldots x_n \in A_n) \) means that
\[
t[a_1 \ldots a_n/x_1 \ldots x_n] = t[b_1 \ldots b_n/x_1 \ldots x_n] \in T[a_1 \ldots a_n/x_1 \ldots x_n]
\]
provided that
\[
\text{for } i < n, \ a_{i+1} = b_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i].
\]
A judgement \( t = s \in T \ (x_1 \in A_1 \ldots x_n \in A_n) \) means that
\[
t[a_1 \ldots a_n/x_1 \ldots x_n] = s[a_1 \ldots a_n/x_1 \ldots x_n] \in T[a_1 \ldots a_n/x_1 \ldots x_n]
\]
provided that
\[
\text{for } i < n, \ a_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i].
\]

The premises and conclusions of the inference rules given in [HAN] are judgements of the forms just discussed. One justifies such a rule by making the conclusion evident on the presupposition that one were to know the premises.

The form of judgement
\[
T \ \text{true} \ (x_1 \in A_1 \ldots x_n \in A_n)
\]
is not explicitly defined since assertions of such forms are supposed to be eliminable in favor of one of the forms already defined. Given an intuitionistic proof of
\[
T \ \text{true} \ (x_1 \in A_1 \ldots x_n \in A_n),
\]
one is supposed to be able to effectively find a term \( t \) and a proof of
\[
t \in T \ (x_1 \in A_1 \ldots x_n \in A_n) .\]

We close this review of Martin-Löf’s type theory by indicating a variation of the theory in which type expressions are distinguished from object expressions. In [Padova] Martin-Löf described two formulations of universes. The one described earlier in this section, under which each universe \( U_k \) is a minimal type closed under all the (already defined) type constructors other than \( U_{k+1} \), is called the formulation à la Russell. The other is called the formulation à la Tarski. Under this formulation, a universe \( U_k \) is inhabited, not by types, but rather, by indices of a family of types, \( T_k(x) \ (x \in U_k) \).

\(^{11}\)When Constable’s set types are defined, there is reason to doubt that this elimination is always possible. The reasons are essentially the same as those, given in chapter 7 of this thesis, for the failure under the same circumstances of the correspondence between \( \Pi \) and \( \forall \) (and between \( \rightarrow \) and “if-then”).
which is defined by mutual recursion with $U_k$. For each type constructor, e.g., $(\forall x : A)B$, there is a canonical form, e.g., $(\pi x : a)b$, in which indices to the constituent types of $(\forall x : A)B$ are used instead of the constituent types themselves. Thus, instead of

$$
\begin{array}{c}
(x \in A) \\
A \in U_k \\
B \in U_k
\end{array}
\frac{}{(\forall x \in A)B \in U_k}
$$

we have

$$
\begin{array}{c}
(x \in T_k(a)) \\
a \in U_k \\
b \in U_k
\end{array}
\frac{}{((\forall x \in a)b \in U_k)}
$$

$$
\begin{array}{c}
(x \in T_k(a)) \\
a \in U_k \\
b \in U_k
\end{array}
\frac{}{T_k((\forall x \in a)b) = (\forall x \in T_k(a))T_k(b)}
$$

The point of the Tarskian universes is that type expressions may be made distinct from object expressions, i.e., expressions for members of types. In particular, there is no longer a need for noncanonical types; they are replaced by noncanonical indices to types. Indeed, in Padova, instead of defining canonical types and using "a type" to mean that $A$ evaluates to a type, one merely defines types (called sets there), and "a type" (or "a set") simply means that $A$ is a defined type. (See below, page 22, for more on this topic.)

1.2 The Type Theory of Nuprl

Nuprl is an automated system for interactively developing proofs and (functional) programs using Bates's refinement methodology [Bates 79]. The primary reference on Nuprl is [Constable et al. 86]. The exploitation of type theory is just one aspect of the design of Nuprl. The system also includes an interactive proof editor, an evaluation mechanism for terms, a definition facility for introducing notation, and a programming language that can use Nuprl proofs as data. Proofs, definition templates, and ML programs may be stored in a library. For a discussion concerning the design of Nuprl's formal proof system, [Harper 85] is highly recommended.

It is the semantics of Nuprl that is of concern in this thesis. The theory of types upon which the Nuprl system [Constable et al. 86] is based was designed by Constable and Bates as a variation on Martin-Löf's theory. The

---

12Programs written in the language ML [Gordon, Milner, & Wadsworth 79] are used in various ways to construct Nuprl proofs [Constable, Knoblock, & Bates 85]. In certain circumstances it is important to be sure that a program intended to construct a proof is correct. This is reason to use Nuprl itself as a programming (and specification) language for constructing proofs. That is the subject of [Knoblock & Constable 86] and of Knoblock's forthcoming thesis [Knoblock 87].
type theory of Nuprl was originally given expression by means of a body of inference rules. The formulations to be given here for the concepts of multi-type and of pointwise functionality are the result of an attempt to make explicit the semantic principles implicit in that body of rules, and they form part of the results of this thesis.

The principal innovations with respect to [HAN] are these:

- An intensional equality between types is used, instead of extensional equality, to form families of types and to define universes.

- Subtype and quotient type constructors are defined.

- The meaning of assumptions within judgements differs radically from that given by Martin-Löf.

We shall not treat certain extensions which have been made to Nuprl. One extension specifies a wide class of (not necessarily atomic) type constructors which determine families of fixed points which are defined as types [Constable et al. 86, Constable & Mendler 85, Mendler, Panangaden, & Constable 86]. The type constructor \((W x \in A)B_x\) of [HAN] is one simple instance: it is essentially a minimal fixed point of \((\Sigma x \in A)(B_x \rightarrow T)\) (in \(T\)). Another extension defines types whose members, so-called partial objects, might have no values [Constable & Smith 87].

The type equality defined in [HAN] is extensional, but in Nuprl it is not, i.e., it is intensional.\(^{13}\) As will be discussed later, the intensional equality between equality types (I-types) in Nuprl permits a certain economy of expression in the forms of judgement. Intensional type equality can be made strong enough to permit the typing of functions over universes\(^{14}\) that perform recursive case analysis on the arguments (and which may have unequal values on some unequal arguments). See [Constable & Zlatin 84, Aczel 82] for formulations of such recursion schemas.\(^{15}\) The difficulty with typing such functions under extensional type equality is that some extensionally equal canonical types will fall under different cases within whatever scheme for recursive analysis we are likely to use.

\(^{13}\)It is interesting that the rules of inference given in [HAN] remain valid if the intensional type equality of Nuprl is adopted. Thus, those rules take no advantage of extensional type equality. For example, there is no rule

\[
\begin{array}{c}
(\exists z \in A) & (\exists z \in B) \\
\hline
\exists z \in B & \exists z \in A \\
\end{array}
\]

\[
A = B
\]

\(^{14}\)or other types of types,

\(^{15}\)Aczel uses his schema in [Aczel 82] to verify some choice axioms under a type-theoretic interpretation of set theory.
Let us write \( T = S \) to mean that \( T \) and \( S \) are equal types of Nuprl. Here are some examples of type equality in Nuprl (using Martin-Löf’s notation for type constructors):

- \( T = \text{I}(A,a,b) \) just when, for some \( A', a', \) and \( b' \), \( T \) evaluates to \( \text{I}(A',a',b') \), and \( A = A' \), and \( a = a' \in A \), and \( b = b' \in A \).

- \( (\Pi x \in A)B \) is a type just when \( A = A' \) and \( B[a/x] = B[a'/x] \) for \( a = a' \in A \).

- \( T = (\Pi x \in A)B \) just when, for some \( A', x' \) and \( B' \), \( T \) evaluates to \( (\Pi x' \in A')B' \) (or \( A' = B' \)), and \( A = A' \), and \( B[a/x] = B'[a'/x] \) for \( a = a' \in A \).

- \( T = \{ x \in A \mid B \} \) \(^{16}\) just when, for some \( A', x' \) and \( B' \), \( T \) evaluates to \( \{ x' \in A' \mid B' \} \) (or \( A' = B' \)), and \( A = A' \), and the following types are inhabited:

\[
(\Pi u \in A)(B[u/x] \rightarrow B'[u/x']),
\]

\[
(\Pi u \in A)(B'[u/x'] \rightarrow B[u/x]),
\]

where \( u \) is free for \( x \) in \( B \) and \( x' \) in \( B' \).

The use of intensional type equality for certain purposes does not prohibit the use of extensional equality for others. For example, one might define a type constructor \( \text{Ext}(A) \) such that, given a type \( A \), \( \text{Ext}(A) \) is a type and is extensionally equal to \( A \), and given types \( A \) and \( B \), \( \text{Ext}(A) = \text{Ext}(B) \) just when \( A \) and \( B \) are extensionally equal. But, of course, a function on types that recursively analyzes its argument must not depend for its value at \( \text{Ext}(A) \) upon its value at \( A \). Similarly, the value of such a function at \( \{ x \in A \mid B_x \} \) must not depend on its values for the family of types \( B_x (x \in A) \).

Now that we have indicated some of the character of intensional equality in Nuprl, we shall establish its type-theoretic sense. It seems that when one defines a type, one is also supposed to specify and define the types that are intensionally equal to it. Since intensionally equal types are also extensionally equal, we may specify at one time the (equality between) canonical members of equal types. This amounts to modifying Martin-Löf’s criteria for type definition; instead of choosing a single canonical term to serve as the type, one specifies a(n open-ended) class of canonical terms to serve as types. Let us call such a modified definition a multi-type definition, and call the specification of the terms that are to serve as types a multi-type, and call the terms that fall under a multi-type its instances. A multi-type definition is to be construed as defining its instances. By \( T = S \) we shall mean that \( T \) and \( S \) evaluate to instances of a defined multi-type, and we shall say \( T \) is a

\(^{16}\) The set type constructor is defined below.
type, or T type, when \( T = T \). Notice that \( T = S \) does not presuppose that \( T \) and \( S \) are types.

By \( t = s \in T \) we shall mean that \( T \) is a type and that \( t \) and \( s \) evaluate
to members of (the value of) \( T \), and that moreover, they evaluate to equal
members.\(^{17}\) By \( t \in T \) we shall mean that \( t = t \in T \). Notice that \( t = s \in T \)
does not presuppose \( t \in T \), \( s \in T \), or \( T \) type, but simply implies them.

Although Nuprl assigns to assumptions a meaning quite different from
that assigned in [HAN], it will be convenient to introduce a form of judgement
with assumptions that is analogous to Martin-Löf’s. In contrast to Martin-
Löf’s judgements, the meanings are given without presuppositions. By \( T =
S \left( x_1 \in A_1 \ldots x_n \in A_n \right) \) we shall mean that

\[
\text{for } i < n, \ \text{type } \ ( x_1 \in A_1 \ldots x_i \in A_i ),
\]

and

\[
T[a_1 \ldots a_n/x_1 \ldots x_n] = S[b_1 \ldots b_n/x_1 \ldots x_n]
\]

provided that for \( i < n, \)

\[
a_{i+1} = b_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i].
\]

Similarly, by \( t = s \in T \left( x_1 \in A_1 \ldots x_n \in A_n \right) \) we shall mean that

\[
T \ \text{type } \ ( x_1 \in A_1 \ldots x_n \in A_n )
\]

and

\[
t[a_1 \ldots a_n/x_1 \ldots x_n] = s[b_1 \ldots b_n/x_1 \ldots x_n] \in T[a_1 \ldots a_n/x_1 \ldots x_n]
\]

provided that for \( i < n, \)

\[
a_{i+1} = b_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i].
\]

Now we turn to the type constructors of Nuprl. The type constructors
common to [HAN] and Nuprl are dependent product, dependent sum,
binary disjoint union, equality, an empty type, and universes. For the sake of
uniformity, we shall use the [HAN] notation for these.\(^{18}\) The analogs of the
other type constructors of [HAN] could have been defined in Nuprl as well.

The type constructors of Nuprl are multi-type constructors, that is, they
define multi-types. In Nuprl, when a type is built from a family of constituent
types (indexed by a constituent type), it is not sufficient, as it is in [HAN],

\(^{17}\) Some recent work on partial functions [Constable & Smith 87] allows definitions of types
that have members with no values.

\(^{18}\) Nuprl uses \( x : A \rightarrow B, x : A \# B, A / B, a = b \) in \( A \), void, and \( U_{n+1} \) (Nuprl universes start
at \( U_1 \)).
that the family be extensional; rather, equal indices must yield (intensionally)
equal types. Except for set types and quotient types, defined below, the
types defined by a given type constructor are equal in Nuprl just when their
Corresponding constituent types are equal.

With regard to new types, the principal advance from [HAN] to Nuprl
is the definition of the so-called set type and quotient type constructors.
The set type \( \{ x \in A \mid B_x \} \), given \( B_x \) type \( x \in A \), is a subtype of \( A \) whose
members are the \( a \in A \) such that \( B_a \) is inhabited, and whose equality is the
obvious restriction of equality in \( A \). The types equal to \( \{ x \in A \mid B_x \} \) are
the set types built from families of constituent types \( B'_x \) (\( x \in A' \)) such that
\( A = A' \) and such that the following types are inhabited:

\[
(\Pi x \in A)(B_x \rightarrow B'_x),
\]

\[
(\Pi x \in A)(B'_x \rightarrow B_x).
\]

It would have been simpler to require only that, for \( a \in A \), \( B_a \) should be in-
habited just when \( B'_a \) is. (The nonequivalence of these conditions is explained
in chapter 7.)

Given a family of types \( E_{x,y} \) \( x \in A, y \in A \) that represents an equivalence
relation, the quotient type \( x, y \in A \mid E_{x,y} \) is a type whose members are the
members of \( A \), but whose equality is represented by \( E_{x,y} \).\(^{19}\) The types equal to
\( x, y \in A \mid E_{x,y} \) are the quotient types built from families of constituent
types \( E'_{x,y} \) \( x \in A', y \in A' \) such that \( A = A' \) and such that the following
types are inhabited:

\[
(\Pi x \in A)(\Pi y \in A)(E_{x,y} \rightarrow E'_{x,y}),
\]

\[
(\Pi x \in A)(\Pi y \in A)(E'_{x,y} \rightarrow E_{x,y}).
\]

Again, it would have been simpler to require only that, for \( a, b \in A \), \( E_{a,b} \)
should be inhabited just if \( E'_{a,b} \) is.

These two type formation operations might be regarded as partially re-
fecting the two steps for giving the membership and equality of a canonical
type definition: a canonical member of \( u, v \in \{ x \in A \mid B \} \mid E \) is formed by

\(^{19}\)Actually, there is a further, and unnecessary, restriction on \( E_{x,y} \) \( x \in A, y \in A \), viz.,
that there are functions of the following types:

\[
(\Pi u \in A)E_{u,u}
\]

\[
(\Pi u \in A)(\Pi v \in A)(E_{u,v} \rightarrow E_{v,u})
\]

\[
(\Pi u \in A)(\Pi v \in A)(\Pi w \in A)(E_{u,v} \rightarrow E_{v,w} \rightarrow E_{u,w})
\]

where \( u, v, w \) are distinct and are free for \( x \) and \( y \) in \( E_{x,y} \). For reasons explained
in chapter 7, this condition is stronger than the condition that \( E_{x,y} \) represent an equivalence relation
on \( A \).
forming an \( a \in A \) such that \( B[a/x] \) is inhabited; two equal canonical members are formed by forming \( a, a' \in \{ x \in A \mid B \} \) such that \( E[a, a'/u, v] \) is inhabited. The set type and quotient type constructors could have been unified in a single constructor \( x, y \in A/E_{x,y} \) which is like quotient except that, rather than requiring (the inhabitation of) \( E_{x,y} \) to be an equivalence relation, we require only that it be transitive and symmetric over \( A \), i.e., its restriction to \( A \) should be a partial equivalence relation. The equal members are the members of \( A \) that make \( E_{x,y} \) inhabited. Thus, a type \( x, y \in A/E_{x,y} \) is extensionally equal to \( x, y \in A/E_{x,y} \), and a type \( \{ x \in A \mid B_x \} \) is extensionally equal to \( x, y \in A/(B_x \times I(A, x, y)) \).

We come now to Nuprl's treatment of assumptions. Nuprl uses one form of judgement:

\[
x_1 : A_1 \ldots x_n : A_n \gg t \in T. \tag{20}
\]

Let us start by considering Nuprl judgements with one assumption. The meaning of \( x \in A \gg t \in T \) is that, for any \( a \) and \( a' \), if \( a = a' \in A \) then \( T[a/x] = T[a'/x] \) and \( t[a/x] = t[a'/x] \in T[a/x] \). Notice that, rather than implying or presupposing that \( A \) is a type, the typehood of \( A \) is part of the assumption (since the typehood of \( A \) is implied by \( a = a' \in A \)). Thus, if \( A \) cannot be defined as a type, because it has no value, say, then we may infer for any \( x, T, \) and \( t \) that \( x \in A \gg t \in T \). In contrast, we cannot infer \( t \in T (x \in A) \) unless we also know that \( A \) is a type. Since we are discussing two forms of assumption, it will be convenient to introduce a distinguishing nomenclature; there will be no need to make the general application of the terminology precise. We shall say an assumption \( x \in A \) is positive within the judgements that, by virtue of that assumption, imply the typehood of \( A \), and we shall say the assumption is negative within the judgements in which the typehood of \( A \) is a part of what is being assumed. The assumption \( x \in A \) is positive within \( t \in T (x \in A) \) and negative within \( x \in A \gg t \in T \). The use of negative assumptions allows one to express the assumption that \( a \) is a member of \( A \) as a negative assumption \( x \in I(A, a, a) \). A positive assumption of this form would be vacuous since for \( I(A, a, a) \) to be a type \( A \) must be a type with member \( a \).

Now we shall consider judgements that use two negative assumptions. The meaning intended for judgements using more assumptions should be clear in light of the explanation for two assumptions. A coarse reading, one

\[z_1 : A_1 \ldots z_n : A_n \gg T \text{ ext } t.\]

The part "ext t" is not displayed by the Nuprl system when it occurs in proofs, but rather, it is extracted from a completed proof. Most proofs are constructed without the user knowing precisely what term is to be extracted.
ignoring type functionality in the assumptions, would render

\[ x \in A, \ y \in B \gg t \in T \]

as

for any \( a, a', b, b' \) such that \( a = a' \in A \) and \( b = b' \in B[a/x] \),
\[ T[a,b/x,y] = T[a',b'/x,y] \]
and \( t[a,b/x,y] = t[a',b'/x,y] \in T[a,b/x,y] \).

It is already clear that part of the assumption \( x \in A \) is to be that \( A \) is a type, but what is the corresponding content of the assumption \( y \in B? \) One extreme would be to take the above reading literally, which would be to assume nothing about the functionality of \( B \) (in \( x \) over \( A \)), but rather, it would be to assume merely that \( B[a/x] \) is a type for each \( a \in A \). Another extreme would be to assume the functionality of \( B \) over \( A \) (in \( x \)). Nuprl takes a middle course which, for each \( a \in A \), assumes that \( B \) is functional when restricted to the members of \( A \) that are equal to \( a \), thus:

for any \( a, a', b, b' \) such that \( a = a' \in A \) and \( b = b' \in B[a/x] \),
\[ T[a,b/x,y] = T[a',b'/x,y] \]
and \( t[a,b/x,y] = t[a',b'/x,y] \in T[a,b/x,y] \).

This form of judgement, which we may call pointwise functionality, is more complex than the forms of [HAN]. The payoff is that a proof by induction over \( N \), say, of \( x \in N \gg t_x \in T_x \) can proceed to demonstrate point-by-point (\( N\)-by-\( N \) in this case), not only that \( t_x \) inhabits \( T_x \), but also that \( T_x \) is type-functional over \( N \). It is not part of the inductive hypothesis that \( T_x \) is type-functional over all of \( N \); rather, only the type-functionality of \( T_x \) at one point is assumed.

It is not essential to judgements of pointwise functionality that the consequent have the form \( t \in T \). Indeed, we shall permit ourselves to use in place of \( t \in T \) any of the other forms already introduced, including hypothetical judgements with positive assumptions, for example,

\[ x_1 \in A_1 \ldots x_n \in A_n \gg (B \text{ type } (x \in A)) \).

It should be clear how the meaning of such judgements is to be given. One reason for including positive assumptions is that full functionality is still a concept that is needed to define types from families of types. For example,

\[ x_1 \in A_1 \ldots x_n \in A_n \gg (\Pi x \in A)B \text{ type} \]

---

\[ ^{21} \text{See chapter 8.} \]
1.3 Overview of the Thesis

just when

\[ x_1 \in A_1 \ldots x_n \in A_n \overset{\text{B type}}{\Rightarrow} (\text{x} \in A \text{) ).} \]

Let us reconsider the fact that Nuprl uses judgements only of the form

\[ x_1 \in A_1 \ldots x_n \in A_n \overset{t \in T}{\Rightarrow} \]

How does one express type equality (and typehood) and member equality? For simplicity, the remarks that follow will be phrased in terms of judgements using only one negative assumption, but they apply as well to forms using any number of negative or positive assumptions. In order to explicitly express \( x \in A \overset{t = s \in T}{\Rightarrow} \), one must use the equality type constructor thus:

\[ x \in A \overset{e \in I(T, t, s)}{\Rightarrow} \]

One may choose what to use for \( e \), and one may always choose the canonical term \( r \) (Nuprl uses "axiom," which is a misnomer). This representation of member equality is justified by the fact that if \( a = b \in A \) then

\[ I(T[a/x], t[a/x], s[a/x]) = I(T[b/x], t[b/x], s[b/x]) \]

just when

\[ T[a/x] = T[b/x] \text{ and } t[a/x] = t[b/x] \in T[a/x] \text{ and } s[a/x] = s[b/x] \in T[a/x]. \]

This is possible because of Nuprl's intensional equality between equality types; mere extensional equality between equality types does not always imply equality of the corresponding components. Turning to type equality, instead of expressing \( x \in A \overset{T = S}{\Rightarrow} \), one must choose a universe \( U_k \) of which \( T \) and \( S \) are members if they are types, and then apply the above translation to \( x \in A \overset{T = S \in U_k}{\Rightarrow} \).

This concludes our review of intuitionistic type theory.

1.3 Overview of the Thesis

The main work of this thesis consists in developing a certain natural non-type-theoretic reinterpretation of type-theoretic concepts. The development is meant to proceed in a way that preserves the suitability of type-theoretic language as a programming language; it is also intended to be neutral with respect to constructivist principles. The body of this thesis may be viewed as having two phases. Chapter 2 through section 4.2 contains the basic substance of the non-type-theoretic reinterpretation of Martin-Löf's theory. Section 4.3 through chapter 8 treats a variety of related matters which are of practical or theoretical interest.

The following concepts are used to give the type-theoretic semantics of the language presented in \( \text{\textquotesingle HAN\textquotesingle} \):
• expression (form of expression, variable, free and bound occurrences)
• value of an expression
• type
• membership in a type
• equality of members of a type
• equality of types
• functionality (expressed by use of assumptions).

The semantics of Nuprl may be given using similar concepts except that

• it requires the use of multi-type definitions to define types,

• instead of using extensional type equality in its formulation of the concept of family of types, it uses the sameness of the multi-types of which the relevant types are instances, and

• instead of using assumptions to express full functionality, it uses them to express pointwise functionality (although full functionality is still used in the definition of particular multi-types).

We shall adopt an obvious and natural reinterpretation of these concepts. Under this interpretation, expressions are construed as forming an inductively defined fixed class of objects, rather than an open-ended body of linguistic forms. Since we shall grant no linguistic significance to these objects, i.e., we shall not suppose they express anything, let us use “term” instead of “expression.” All the other concepts are taken as fixed relations between terms. Evaluation is taken to be a relation between terms that defines a partial function. Rather than admitting the possibility of defining new types, we shall take typehood to be a fixed property of terms, and \( t \in T \) and \( t = s \in T \) to be fixed two- and three-place relations between terms; clearly, then, \( t \in T \) is just \( t = t \in T \). Equality between types is just another relation between terms, and so, since every type is equal to itself, \( T \) type is simply \( T = T \).

The \( T \)-functionality of \( t \), or the type-functionality of \( T \), under assumptions \( (x_1 \in A_1 \ldots x_n \in A_n) \) is to be simply a relation between the terms

\[
t, T, A_1, \ldots, A_n, x_1, \ldots, x_n.
\]

It will be convenient to give uniform treatment to judgements of functionality with any number of assumptions. By sequents we shall mean objects (which may be defined by induction from terms) that have the structure of the notations used to express judgements with zero or more assumptions.
Functionality is then formulated simply as a fixed property of sequents. An inference rule is taken to be correct if the conclusion is true whenever the premises are, thus, inference rules fall outside the semantics.

Chapter 2 introduces the kind of type assignment systems that form the basis of our reinterpretation. A type assignment system (type system, for short) will consist of a class of terms, an evaluation relation, and two relations between terms taken to be equality between types and equality between members of a type. A type assignment system is defined, in chapter 4, which is analogous to the body of types defined in [HAN].\textsuperscript{22} The definition of such a type system is needed to show that the approach to semantics taken here is applicable.

The balance of the thesis treats several topics concerning the use of type assignment systems:

- functionality (pointwise and full)
- universes
- avoiding explicit reasoning about particular types in certain situations
- open-endedness
- representing propositions by types.

Pointwise functionality is treated last, in chapter 8, and the various other topics are treated in terms of full functionality. Chapter 3 introduces sequents and defines an analog of functionality as expressed by the judgements defined in [HAN]. The chapter ends with a collection of general facts about functionality, i.e., facts that do not depend upon the definition of particular types; the general rules of [HAN] (reinterpreted) are instances of some of these facts.

The definition of a hierarchy of universes is an important feature of [HAN] and Nuprl. Each universe $U_n$ reflects in its membership (and equality) the types constructed by the type constructors other that $U_{n+i}$. The formulation of reflective hierarchies of universes in our non-type-theoretic setting is carried out in chapter 4. One frustration of using a reflective hierarchy is the apparent necessity of independently proving theorems that differ only in the choice of universes, but are true for essentially the same reasons. Some relief is offered in section 4.3 in the form of a scheme for expressing a certain kind of universe polymorphism. The universe-polymorphic forms of assertion to be elaborated generalize over universe indices in a way that exploits the fact that many statements about universes depend only upon simple arithmetic relations between universe indices. It will be argued that exclusive use of

\textsuperscript{22}This definition is not given until chapter 4 because the universe hierarchy is needed.
universe-polymorphic statements is desirable even though such statements may be strictly stronger than their non-polymorphic analogs. The argument will concern the anticipation of adding new types.

Chapter 5 treats a class of inferences that permit the replacement of subterms without requiring explicit consideration of particular types. For example, if \( t \), perhaps having free variables, is a redex with contractum \( s \), then from

\[
t \in T \left( x_1 \in A_1 \ldots x_n \in A_n \right)
\]

one may infer

\[
t = s \in T \left( x_1 \in A_1 \ldots x_n \in A_n \right).
\]

Other examples are Nuprl's direct computation rules; these rules have proved useful, and they are based upon the ideas presented in chapter 5.

Chapter 6 presents an attempt to recover some of the open-endedness inherent in the theory of types. The result is not entirely satisfactory because it fails to accommodate generally the addition of new type constructors. However, it does allow for the addition of arbitrary new types and new terms, and this is sufficient for adding new families of types in extension. Also, the anticipation of new types allows us to make precise the argument for exclusive use of universe-polymorphic assertions.

The subject of chapter 7 is the representation of propositions as types. The standard intuitionistic correspondence between propositions and types is not suitable for use with our non-type-theoretic semantics. Naturally, nonconstructivists will not accept the correspondence, and intuitionists may doubt the formal correspondence to be valid under the non-type-theoretic interpretation because that interpretation is not sufficiently open-ended. The standard formal correspondence also breaks down type-theoretically when Constable's set type and quotient type constructors are defined, as will be shown. No attempt will be made in this thesis to reformulate the correspondence between propositions and types in a way that preserves its logical role and also accommodates set types and quotient types. Rather, a weaker correspondence will be presented which is valid both constructively and classically, and is valid under both the type-theoretic and non-type-theoretic semantics. Under the weak correspondence, a type that represents a proposition will still be inhabited just when the proposition is true, but it will often be deficient in computational content. Thus, the weak correspondence cannot be used to explain intuitionistic logic, but it can still be of service in practice when the computational content is not explicitly needed. Chapter 7 closes with a discussion of whether propositions ought to be represented by types when the non-type-theoretic semantics is used.

Chapter 8 is about pointwise functionality. It includes a precise definition of the non-type-theoretic analog of pointwise functionality as well as proofs of several properties of that analog. It will be indicated how the results of
earlier chapters are to be applied when pointwise functionality is used instead of full functionality.

1.4 Related Reinterpretations

We shall discuss Michael Beeson's recursive realizability interpretation for Martin-Löf's type theory, and Jan Smith's formalized interpretation into a type-free language.

Beeson's Models

Below, in section 4.2, we shall define a sequence $\text{HAN}_n$ of type assignment systems, each $\text{HAN}_n$ consisting of a property of terms $A$ type $n$ and a three-place relation $a = b \in nA$. The type system $\text{HAN}_n$ is supposed to capture the types that are definable from the type constructors that are explicitly given in [HAN], using only the forms of expression introduced there.

In [Beeson 82], in order to obtain some consistency and independence results pertaining to Martin-Löf's formal system, Beeson presents a sequence $M_nW$ of models for the sequence $\text{ML}_nW$ of formal systems. The system $\text{ML}_nW$ consists of the inference rules given in [HAN], excluding those about universes $U_{n+1}$. The model $M_nW$ may be viewed as a recursive realizability interpretation. The basic difference between $\text{HAN}_n$ and $M_nW$ is that $\text{HAN}_n$ is a term model based upon the so-called lazy evaluation procedure described in [HAN], whereas $M_nW$ uses numbers instead of terms, and uses call-by-value computation semantics. Beeson codes the canonical term constructors, except for lambda, and uses $\{m\}(n)$ instead of the application of one term to another.

Beeson defines some relations between numbers, including $M_nW \models i = j$ and $M_nW \models i = j \in k$. The terms of [HAN] are made to correspond to numbers by an effective partial function on closed terms, $t$, also written $t^\sim$. The connection with [HAN] is made by the following facts.

If $\text{ML}_nW$ proves $A = B$ then $M_nW \models \hat{A} = \hat{B}$, and

if $\text{ML}_nW$ proves $a = b \in A$ then $M_nW \models \hat{a} = \hat{b} \in \hat{A}$.

This model deviates immediately from the semantics even when restricted to the terms and type constructors explicitly given in [HAN]. For example, if $t$ is $(\lambda z)(z(z),0)$ then $t(t)$ evaluates to $(t(t),0)$, hence $(\text{E}x,y)(t(t),0)$ evaluates to 0, and so the judgement $(\text{E}x,y)(t(t),0) = 0 \in N$ is valid. But, $t(t)^\sim$ is undefined, hence, $(\text{E}x,y)(t(t),0)^\sim$ is also undefined, and so, it is false that $M_nW \models (\text{E}x,y)(t(t),0)^\sim = 0 \in N$. 

While there may be no theoretically significant difference between the realizability model $M_nW$ and the term model $\text{HAN}_n$, the latter is a more direct reinterpretation of the semantics of type definition given in $\text{[HAN]}$, and $\text{HAN}_n$ is more convenient to use when one is concerned directly with the terms.

**Smith’s Interpretation**

Peter Aczel, in the last section of [Aczel 80], describes how to model a variant of the system of [Martin-Löf 73] using Frege structures, a concept he introduces in that paper. It fell to Jan Smith [Smith 84] to adapt Frege structures to the interpretation of Martin-Löf’s later systems [HAN, Padova].

The structure of the interpretation is roughly as follows. A formalization, $\text{LT}$, of a logical theory is set down which can be modeled using Frege structures. A formalization, $\text{TT}$, of type theory is given which is nearly a subsystem of that shown in $\text{[HAN]}$. A mapping from expressions of $\text{TT}$ to expressions of $\text{LT}$ is given which, for the most part, reflects the definitions of the types given in $\text{[HAN, Padova]}$. Finally, a mapping from the judgement forms of $\text{TT}$ to those of $\text{LT}$ is given which reflects the semantics of type-theoretic judgements and which preserves formal derivability.

Here are the differences between the system $\text{TT}$ and the system shown in $\text{[HAN]}$:

- The forms of object and type expressions are kept distinct and the formulation of universes is à la Tarski. (See above, page 9.)

- Only a single universe is used.

- $W$-types are omitted.

The distinction between object expressions and type expressions calls for a different reinterpretation of types than the one used in this thesis. When one defines a type, one no longer associates an object term with it. All that is needed is a notation for the equality relation between members of the defined type. Rather than reading $a = b \in A$ as a three place relation between terms, we may read it as a variable ranging over two-place relations on terms, just as $aRb$ might be, with $A$ playing the same role as $R$. The $A$ in $a = b \in A$ is just a conveniently separable piece of notation that streamlines our construction of notations for types, which we might regard as “properly” having the form $(t) = (t) \in A$. For example, the sign for the cross product of two relations $(t) = (t) \in A$ and $(t) = (t) \in B$ is conveniently rendered as $(t) = (t) \in A \times B$, an analogous notational device being the conventional use of $(t)(R \circ S)(t)$ to designate the composition of $(t)R(t)$ with $(t)S(t)$. Another way to look at it
would be to define a sign for cross product that is more generally applicable such as

\[(a) = (b) \in \text{Cross}(a, c, \Phi a c; b, d, \Psi b d),\]

where the variables to the left of a dot bind the free occurrences to the right, and let \(t = s \in A \times B\) stand as a convenient abbreviation for

\[t = s \in \text{Cross}(a, c. a = c \in A; b, d. b = d \in B).\]

The syntactic class of possible\(^{23}\) type expressions is, then, simply a class of notational forms that can be conveniently detached from the syntactic form \((a) = (b) \in \ldots\) and which can be easily combined to form other such conveniently detachable forms. So, it seems that the natural reinterpretation of the concept of type, under these circumstances, is simply to say that a type is a partial equivalence relation (between terms) whose relata must have values to which they are related. And there appears to be no reason to further restrict which relations are to be considered types. Thus, to interpret a type expression \(A\), one need only indicate the circumstances under which \((a) = (b) \in A\) may be used to signify a relation\(^{24}\) and which relation it signifies.

What has just been described is the semantic analog to Smith’s method for formally interpreting types in LT. For each type expression \(A\) of TT, he gives a propositional formula \(A'[z', z'']\) of LT which formally represents \(z' = z'' \in A\). LT has judgements of the forms “\(A\) prop” and “\(A\) true”, some judgements presupposing others, and its variables are interpreted to range over untyped terms. Smith gives a mapping from judgements of TT to judgements (with assumptions) of LT which makes functionality explicit, and which preserves the presuppositional structure of judgments\(^{25}\). He then gives several cases of his proof (by induction on the derivations in TT) that whenever a judgement is derivable in TT, the corresponding judgements are derivable in LT.

Let us now turn to the interpretation of universes. LT has a reflective mechanism paralleling that of TT.\(^{26}\) There are two predicates \(P(a)\) and \(T(a)\) which are used to reflect the propositions not having constituent propositions that are themselves of the forms \(P(a)\) and \(T(a)\). Except for \(P(a)\) and \(T(a)\), for each form of proposition in LT, e.g., \((\forall x)B\), there is an object form, e.g., \((\forall x)b\), where \(b\) is the object corresponding to \(B\). The proposition \(P(a)\) is true when \(a\) is an object corresponding to a proposition \(A\); and, supposing \(P(a)\) to be true, \(T(a)\) is a proposition that is equivalent to \(A\).

---

\(^{23}\)We say possible because one may require that particular presuppositions be met in order for a particular expression of the class to signify a type.

\(^{24}\)This is where any presuppositions would come in.

\(^{25}\)This is in contrast to the interpretation of judgements that is used in this thesis in which every judgement is given meaning without presuppositions.

\(^{26}\)See page 9.
interpretation of the universe of TT, the members are all the objects that, under P and T, represent partial equivalence relations between terms.\textsuperscript{27}

As Smith points out, this does not follow the semantical explanation of universes [Padova] according to which they are minimal types closed under certain specified type constructors. The difficulty with properly interpreting universes in LT seems to lie in the lack of a method for interpreting inductive definitions in LT. Another (slight) anomaly resulting from this lack is the fact that, unlike the other constructors, the membership in N is explicitly built into LT, which has a predicate, N(a), and an elimination rule. The terms satisfying N() have the forms 0 and suc(a). The interpretation of N is simply the formula (N(z') & [z' = z'']) , which works because equality between terms in LT is weak enough to permit any term satisfying N() to be normalized to a simple iteration of suc() from 0. The lack of inductive definition also excludes the interpretation of W-types.\textsuperscript{28} Finally, even if the entire hierarchy of universes had been interpreted by using a hierarchy of reflection predicates P_k and T_k,\textsuperscript{29} the inductive character of this hierarchy would be internal to LT.

The principal significance of LT is that it is known to have Frege structures as models. If one were not concerned with this one might attempt to extend LT radically in order to permit an interpretation that more closely followed the type-theoretic semantics. One possibility would be to add a means for expressing inductive definitions; an interpretation would make explicit the inductive nature of the definitions of N, W-types, universes, and universe hierarchies. Another would be to build directly into LT the various relational symbols, quantifiers, and rules necessary for expressing the types; an interpretation into this system would still serve to make explicit the structure of the hypothetical judgements of the type theory.

1.5 Notation

Our standard logical connectives and quantifiers are

\[ \forall x. \Phi(x) ; \exists x. \Phi(x) ; \text{not} \ \Phi \ ; \text{if} \ \Phi \ \text{then} \ \Psi ; \]

\[ \Psi \ \text{if} \ \Phi ; \Phi \ \& \ \Psi ; \ \Phi \ \text{or} \ \Psi ; \Phi \ \text{iff} \ \Psi . \]

\textsuperscript{27}A term r for which (\forall uv)P(r(u, v)) is true represents the relation (in a, b) of T(r(a, b)) being true.

\textsuperscript{28}The trick used for N would not work because equal members of a W-type may not be related by the equality relation of LT. Thus, we would again need an inductive definition of equality between W-type members.

\textsuperscript{29}And it seems likely that one could model this extension with Frege structures.
None of these has precedence over another, and the scope to the right is always large, therefore the infix operators are right associative. The pair "if — then" delimits scope in the way that matching "( — )" do. Indentation is used extensively to delimit scope, the rule of thumb being that the scope to the right includes the text on succeeding lines as long they are indented farther in than the operator.

The sign "=" is not used for identity; identity, and sometimes extensional equivalence between functions or relations, is indicated by "is".

We shall use slightly nonstandard notations for many complex higher order functions. They will usually look something like

$$\Phi(i.a_i; k, \Theta. \Psi(k; t, s.t\Theta s)),$$

where $\Phi$ is a "name" for the function (relation). The arguments are separated by "," and the variables preceding the dot in each argument bind the free occurrences in the expression to the right of the dot. When the variables on the right side of a dot are easily suppressed and the correspondence between binding and bound variables is clear, we may take the liberty of eliding them, e.g., $\Phi(a; \Psi)$ for the above example. An example of a substitution instance of the above notation, substituting for $a$ and for $\Psi$, is

$$\Phi(i.i + i^2; k, \Theta. \forall p. \exists r. k < r \text{ if } p\Theta r ).$$

Another example, this time substituting just for $\Phi$, is

$$\exists n. \Psi(a_n; t, s.t = s + s).$$

Performing all these substitutions, we get,

$$\exists n. \forall p. \exists r. n + n^2 < r \text{ if } p = r + r.$$  

We shall sometimes employ the following notation within a proof to indicate its structure:

- **Proof**: Introduces a proof.
- **arb x**: Introduces a hypothetical variable.
- **assume $\Phi$.**: Introduces a hypothesis.
- **arb x s.t. $\Phi$.**: Combines the previous two forms.
- **show $\Phi$.**: Indicates a goal.
- **enough to show $\Phi$.**: Indicates reduction of previous **show** or **enough**.
- **qed .**: Discharges last undischarged **show** or **enough**.
- **QED .**: Discharges all remaining hypotheses and goals.
A hypothesis is discharged by indenting it as well as indenting every line that follows until the hypothesis has been discharged. The exception is that a sequence of lines at the end of the proof need not be indented. We shall match (and nest) occurrences of show with qed, and with possibly intervening occurrences of enough. Each enough is meant to entail the previous (matching) enough or show and qed is meant to assert the previous (matching) enough or show. The exception is that QED matches all shows and enoughs that remain unmatched by qeds.
Chapter 2

Type Assignment Systems

Here we shall specify the kind of type assignment systems which form the core of our non-type-theoretic reinterpretation. The open-ended body of expression forms and type definitions that form the core of intuitionistic type theory is supplanted by a fixed class of terms and three fixed relations which replace the notions of evaluation, type equality, and member equality.

An evaluation system consists of its terms and their forms, and its evaluation relation, \( t \vdash s \), between terms. We do not suppose that these terms have any linguistic significance; they are just mathematical objects, and the terms of an evaluation system will typically be specified inductively. We say a term \( t \) is the value of term \( s \) when \( t \vdash s \). There are certain constraints on the evaluation relation which we shall include among our defining criteria for evaluation systems. If a term has a value then it has only one value, i.e., evaluation is a partial function. The value of a term is also its own value, i.e., \( t \vdash t \) if \( t \vdash s \). By canonical term we mean a term that is a value, or equivalently, a term that is its own value. When the partial function defined by the evaluation relation is computable, and identity of terms and the property of being canonical are computably decidable, we call the evaluation system a computation system. Although we will be interested primarily in computation systems, most of the ideas we develop here will make sense for any evaluation system, and this makes for a convenient level of abstraction.

The terms are built using variables and operators, which may bind variables in the usual sense, and of course, a term has finitely many subterms. Open terms, those with free variable occurrences, do not have values. The point of having open terms is that substitution for free variables is used. We shall assume that substituting terms for free variables in a term results in a term.

Let the simultaneous substitution of terms for free variables within a term be indicated by

\[
t[x_1 \ldots x_n/s_1 \ldots s_n]
\]
where \( 0 \leq n \), and \( x_1 \ldots x_n \) is a list (of length \( n \)) of variables (not necessarily distinct), and \( s_1 \ldots s_n \) is a list (of length \( n \)) of terms. It is handy to permit multiple occurrences of the same variable among the targets for replacement, all but the last ignored. The term \( t[s_1 \ldots s_n/x_1 \ldots x_n] \) is the result of replacing each free occurrence of \( x_i \) in \( t \) by \( r_i \), for \( 1 \leq i \leq n \), where \( r_i \) is \( s_j \), where \( j \) is the greatest \( k \leq n \) such that \( x_i \) is \( x_k \).

Let \( |\bar{s}| \) be the length of list \( \bar{s} \) of terms. We say term list \( \bar{s} \) is free for variable list \( \bar{x} \) in term \( t \), or free \( t[\bar{s}/\bar{x}] \), when \( |\bar{s}| \) is \( |\bar{x}| \) and the substitution of \( \bar{s} \) for \( \bar{x} \) into \( t \) is free of capture.

For convenience we shall also notate substitution into a list of terms. The list of terms \( (t_1 \ldots t_n)[\bar{s}/\bar{x}] \) is simply \( t_1[\bar{s}/\bar{x}], \ldots, t_n[\bar{s}/\bar{x}] \). And naturally, we shall use free \( t[\bar{s}/\bar{x}] \) to indicate the obvious relation. From time to time the following fact about substitution will be useful.

\[
t[\bar{r}/\bar{x}][\bar{s}/\bar{y}] = t[\bar{s}, \bar{r}[\bar{s}/\bar{y}] / \bar{y}, \bar{x}] \text{ if free } t[\bar{r}/\bar{x}] \text{ & } |\bar{s}| \text{ is } |\bar{y}|.
\]

Let us take a more detailed look at evaluation systems. An evaluation system consists of, or at least is determined by, five properties and relations:

- The evaluation relation \( t \leftrightarrow s \) between terms.
- The property \( \text{var}(t) \) of a term being a variable.\(^1\)
- A relation \( \text{form}(t; k; \bar{x}; \bar{e}) \) identifying the form (i.e. the operator, subterms and binding variables) of terms other than variables; \( \bar{e} \) is a list of terms and \( \bar{x} \) is a list of lists of variables. We shall not prescribe which objects \( k \) is to range over since the range may differ between evaluation systems.
- A function \( \#\text{subt}(k) \) giving the number of subterms taken by each operator.
- A function \( \#\text{bv}(k, i) \) giving the number of variables that become bound in each subterm of the terms of each form.

Let us say that an evaluation system skeleton consists of five such relations and properties. To form an evaluation system, these properties and relations must satisfy the relation

\(^1\)We shall implicitly restrict the range of \( u, v, w, z, y, z \), and variations such as subscripting and overbarring, to evaluation system variables, variable-valued functions, lists of variables, etc.
if \( \var(t) \& \var(s) \) then \( t \) is or isn’t \( s \)
& if \( \text{form} (t; k ; (r_{1,1} \ldots r_{1,m(t)}), \ldots, (r_{n,1} \ldots r_{n,m(n)}); e_1 \ldots e_l) \)
then not \( \var(t) \)
& \( \& n \) is \( l \) and is \( \#\text{subt}(k) \)
& \( \forall i < n. m(i + 1) = \#\text{bv}(k, i + 1) \)
& \( \forall j < m(i + 1). \var(r_{i+1,j+1}) \)
& if \( \text{form}(t; k; \bar{x}; \bar{e}) \& \text{form}(t'; k; \bar{x}; \bar{e}) \) then \( t \) is \( t' \)
& if \( \text{form}(t; k; \bar{x}; \bar{e}) \& \text{form}(t; k'; \bar{y}; \bar{s}) \) then \( k \) is \( k' \) & \( \bar{x} \) is \( \bar{y} \) & \( \bar{e} \) is \( \bar{s} \)
& \( \Phi(t) \) if \( \forall s, k, \bar{x}, \bar{e}, k_1 \ldots e_n. \)
\& \( \Phi(s) \) if \( \var(s) \) or \( \text{form}(s; k; \bar{x}; e_1 \ldots e_n) \)& \( \forall i < n. \Phi(e_{i+1}) \)
& if \( t \vdash s \) then \( s \) is closed & \( t \vdash t' \)
& if \( t \vdash s \) & \( t' \vdash s \) then \( t \) is \( t' \).

No variable is built from any of the indexed operators, and the operator index determines the number of immediate subterms and the number of variables that become bound in each. No two terms have the same form, and no term has two forms. All terms can be built from variables by finite iteration of operators. Terms with values are closed, evaluation is idempotent, and no term has two values.

Many of our definitions and assertions will be implicitly parameterized by evaluation system skeletons. When an evaluation system skeleton parameter is implicit we shall implicitly assume that it forms an evaluation system.

Substitution, capture, closedness, and \( \alpha \)-convertibility can all be defined from evaluation systems. For example,

\[
y[()]/() \text{ is } y \\
& y[\bar{e}, s/\bar{x}, y] \text{ is } s \\
& y[\bar{e}, s/\bar{x}, z] \text{ is } y[\bar{e}/\bar{x}] \text{ if } y \text{ is not } z \\
& \text{if } \text{form}(t; k; \bar{x}_1 \ldots \bar{x}_n; e_1 \ldots e_n) \text{ then} \\
& \text{form}(t[\bar{s}/\bar{z}]; k; \bar{x}_1 \ldots \bar{x}_n; r_1 \ldots r_n), \\
& \text{where } \forall i < n. r_{i+1} = e_{i+1}[\bar{s}, \bar{x}_{i+1}/\bar{z}, \bar{x}_{i+1}].
\]

A type assignment system, or type system for short, consists of an evaluation system, a two-place relation \( T = S \) on terms, and a three-place relation \( t = s \in T \) on terms such that

\[
T = S \text{ is transitive and symmetric} \\
& T = S \text{ if } \exists T'. T' \vdash T \& T' = S \\
& t = s \in T \text{ is transitive and symmetric in } t \text{ and } s \\
& t = s \in T \text{ if } \exists t'. t' \vdash t \& t' = s \in T \\
& T = T \text{ if } t = s \in T \\
& t = s \in T \text{ if } T = S \& t = s \in S.
\]
Let us say that an equality skeleton consists of a two-place and a three-place relation on terms, which need not satisfy the above conditions, and let us say that a type system skeleton consists of an evaluation system skeleton and an equality skeleton. Thus, a type system is a type system skeleton that satisfies the conditions given above. Many of our definitions and assertions will be implicitly parameterized by type system skeletons. When a type system skeleton parameter is implicit we shall implicitly assume that it forms a type system. And in keeping with our convention about implicit evaluation system parameters, if we say of an equality skeleton that it is a type system, what is meant is that its combination with the implicit evaluation system is a type system.

We say $T$ is a type, or "$T$ type", when $T = T$. We say $t$ is a member of $T$, or $t \in T$, when $t = t \in T$, and we say $T$ is inhabited, or $\exists t \in T$, when $\exists t. t \in T$. The relation $T = S$ is an equivalence relation when restricted to types, and $t = s \in T$ is an equivalence relation in $t$ and $s$ when restricted to members of $T$. We call $T = S$ type equality, and $t = s \in T$ member equality or equality in $T$ equality in $T$ or $T$ equality.

We say $A$ and $B$ are extensionally equal when

$$A \text{ type } & B \text{ type } \& \forall a, b. a = b \in A \iff a = b \in B.$$ 

The type equality defined in [HAN] is extensional, but in Nuprl it is not, i.e., it is intensional.\(^2\)

Let us compare type assignment systems with type definitions of the kind described in [HAN]. If the only kind of type equality we wanted to treat were extensional, as in [HAN], we would build type systems using the property of being a type, and the relation of type equality would be derivative. But I should like to accommodate intensional type equality as well. Another difference in approach is that the [HAN] view of type presents the membership and the equality of a given type as constituents of the definition of the type, and the equality in a type is presented as a relation only on the members. This may be necessary to Martin-Löf's open-ended foundational account, but in our non-type-theoretic account of type systems, type definition is considered complete and $t = s \in T$ is a single three-place relation on all terms.

It is easy to get the impression from [HAN] that the canonical members of types can be defined first, and the other members are such simply because they evaluate to canonical members. But in fact, because of lazy evaluation, one must sometimes use the concept of arbitrary member to arrive at that of

---

\(^2\) This condition has been weakened by Constable and Smith in their work on partial functions [Constable & Smith 87]. They allow for the definition of types which have members with no values.

\(^3\) For a discussion of intensional type equality see section 1.2, page 11.
canonical member. Of course, to define all canonical members of all types, one would expect to have to define all members of all types because of type constructors like \( T \times S \). The definition of the canonical members of \( T \times T \) depends on the definition of all members of \( T \). But it is not simply that the definition of canonical members of one type may depend on the definition of noncanonical members of another type. The definition of canonical members of \( N \) in [HAN] depends on the concept of arbitrary member of \( N \) itself; the canonical members of \( N \) are 0 and \( \text{suc}(t) \) for any member \( t \) of \( N \), canonical or not. So there is no sense in which the concept of canonical member is prior to that of member.

There is a similar situation with the canonical and noncanonical types. One expects the definition of all canonical types to depend on the definition of all noncanonical types because of type constructors that take noncanonical types as arguments. For example, \( T \times T \) is a canonical type just if \( T \) is a type, canonical or not. Is there a sense in which the concept of canonical type is prior to that of type? No definition in [HAN] of a canonical type \( T \) depends on the definition of a noncanonical type \( S \) that evaluates to \( T \); it is the definition of \( S \) which depends on \( T \). Here we see a sort of priority. But our notion of type assignment system suggests no such priority. Indeed we could define \( N \) to be a noncanonical term that evaluates to \( +(N) \), where \( +(T) \) is canonical for all \( T \), and

\[
  t = s \in +(T) \text{ iff } (0 \leftarrow t, s) \text{ or } \exists t', s'. \text{suc}(t') \rightarrow t \& \text{suc}(s') \rightarrow s \& t' = s' \in T. \quad 4
\]

(I have found \( +(T) \) quite handy in practice.) We could then define \( N \) to be a type, with the relation \( t = s \in N \) as strong as possible. This relation would be the same as in [HAN], but the priority of canonical over noncanonical types is obviously violated.

### 2.1 An Example: a System of Finite Types

I do not want to prescribe the form of definition for type systems, but all along I have had in mind recursive definition.\(^5\) We shall finish this section with a small example of a type system, \( \text{FIN} \), of finite types over \( N \) and initial segments of \( N \). This example is meant to be somewhat exotic (to those of us accustomed to using typed terms or the languages of [HAN] or Nuprl) in order to help us avoid making inappropriate assumptions about type systems and type system definitions. A definition of a more orthodox type system, namely, the system of types built using the type constructors

---

4 By \( r \leftarrow t, s \) is meant that \( r \) is the value of \( t \) and of \( s \).

5 Martin-Löf's explanations of the expressions and type definitions of his [HAN] theory are not intended as recursive definitions of mathematical systems of objects and relations.
defined in [HAN], may be found on page 48 in the chapter about universes. The evaluation system of our example has denumerably many variables and three operators, namely, $A \rightarrow B$ and $t(a)$, which are not binding operators, and $\lambda x b$, which binds the free $x$ in $b$. Evaluation is the strongest relation $t\rightarrow s$ such that

$$\lambda x b \rightarrow \lambda x b \text{ if } \lambda x b \text{ is closed}$$
$$\& (A \rightarrow B) \rightarrow (A \rightarrow B) \text{ if } A \rightarrow B \text{ is closed}$$
$$\& e \rightarrow t(a) \text{ if } \tau \rightarrow t \& e \rightarrow \tau(a)$$
$$\& e \rightarrow (\lambda x b)(a) \text{ if } e \rightarrow b[a/x] \& a \text{ is closed.}$$

Obviously, this evaluation system is a computation system.

We shall encode numbers. Let $0_{x,y}$ be $\lambda x \lambda y z$, and let $\text{suc}_{x,y,z}(t)$ be $\lambda x \lambda y \lambda z (\lambda z(y(z)((z(x))(y))))(t)$. If $x$ is not $y$, $0_{x,y}$ will represent zero. If $x, y, z$ are distinct then $\text{suc}_{x,y,z}(t)$ will represent the successor of any number represented by $t$. The idea is that $n(b)(f)$ is the value at $n$ of the partial function defined by primitive recursion with base case $b$ and inductive step $f$. In our little example we shall have a type of these numbers, and each number will itself be the type of smaller numbers, rather like the usual encoding of finite ordinals in set theory. Type equality will be extensional.

Our sample type system may be given by the following recursive definition.

$A$ type if $B$ type $\& B \rightarrow A$

$0_{x,y}$ type if $x, y$ are distinct

$C_0$ type if $\forall i. \exists x, y, z$ distinct. $\text{suc}_{x,y,z}(C_{i+1}) \rightarrow C_i$

$\text{suc}_{x,y,z}(A)$ type if $A$ type $\& x, y, z$ are distinct

$A \rightarrow B$ type if $A, B$ type

$a = b \in A$ if $a = b \in B \& B \rightarrow A$

$a = b \in A$ if $c = b \in A \& c \rightarrow a$

$a = b \in A$ if $a = c \in A \& c \rightarrow b$

$0_{u,v} = 0_{u',v'} \in \text{suc}_{x,y,z}(A)$ if $A$ type $\& x, y, z$ distinct

$\& u, v$ distinct $\& u', v'$ distinct

$\text{suc}_{u,v,w}(a) = \text{suc}_{u',v',w'}(b) \in \text{suc}_{x,y,z}(A)$ if $a = b \in A$

$\& x, y, z$ distinct

$\& u, v, w$ distinct

$\& u', v', w'$ distinct
\( t = s \in A \rightarrow B \) if \( A, B \) type & \( t \) and \( s \) have values & \( \forall a, b. \ t(a) = s(b) \in B \) if \( a = b \in A \)
\[ A = B \] if \( A, B \) type & \( \forall a, b. \ a = b \in A \) iff \( a = b \in B \).

Let 0 be \( 0_{z,y} \), and let \( N \) be
\[ (\lambda w \ \text{succ}_{z,y,z}(w(w)))(\lambda w \ \text{succ}_{z,y,z}(w(w))) \]
for some distinct \( w, x, y, z \). 0 is an empty type, and the equal members of \( N \) are the encodings of equal numbers. The members of a type \( A \rightarrow B \) are the terms that have values whose applications are \( B \) functional over \( A \) in the argument; equality is extensional. An odd feature of this type system is that \( t = s \in 0 \rightarrow 0 \) iff \( t \) and \( s \) have values, so for example, every type and every member of every type is a member of \( 0 \rightarrow 0 \).

This definition is not as clear as possible. If we were to take the clauses of the above definition as properties of type systems, we would find that there are no strongest type and member equalities satisfying them because of the non-monotonicity in \( a = b \in A \) of the clause defining \( A \rightarrow B \) equality. For example, let \( d \) be a closed term that has no value. The intended definiendum is a type system for which 0 is empty and, therefore, \( 0 \rightarrow 0 \) has \( \lambda x d \) as a member. We can also find a type system that satisfies the clauses of the definition and for which 0 has 0 as a member and \( 0 \rightarrow 0 \) does not have \( \lambda x d \) as a member. No type system satisfying the clauses of the definition can have a member equality that is as strong as the member equalities of both these type systems. The remainder of this section is devoted to making the definition more explicit in order to make clear the nature of the induction. Readers may omit it without much loss.

The clause defining \( A = B \) does not contribute to the inductive aspect of the definition and we may define \( A = B \) after inductively defining typehood and member equality. Let
\[
\text{FINbody}(\ A, a, b. \ \psi'(A; a; b) \ ; \ A, a, b. \ \psi(A; a; b) \ ; \ A. \ \phi'(A) \ ; \ A. \ \phi(A) \ ),
\]
which we may abbreviate as \( \text{FINbody}(\psi'; \psi; \phi'; \phi) \), be the relation
\( \forall A, B, C, a, b, c, t, s, x, y, z, u, v, w, u', v', w' \).

if \( x, y, z \) distinct & \( u, v, w \) distinct & \( u', v', w' \) distinct
then \( \phi(A) \) if \( \phi(B) \) & \( B \vdash \)
\( \& \phi(0_{x,y}) \\
\& \phi(C_0) \) if \( \forall i. \exists x, y, z \) distinct. \( \text{suc}_{x,y,z}(C_{i+1}) \vdash C_i \\
\& \phi(\text{suc}_{x,y,z}(A)) \) if \( \phi(A) \\
\& \phi(A \vdash B) \) if \( \phi'(A) \) & \( \phi(B) \\
& \psi(A; a; b) \) if \( \psi(B; a; b) \) & \( B \vdash \)
& \psi(A; a; b) \) if \( \psi(A; c; b) \) & \( c \vdash a \\
& \psi(A; a; b) \) if \( \psi(A; a; c) \) & \( c \vdash b \\
& \psi(\text{suc}_{x,y,z}(A); 0_{u,v}; 0_{u',v'}) \) if \( \phi(A) \\
& \psi(\text{suc}_{x,y,z}(A); \text{suc}_{u,v,w}(a); \text{suc}_{u',v',w'}(b)) \) if \( \psi(A; a; b) \\
& \psi(A \vdash B; t; s) \) if \( \phi'(A) \) & \( \phi(B) \) & \( t \) and \( s \) have values
\& \( \forall a, b. \psi(B; t(a); s(b)) \) if \( \psi'(A; a; b) \).

The \( \psi' \) indicates the previously mentioned non-monotonic occurrence. Notice that typehood of a particular term and equality in a particular type depend only upon the typehood of and equalities in finitely many other terms. This permits us to stratify typehood into a sequence of properties and member equality into a sequence of subrelations. Of course, this is not possible in Nuprl or [HAN] because types may be formed from infinite families of other types.

For all \( n \), let \( \mu_n \) and \( \tau_n \) be the strongest \( \psi \) and \( \phi \) such that

\[
\text{FINbody}(\mu_{\exists n}; \psi; \tau_{\exists n}; \phi),
\]

where \( \mu_{\exists n}(A; a; b) \) is \( \exists i < n. \mu_i(A; a; b) \), and \( \tau_{\exists n}(A) \) is \( \exists i < n. \tau_i(A) \). It should be clear that this defines \( \mu_n \) and \( \tau_n \). We may then define \( \text{FIN} \) by

\( A \text{ type}_{\text{FIN}} \) iff \( \exists n. \tau_n(A) \)

and

\( a = b \in_{\text{FIN}} A \) iff \( \exists n. \mu_n(A; a; b) \)

and

\( A =_{\text{FIN}} B \) iff \( A, B \) type_{\text{FIN}} \& \( \forall a, b. a = b \in_{\text{FIN}} A \) iff \( a = b \in_{\text{FIN}} B \).
Chapter 3

Functionality and Sequents

When developing a formalized language of type assignment, one must consider the kinds of assertion to be formalized. The kinds with which we shall here concern ourselves will be similar to the forms of judgement of [HAN], except that our assertions are about fixed relations over a fixed class of terms, and are defined without presuppositions. The key concept behind these assertions is that of functionality.\(^1\) We say a term \(T\) is type functional over term \(A\) in variable \(x\) when

\[
A \text{ type } \& \forall a, a'. \ T[a/x] = T[a'/x] \text{ if } a = a' \in A.
\]

We say a term \(t\) is \(T\) functional over \(A\) in \(x\) when

\[
T \text{ is type functional over } A \text{ in } x
\]

and

\[
\forall a, a'. \ t[a/x] = t[a'/x] \in T[a/x] \text{ if } a = a' \in A.
\]

Martin-Löf's \(\Pi\) and \(\Sigma\) type constructors depend upon functionality. The terms \((\Pi x \in A) \ B\) and \((\Sigma x \in A) \ B\) are types just when \(B\) is type functional over \(A\) in \(x\). And the canonical members of a type \((\Pi x \in A) \ B\) are the closed terms of form \((\lambda y)b\) such that the application \(((\lambda y)b)(x)\) is \(B\) functional over \(A\) in \(x\).

The main kinds of assertions that we will consider consist in the attribution of certain properties to sequents. For the sake of a perspicuous definition of functionality, we shall "double" the components of a sequent. A sequent is either the atomic sequent \(\bullet\) or a pair of terms \(T = S\) or a pair of terms \(\exists T = S\) or a quadruple of terms \(t = s \in T = S\) or else a quadruple \(A,B:x \sigma\) where \(A\) and \(B\) are terms, \(x\) is a variable, and \(\sigma\) is a sequent. Sequents of the first four forms we shall call simple, while those of the fifth form we shall

\(^1\)Pointwise functionality, the notion expressed in Nuprl by use of assumptions, will be treated last, in chapter 8.
call complex. Occurrences of variables in a sequent are free or bound. The free variables of a sequent $A, B : x \sigma$ are those free in $A$, those free in $B$, and those other than $x$ free in $\sigma$. The free variables of a simple sequent are those free in the component terms.

The simple sequent $T = T$ may be designated by "$T$ type", the simple sequent $T = T$ by "$T$ type", the simple sequent $t = s \in T = T$ by "$t = s \in T$", and the simple sequent $t = t \in T = T$ by "$t \in T$". The complex sequent $A, A : x \sigma$ may be designated by "$A : x \sigma$". In practice, it is expected that all (sub-sequents of) the sequents used to make assertions will have one of these forms or the form $T = S$. The more general form of sequent is used here in order to conveniently give a precise treatment of the properties that those assertions attribute to those sequents.

By declaration (of $x$) is meant a triple $A, B : x$ where $A$ and $B$ are terms, and $x$ is a variable. It is handy to be able to build a sequent from a list of declarations. Let us define the sequent-valued function $\Delta \triangleright \sigma$, where $\sigma$ ranges over sequents and $\Delta$ over declaration lists, such that

$$(\cdot) \triangleright \sigma \text{ is } \sigma$$

and

$$(A, B : x \Delta) \triangleright \sigma \text{ is } A, B : x (\Delta \triangleright \sigma).$$

Clearly, each sequent is $\Delta \triangleright \sigma$ for only one simple sequent $\sigma$, and for only one declaration list $\Delta$ given that $\sigma$ is simple. If $\sigma$ is a simple sequent, then the declarations of $\Delta \triangleright \sigma$ are the declarations in the list $\Delta$, and $\sigma$ is the consequent of $\Delta \triangleright \sigma$. Other handy functions of declaration lists are $|\Delta|$, which gives the number of declarations in $\Delta$, and $\hat{\Delta}$, which extracts the list of declared variables, i.e., $\hat{\Delta}$ is $x_1 \ldots x_n$ if $\Delta$ is $\{A_{i+1}, B_{i+1} : x_{i+1}\}_{i < n}$. We shall take the liberty of abbreviation $(\Delta \triangleright \cdot)$ by $\Delta$.

We shall define a property of sequents, $F_n \sigma$, which is an extension to the concepts of functionality defined on page 35. It is modeled after the judgement forms of $[HAN]$. $F_n$ will be defined by induction on the number of declarations. The definition of $F_n$ at a sequent $A, B : x \sigma$ depends on its values at substitution instances of $\sigma$; we shall need a kind of parallel substitution function on sequents. Define $\sigma[\bar{a}; \bar{b}/\bar{x}]$, where $\sigma$ is a sequent, $\bar{a}$ and $\bar{b}$ are lists (length $n$) of terms, and $\bar{x}$ is a list (length $n$) of variables, to be the function such that

$$(\cdot)[\bar{a}; \bar{b}/\bar{x}] \text{ is } \cdot$$

and

$$(T = S)[\bar{a}; \bar{b}/\bar{x}] \text{ is } T[\bar{a}/\bar{x}] = S[\bar{b}/\bar{x}]$$

$^2$By $\{\delta_i\}_{i < n}$, where $\delta_i$ is a declaration-valued function in $i$ over the natural numbers, and $n$ is a natural number, is meant the list that is empty if $n$ is 0, and is $\delta_0 \ldots \delta_{n-1}$ otherwise.
and
\[(\exists \in T = S)[\overline{a}; \overline{b}/\overline{x}] \text{ is } \exists \in T[\overline{a}/\overline{x}] = S[\overline{b}/\overline{x}]\]
and
\[(t = s \in T = S)[\overline{a}; \overline{b}/\overline{x}] \text{ is } t[\overline{a}/\overline{x}] = s[\overline{b}/\overline{x}] \in T[\overline{a}/\overline{x}] = S[\overline{b}/\overline{x}]\]
and
\[(A, B : z \sigma)[\overline{a}; \overline{b}/\overline{x}] \text{ is } A[\overline{a}/\overline{x}], B[\overline{b}/\overline{x}]: z \sigma[\overline{a}, z; \overline{b}, z/\overline{x}, z].\]

Note that \((\Delta \circ \sigma)[\overline{a}; \overline{b}/\overline{x}] \text{ is } \Delta[\overline{a}; \overline{b}/\overline{x}] \circ \sigma[\overline{a}, \Delta; \overline{b}, \Delta/\overline{x}, \Delta].\) We shall define \(\sigma[\overline{a}/\overline{x}]\) to be \(\sigma[\overline{a}; \overline{a}/\overline{x}]\), and we shall use free \(\sigma[\overline{a}; \overline{b}/\overline{x}]\) and free \(\sigma[\overline{a}/\overline{x}]\) with the obvious senses similar to free \(t[\overline{s}/\overline{x}]\).

We now define Fn.

Fn •.

Fn \(T = S\) iff \(T = S\).

Fn \(\exists \in T = S\) iff \(T = S \& \exists \in T\).

Fn \(t = s \in T = S\) iff \(T = S \& t = s \in T\).

Fn \(A, B : x \sigma\) iff \(A = B \& \forall a, b. \text{ Fn } \sigma[a;b/x]\) if \(a = b \in A\).

It may prove informative to see Fn unwrapped. By induction on \(n\),

\[
\text{Fn } \{A_{i+1}, B_{i+1} : x_{i+1}\}_{i<n} \circ \sigma \\
\text{ iff } \text{Fn } \{A_{i+1}, B_{i+1} : x_{i+1}\}_{i<n} \\
\& \forall a_1 \ldots a_n, b_1 \ldots b_n. \\
\text{Fn } \sigma[a_1 \ldots a_n; b_1 \ldots b_n/x_1 \ldots x_n] \\
\text{ if } \forall i < n. a_{i+1} = b_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i].
\]

Let us compare Fn with the forms of judgement of [HAN], which are written as:

\(T\) type \((x_1 \in A_1 \ldots x_n \in A_n)\)

\(T = S\) \((x_1 \in A_1 \ldots x_n \in A_n)\)

\(t \in T\) \((x_1 \in A_1 \ldots x_n \in A_n)\)

\(t = s \in T\) \((x_1 \in A_1 \ldots x_n \in A_n)\).
The judgement $T$ type () means that $T$ evaluates to a defined canonical type. Our analog for this is $\text{Fn } T$ type. The judgement $T = S$ () is given meaning presupposing that $T$ type () and $S$ type (); under that presupposition, it means that $T$ and $S$ evaluate to canonical types that have the same canonical objects and the same pairs of equal canonical objects. The assertion $\text{Fn } T = S$ is similar except that its meaningfulness does not depend on $T$ and $S$ being types; also, the type equality of the type system need not be extensional. The meaningfulness of $t \in T$ () presupposes that $T$ type (), and under that presupposition, means that $t$ evaluates to a canonical object of the canonical type to which $T$ evaluates. The assertion $\text{Fn } t \in T$ is the analog, but its meaningfulness does not depend on $T$ being a type. The meaningfulness of $t = s \in T$ () presupposes that $T$ type (), $t \in T$ (), and $s \in T$ (), and it means that the values of $t$ and $s$ are equal in the type that is the value of $T$. The analog is $\text{Fn } t = s \in T$, and again, its meaningfulness has not been made to depend upon whether $T$ is a type or whether $t$ and $s$ are members of $T$. Note that in all these cases, the $\text{Fn}$ analog of a $[\text{HAN}]$ judgement implies the analogs of the presuppositions of that judgement.

The meaningfulness of a judgement of $[\text{HAN}]$ with hypotheses

$$(x_1 \in A_1 \ldots x_n \in A_n)$$

presupposes that

$$\text{for } i < n, \ A_{i+1} \text{ type } (x_1 \in A_1 \ldots x_i \in A_i),$$

the $\text{Fn}$ analog of which is that $\text{Fn } A_1 : x_1 \ldots A_n : x_n$. In contrast, $\text{Fn}$ is meaningful for any sequent and $\text{Fn } \Delta \triangleright \sigma$ implies $\text{Fn } \Delta$. The judgement

$$T \text{ type } (x_1 \in A_1 \ldots x_n \in A_n)$$

means that for any $a_1 \ldots a_n$ such that

$$\text{for } i < n, \ a_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i] () ,$$

$$T[a_1 \ldots a_n/x_1 \ldots x_n] \text{ type } () ,$$

and moreover, for any $a_1 \ldots a_n$ and $b_1 \ldots b_n$ such that

$$\text{for } i < n, \ a_{i+1} = b_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i] () ,$$

$$T[a_1 \ldots a_n/x_1 \ldots x_n] = T[b_1 \ldots b_n/x_1 \ldots x_n] () .$$

The obvious analog is $\text{Fn } A_1 : x_1 \ldots A_n : x_n \triangleright T$ type. The judgement

$$T = S \ (x_1 \in A_1 \ldots x_n \in A_n)$$
presupposes that

\[ T \text{ type } (x_1 \in A_1 \ldots x_n \in A_n) \text{ and } S \text{ type } (x_1 \in A_1 \ldots x_n \in A_n). \]

And it means that for any \( a_1 \ldots a_n \) such that

for \( i < n \), \( a_{i+1} \in A_{i+1}[a_1 \ldots a_i/x_1 \ldots x_i] \),

\[ T[a_1 \ldots a_n/x_1 \ldots x_n] = S[a_1 \ldots a_n/x_1 \ldots x_n] \).

The analog, \( \text{Fn } A_1 : x_1 \ldots A_n : x_n \Rightarrow T \) type, implies the analogs of the presuppositions since, as will be seen, \( \text{Fn } \Delta \Rightarrow T = S \) implies \( \text{Fn } \Delta \Rightarrow T \) type and \( \text{Fn } \Delta \Rightarrow S \) type. The situation is similar for the two other forms of hypothetical judgement in [HAN].

The Fn analog of a [HAN] judgement implies the analogs of the presuppositions of that judgement. This is appropriate when we want to adopt forms of judgement approximating those of [HAN], yet avoid the complexity of having all those presuppositions. After all, if one wanted to do away with presuppositions for meaningfulness, then a judgement should either imply the erstwhile presuppositions or else hypothesize them, and the former would seem to result in a practice more similar to the use of the [HAN] theory than would the latter.

### 3.1 Fn Facts

This section will consist of a short study of Fn. Familiarity with the following facts about Fn is, in the author's opinion, important for using Fn fluently.\(^3\) These facts will be applied tacitly in later chapters. From these facts, it will be easy to see that all the general rules of inference given in [HAN] would be correct if the judgements were reinterpreted as their Fn analogs.

In order to indicate optional forms of simple sequents, we shall use a variable \( t = s \in T = S \) ranging over four-place sequent-valued functions\(^4\) and we shall restrict it by assuming that

\[ \forall t, s, T, S. \ t = s \in T = S \text{ is } T = S \]

\(^3\)Most of the facts follow easily from statements of the form

\[ \forall \Delta, \sigma. \text{Fn } \Delta \Rightarrow \sigma_0 \text{ if } \Psi(i, \sigma_i) \land \forall i < k + 1. \text{Fn } \Delta \Rightarrow \sigma_{i+1}, \]

for some \( k \) and some \( \Psi \) such that

\[ \forall \sigma, a, b, z. \Psi(i, \sigma_i[a; b/z]) \text{ if } \Psi(i, \sigma_i) \land a, b \text{ are closed.} \]

By induction on \( |\Delta| \), such a statement follows from the base case (empty \( \Delta \)).

\(^4\)We have already been using \( t = s \in T = S \) as such a variable.
or
\[
\forall t, s, T, S.\, t = s \in T = S \text{ is } \exists \in T = S
\]
or else
\[
\forall t, s, T, S.\, t = s \in T = S \text{ is } t = s \in T = S.
\]
We shall also assume that \( t = s \in T \) is \( t = s \in T = T \), and that \( t \in T \) is \( t \in T \).

The dagger \( \dagger \) and double dagger \( \ddagger \) prefixed to some of the statements below indicate statements which are true not only of \( \text{Fn} \), but also of another property of sequents defined in a later chapter.\(^5\)

\( \dagger \) \text{Fn} \( \Delta \) if \( \text{Fn} \Delta \gg \sigma \).\(^6\)

\( \dagger \) The relation \( \text{Fn} \Delta \gg t = s \in T = S \) is symmetric and transitive in \( t, T \) and \( s, S \).

\( \dagger \) \text{Fn} \( \Delta \gg t = s \in T = S \) if \( \text{Fn} \Delta \gg s = t \in S = T \).\(^7\)

\( \dagger \) \text{Fn} \( \Delta \gg t = s \in T = S \) if \( \text{Fn} \Delta \gg t = r \in T = R \)

\& \text{Fn} \( \Delta \gg r = s \in R = S \).\(^8\)

Combining symmetry and transitivity gives us that

---

\(^5\) Statements so prefixed remain true when the property \( \text{Fn} \sigma \) is replaced by \( \text{PwFn} (m; \sigma) \), which is defined in chapter 8, for \( m \leq |\Delta| \) if the prefix is \( \dagger \), and for any \( m \) if the prefix is \( \ddagger \).

\(^6\) This corresponds to the presupposition by a hypothetical judgement in [HAN] about its assumptions. Also, \( \text{PwFn} \Delta \gg \).

\(^7\) To prove this (without the dagger), prove the lemma

\[
\text{Fn} \left\{ A_{i+1}, B_{i+1} : x_{i+1} \right\}_{i < n} \gg t = s \in T = S \\
\text{if} \quad \text{Fn} \left\{ C_{i+1}, D_{i+1} : x_{i+1} \right\}_{i < n} \gg s = t \in S = T \\
\& \quad \text{Fn} \left\{ A_{i+1}, D_{i+1} : x_{i+1} \right\}_{i < n} \\
\& \quad \text{Fn} \left\{ A_{i+1}, B_{i+1} : x_{i+1} \right\}_{i < n},
\]

then identify \( A \) with \( C \) and \( B \) with \( D \). The trick to proving this lemma, by induction on \( n \), is this — when proving the induction step one reaches a point where one tries to show, by applying the induction hypothesis, that for \( a \) and \( b \) such that \( a = b \in A_1 \),

\[
\text{Fn} \left\{ \left\{ A_{i+2}, B_{i+2} : x_{i+2} \right\}_{i < n} \gg t = s \in T = S \right\}[a; b/x_1];
\]

but, to do this one must observe the identities between various terms mentioned in the induction hypothesis. Use

\[
\text{Fn} \left\{ \left\{ C_{i+2}, D_{i+2} : x_{i+2} \right\}_{i < n} \gg s = t \in S = T \right\}[b; a/x_1],
\]

\[
\text{Fn} \left\{ \left\{ A_{i+2}, D_{i+2} : x_{i+2} \right\}_{i < n} \right\}[a; a/x_1],
\]

and

\[
\text{Fn} \left\{ \left\{ A_{i+2}, B_{i+2} : x_{i+2} \right\}_{i < n} \right\}[a; b/x_1].
\]

This explains the choice of lemma.

\(^8\) To prove this (without the dagger), prove the lemma
† if $\text{Fn } \Delta \triangleright t = s \in T = S$ then $\text{Fn } \Delta \triangleright t \in T$ & $\text{Fn } \Delta \triangleright s \in S$.\(^9\)

Type-valued functions can be replaced by equal type-valued functions in certain obvious places throughout a sequent.

† $\text{Fn } \Delta \triangleright t = s \in T = T'$ if $\text{Fn } \Delta \triangleright t = s \in S = S'$ & $\text{Fn } \Delta \triangleright T = S'$ & $\text{Fn } \Delta \triangleright S = T'$.

† $\text{Fn } \Delta \triangleright A, A': x \sigma$ if $\text{Fn } \Delta \triangleright B, B': x \sigma$ & $\text{Fn } \Delta \triangleright A = B'$ & $\text{Fn } \Delta \triangleright B = A'$.

Equality of type-valued functions can be explicitly extracted from sequents.

† $\text{Fn } \Delta \triangleright T = T'$ if $\text{Fn } \Delta \triangleright t = s \in T = T'$.\(^10\)

† $\text{Fn } \Delta \triangleright A = A'$ if $\text{Fn } \Delta \ A, A': x \triangleright \sigma$.

With these theorems in hand we should stop annoying ourselves with all this doubling of types when in practice we expect to use doubling only in sequents of form $T = S$.\(^11\)

\[
\text{Fn } \{A_i, B_i_1 : z_i_1\}_{i < n} \triangleright t = s \in T = S
\]

if $\text{Fn } \{A_i, C_i_1 : z_i_1\}_{i < n} \triangleright t = r \in T = R$

& $\text{Fn } \{A_i, B_i_1 : z_i_1\}_{i < n} \triangleright r = s \in R = S$.

and then identify $B$ with $C$. In the proof of the induction step, when trying to show

$\text{Fn } \{\{A_i, B_i_2 : z_i_2\}_{i < n} \triangleright t = s \in T = S\}[a; b/z_1],$

use

$\text{Fn } \{\{A_i, C_i_2 : z_i_2\}_{i < n} \triangleright t = r \in T = R\}[a; a/z_1]$

and

$\text{Fn } \{\{A_i, B_i_2 : z_i_2\}_{i < n} \triangleright r = s \in R = S\}[a; b/z_1].$

\(^9\)This corresponds to the presupposition by $T = S (...)$ that $T$ type $(...)$, and the presupposition by $t = s \in T (...)$ that $t \in T (...) \& s \in T (...)$.  

\(^10\)This corresponds to the presupposition by $t \in T (...)$ that $T$ type $(...)$.  

\(^11\)From the previous four theorems it follows that

† $\text{Fn } \Delta \triangleright t = s \in T = T'$ iff $\text{Fn } \Delta \triangleright T = T'$ & $\text{Fn } \Delta \triangleright t = s \in T$

and

† $\text{Fn } \Delta \triangleright A, A': x \sigma$ iff $\text{Fn } \Delta \triangleright A = A'$ & $\text{Fn } \Delta \triangleright A : x \sigma$.  

If type equality is extensional
then \( \text{Fn } \Delta \triangleright A = B \) if \( x \) is not free in \( A, B \)
& \( \text{Fn } \Delta \triangleright B : \bar{x} . x \in A \)
& \( \text{Fn } \Delta \triangleright A : \bar{x} . x \in B \). \(^{12}\)

\[ \vdash \text{Fn } \Delta \triangleright \exists \bar{x} \in T \text{ if } \text{Fn } \Delta \triangleright t = s \in T. \]

Fn is preserved under change of declared variables.

\[ \vdash \text{Fn } \Delta \triangleright A : y . \sigma[y/x] \text{ if } \text{Fn } \Delta \triangleright A : x . \sigma \]
& \( \text{free } \sigma[y/x] \)
& \( y \) is \( x \) if \( y \) is free in \( \sigma \).

Sequents may be composed with functional expressions.

\[ \vdash \text{Fn } \Delta \triangleright \sigma[t; t'/z] \text{ if } \text{free } \sigma[t; t'/z] \]
& \( \text{Fn } \Delta \triangleright t = t' \in T \)
& \( \text{Fn } \Delta \triangleright T : z . \sigma \).

Similarly,

\[ \vdash \text{Fn } \Delta \triangleright \sigma \text{ if } z \text{ not free in } \sigma \]
& \( \text{Fn } \Delta \triangleright \exists \bar{z} \in T \)
& \( \text{Fn } \Delta \triangleright T : z . \sigma \).

The remainder of the Fn facts justify the use of natural deduction, the proof style of [HAN].

\[ \vdash \text{Fn } \Delta \ A : x . \Delta' \triangleright x \in A \text{ if } \text{Fn } \Delta \ A : x . \Delta' \]
& \( \Delta' \) not free in \( x \in A \)
& \( x \) not free in \( A \).

\[ \vdash \text{Fn } \Delta \ A : x \text{ if } \text{Fn } \Delta \triangleright A \text{ type}. \]

The last couple of facts provide means for merging and permuting declaration lists.

\[ \vdash \text{Fn } \Delta \Delta' \triangleright \sigma \text{ if } \Delta' \text{ not free in } \sigma \]
& \( \text{Fn } \Delta \triangleright \sigma \)
& \( \text{Fn } \Delta \Delta' \). \(^{16}\)

\(^{12}\) This corresponds to no rule of [HAN].

\(^{13}\) This justifies the substitution rules of [HAN]. For a related fact about PwFn, see (I), page 98.

\(^{14}\) For a related fact about PwFn, see (1'), page 98.

\(^{15}\) This justifies the assumption rule of [HAN]. Also, PwFn \( \Delta \triangleright \triangleright \) (see page 97).
$\dagger \text{Fn } \Delta \triangleright A : x B : y \sigma \text{ if } \text{Fn } \Delta \triangleright B : y A : x \sigma$
& \text{Fn } \Delta \triangleright A : x
& \text{x not free in } B
& \text{y not free in } A
& (x \text{ is not } y) \text{ or } (x, y \text{ not free in } \sigma).^{17}

^{16} \text{For a related fact about PwFn, see (J), page 98.}
^{17} \text{For a related fact about PwFn, see (K), page 98.}
Chapter 4

Universes

It seems that people using a type system sooner or later want to use complexes that have types as components (e.g., pairs \((A, a)\) such that \(a \in A\)), and want to use functions taking types as arguments (e.g., a definition by primitive recursion of a sequence of types \(T_n\) uses a function \(H(n; A)\) such that \(T_{n+1}\) is \(H(n; T_n)\)), and so forth. Let us call this practice the use of types as objects. One could modify \(F_n\) and the forms of sequents to accommodate some judgements about these kinds of objects; but if we were to do this to our satisfaction, we would find ourselves duplicating the entire apparatus of type assignment. If there were a type \(V\) of all types, i.e., \(A = B \in V\) iff \(A = B\), the problem would be solved. But there is no type system that has such a type and yet is closed under the type constructors of [HAN]. (Martin-Löf [Martin-Löf 72] adapted a proof from [Girard 72].)

But there is another way. In [Martin-Löf 73] Martin-Löf explains the method of introducing universes, which are types whose equal members are equal types, and which are closed under the type constructors that we were accustomed to using prior to the introduction of the universes. Martin-Löf calls this closure property of universes the reflection principle. If we first add universe \(V_0\), we get a new type constructor (\(V_0\) formation) and we may then want to reflect the extended type system in a universe \(V_1\), and so forth, giving rise to a sequence of universes \(V_i\). We shall develop this method in our framework for type assignment.

This chapter also includes a definition of the type system of [HAN], or rather, the one that results from using the computation system built using only the term constructors already introduced in [HAN] and using only the type constructors already defined there. This type system is defined here rather than in the chapter about type assignment systems because the use of universes is essential to it.

We shall also introduce a means for expressing a certain kind of polymorphism with regard to universes. This is intended primarily to save users the
4.1 Universes and Type System Hierarchies

A hierarchy of type systems can be produced methodically once one knows the constructors under which all the systems are to be closed and one knows the sequence of canonical terms to be used as universes. Although it is not necessary, we shall require that a unique canonical universe be introduced at each level of the hierarchy since it simplifies our work and since it is adequate for the purpose of the hierarchy.

We must do a bit of preparation. Recall that a type system was defined as consisting of an evaluation system and two relations on terms satisfying certain properties. An equality skeleton consists of a two-place and a three-place relation on terms. Thus, a type system is an evaluation system together with an equality skeleton satisfying certain relations. We need the distinction because we shall use some functions intended to have type systems as values, but which may, on some arguments, produce skeletons that do not meet the additional criteria for producing type systems.

When we need to discuss relations between equality skeletons, we may indicate the type equality of skeleton $T$ by $A =_T B$ and indicate the equality in $A$ by $a = b \in TA$. Similarly, $A$ type$_T$ means that $A =_T A$, while $a \in TA$ means that $a = a \in TA$.

We say a skeleton $T$ is an extension of skeleton $S$, or $T$ ext $S$, when

$$\forall A, B. \text{ if } A, B \text{ type}_S$$
$$\text{ then } A =_T B \iff A =_S B$$
$$\& \forall a, b. a = b \in TA \iff a = b \in SA.$$

The relation $T$ ext $S$ is reflexive and, when restricted to type systems, transitive.

The empty type system $\bot$ is the type system with no types, i.e., the relations $A =_\bot B$ and $a = b \in \bot A$ are everywhere false. Every equality skeleton is an extension of $\bot$.

Let $T \cup S$ be the skeleton-valued function such that

$$A =_{T \cup S} B \iff A =_T B \text{ or } A =_S B$$

and

$$a = b \in _{T \cup S} TA \iff a = b \in TA \text{ or } a = b \in SA.$$

1There is an empty type system $\bot_E$ for each evaluation system $E$. 
If \( T \) ext \( S \) then \( T \cup S \) is \( T \).

By \( V \) reflect \( T \), where \( V \) is a term and \( T \) is a skeleton, is meant the equality skeleton such that

\[
A =_\text{reflect} T B \iff V \to A, B
\]

and

\[
a = b \in _\text{reflect} T A \iff V \to A \& a =_T b.
\]

We shall assume that the body of type construction methods from which a hierarchy is to be constructed can be expressed as a one-place skeleton-valued function of skeletons \( S(T) \), which we shall call a hierarchy generator. The intention is that we be able to insert new types through the argument \( T \).

Define \( (T.S(T))^*(i.V_i)_n \), which we may abbreviate by \( S^*(i.V_i)_n \), where \( S(T) \) ranges over one-place skeleton-valued functions of skeletons, and \( V_i \) ranges over sequences of terms, and \( n \) ranges over natural numbers, to be the skeleton-valued function such that

\[
S^*(i.V_i)_0 \text{ is } S(\bot)
\]

and

\[
S^*(i.V_i)_{n+1} \text{ is } (T.S(T \cup V_0 \text{ reflect } S(\bot)))^*(i.V_{i+1})_n.
\]

The intention is to generate a hierarchy from \( S \) with universes \( V_i \). In other words, to generate via \( S \) a hierarchy of equality skeletons with universes \( V_i \), start with \( S(\bot) \), which adds no new types; then to generate the remainder, reflect this type system through a new universe constructor \( V_0 \) and generate, via the enlarged generator, a hierarchy with the remaining universes \( V_{i+1} \). Another way to look at such a hierarchy is as the sequence \( S(\text{spine}_n) \), where \( \text{spine}_0 = \bot \) and \( \text{spine}_{i+1} \) is \( \text{spine}_i \cup V_i \text{ reflect } S(\text{spine}_i) \).

As a convenience, let \( (T.S(T) ; R)^*(i.V_i)_n \), which we may abbreviate as \( (S; R)^*(i.V_i)_n \), be the function \( (T.S(T \cup R))^*(i.V_i)_n \). Then

\[
(S; R)^*(i.V_i)_0 \text{ is } S(R)
\]

and, by associativity of \( \cup \),

\[
(S; R)^*(i.V_i)_{n+1} \text{ is } (S; (V_0 \text{ reflect } S(R)) \cup R)^*(i.V_{i+1})_n.
\]

Of course, the sequence of terms to be added as universes must be chosen so as not to corrupt the type constructors of the generator. The remaining criteria for proper hierarchy generation are expressed in a technical notion of reflective hierarchy. Given a sequence \( T_n \) of equality systems and a sequence \( V_i \) of terms, we may say that \( T_n \) is a reflective hierarchy with universes \( V_i \), when
\forall n. \mathcal{T}_n is a type system \\
& \mathcal{T}_{n+1} \text{ ext } \mathcal{T}_n \\
& \mathcal{T}_{n+1} \text{ ext } \mathcal{V}_n \text{ reflect } \mathcal{T}_n \\
& \mathcal{V}_n \text{ is canonical} \\
& \text{not } \mathcal{V}_n \text{ type } \mathcal{T}_n.

We say \( \mathcal{T}_n \) is a reflective hierarchy when for some sequence \( \mathcal{V}_i \), \( \mathcal{T}_n \) is a reflective hierarchy with universes \( \mathcal{V}_i \).

How are we to use such a hierarchy? The definition of \( \text{Fn} \) given in the previous chapter was, of course, parameterized by type systems, and now we need to make that dependence explicit in our notation. Let \( \text{Fn}_\mathcal{T} \sigma \) indicate the relevant property of sequents for equality skeleton \( \mathcal{T} \). The most direct way of using a hierarchy would be by means of assertions relating individual type systems of the hierarchy to sequents by \( \text{Fn} \). But we can do better.

If \( \mathcal{T} \text{ ext } \mathcal{S} \) and \( \text{Fn}_\mathcal{S} \sigma \) then \( \text{Fn}_\mathcal{T} \sigma \). (This is easily proved by induction on the length of \( \sigma \).) So for a reflective hierarchy \( \mathcal{T}_n \),

\[ \text{Fn}_{\mathcal{T}_{n+1}} \sigma \text{ if } \text{Fn}_{\mathcal{T}_n} \sigma. \]

It seems that normally we shall be concerned only that for some level of the hierarchy (hence everywhere above it) \( \text{Fn} \) holds. We shall assume this to be so in our discussions of reflective hierarchies. We may make our assertions about \( \text{Fn} \) consist in relating reflective hierarchies to sequents by the relation

\[ \exists j. \forall n \geq j. \text{Fn}_{\mathcal{T}_n} \sigma, \]

which we may indicate by \( \text{HFn}(n.\mathcal{T}_n) \sigma \). The essential facts pertaining to the relation between \( \text{HFn} \) and universes are these; supposing that \( \mathcal{T}_n \) is a reflective hierarchy with universes \( \mathcal{V}_i \),\(^2\)

\[ \dagger \text{HFn}(n.\mathcal{T}_n) V_j \in V_m \text{ iff } j < m \]

and

\[ \dagger \text{HFn}(n.\mathcal{T}_n) \Delta \triangleright A = B \in V_{j+m} \text{ if } \text{HFn}(n.\mathcal{T}_n) \Delta \triangleright A = B \in V_j \]

and

\[ \dagger \text{HFn}(n.\mathcal{T}_n) \Delta \triangleright A = B \text{ iff } \exists j. \text{HFn}(n.\mathcal{T}_n) \Delta \triangleright A = B \in V_j. \]

We need never use sequents whose consequent is \( A = B \) if we are willing to engage in the tedium of specifying a level at which \( A \) and \( B \) are type functional.

In order to adapt knowledge about \( \text{Fn} \) to \( \text{HFn} \), it is useful that

\(^2\)See page 40 for the significance of the daggers \( \dagger \).
∀T, σ, k. if (∃j.∀n ≥ j. Fn_{T_n} σ_0 if ∀i < k. Fn_{T_n} σ_{i+1})
then HFn(n.T_n) σ_0 if ∀i < k. HFn(n.T_n) σ_{i+1}.

For example, the facts about Fn expressed at the end of section 3.1, pages 39–43, carry over to the use of HFn(n.T_n) when T_n is a reflective hierarchy.

Another approach to using a reflective hierarchy is to unite it into a single type system. We will not use this formulation; it is discussed only because it is similar to the theories of [HAN] and Nuprl [Constable et al. 86]. Let \textunderscore nT_n, where T_n is a sequence of equality skeletons, be the skeleton such that

A = \textunderscore n T_n B iff \exists n. A =_{T_n} B

and

a = b \in \textunderscore n T_n A iff \exists n. a = b \in _{T_n} A.

If T_n is a reflective hierarchy in n then \textunderscore n T_n is a type system and is an extension of T_n for each n. When T_n is a reflective hierarchy, the relation Fn_{\textunderscore n T_n} σ is equivalent to the relation HFn(n.T_n) σ if every type-valued function belonging to \textunderscore n T_n is a type-valued function of one of the T_n, that is, if

∀Δ. if Fn_{\textunderscore n T_n} Δ then \exists n. Fn_{T_n} Δ.

Note that it is not enough merely that each type inhabit a universe of the hierarchy.

As a last point, the fact that a reflective hierarchy can be united to give a type system provides us with a mechanism for producing transfinite hierarchies, but the \( \omega \)-order hierarchy is sufficient to secure the use of types as objects.

4.2 An Example: Martin-Löf's Types

Here we shall define a type system hierarchy that is the non-type-theoretic analog of the types defined in Martin-Löf's paper [HAN]. A longer discussion of this definition may be found in [Allen 87]. The theory of types presented in [HAN] is open-ended in the sense that additional terms may be introduced and additional type constructors may be defined beyond those explicitly given there. In contrast, the type system to be defined here is closed, having only those terms and type constructors already given. The computation system is that embodied in the procedure given in [HAN] for evaluating terms.\(^3\)

\(^3\)To assure uniqueness of the results of evaluation we shall assume there is a designated variable \( v \) such that the contractum of \((T x, y, z)(\text{sup}(a, b), d)\) is always

\[ d(a, b, (\lambda v)(T x, y, z)(b(v), d) / x, y, z). \]
4.2 An Example: Martin-Löf’s Types

Before proceeding with the definition proper, we shall consider the sort of definition that one might first attempt. One might define typehood and member equality by mutual recursion, where extensional type equality is simply defined by

\[ T = S \text{ iff } T, S \text{ type } \& \forall t s. \; t = s \in T \text{ iff } t = s \in S. \]

The clauses for defining \( \Pi \)-types might be

\[ (\Pi x \in A)B \text{ type if } A \text{ type } \& \forall a a'. \; B[a/x] = B[a'/x] \text{ if } a = a' \in A \]

and

\[ t = t' \in (\Pi x \in A)B \text{ if } (\Pi x \in A)B \text{ type } \]
\[ \& \exists u b u' b'. \; (\lambda u)b \leftarrow t \text{ } \& \text{ } (\lambda u')b' \leftarrow t' \]
\[ \& \forall a a'. \; b[a/u] = b'[a'/u'] \in B[a/x] \]
\[ \text{ if } a = a' \in A. \]

What is unusual about these clauses is that one of the definienda, member equality, occurs negatively on the right hand sides. Thus, the recursive definition does not work by presenting the usual sort of monotonic operator on, in this case, pairs of properties and relations with the aim of designating the least fixed point as the pair of definienda.

What is intended is that we are somehow to understand that “whenever” a type is defined, its membership (equality) is completely defined as well. If we were to work out the other clauses of this definition we would find that whenever member equality is used, only its restriction to “already” defined types is needed, and the right hand sides of the clauses are (strictly) positive in typehood. Let us call the vague principle that licenses such inductive definition half-positive induction. Beeson gives a half-positive inductive definition in [Beeson 82] for his recursive realizability interpretation of [HAN]. He then indicates how to give a standard inductive definition of the model (by means of a device he attributes to Aczel), but this definition depends upon excluded middle. In [Beeson 85] he also mentions the stratification of typehood and member equality using classical set-theoretic ordinals; this is the classical transfinite analog of the \( \omega \)-order stratification that was used to define FIN in section 2.1.

The approach we shall take here is to make precise the nature of the half-positive induction. It is simply this: the half-positive recursive definition of “\( T \text{ type} \)” and “\( t = s \in T \)” is a less than clear definition of the relation

\[ T \text{ is a type with equality } \phi, \]
which can be defined by ordinary induction using an operation on such relations which is strictly positive, hence monotonic, in its argument. Then let \( t = s \in T \) mean that for some \( \phi \), \( T \) is a type with equality \( \phi \) and \( t \phi s \).\(^4\) In the author’s opinion, this form of definition is clearer than the original half-positive form as well as the other suggested classical “translations.”

Since type equality in our example will be extensional, let us restrict our attention to such. We may characterize an extensional type system by a relation \( \tau T \phi \) between terms \((T)\) and two-place relations on terms \((\phi)\) such that

\[
T =_\tau S \text{ iff } \exists \phi. \tau T \phi \& \tau S \phi
\]

and

\[
t = s \in _\tau T \text{ iff } \exists \phi. \tau T \phi \& t \phi s.
\]

We shall use \( \sigma \) and \( \tau \) as variables ranging over such relations, which we shall call possible type systems. Our intention will be to characterize extensional type systems by \( \tau T \phi \) that define partial functions in \( T \), that is, \( \tau \) such that

\[
\forall T \phi \phi'. \phi \text{ is } \phi' \text{ if } \tau T \phi \& \tau T \phi'.
\]

To define our universe hierarchy, we shall define \( \mu(\sigma) \) as the least fixed-point of an operator, \( \text{TyF}(\sigma; \tau) \), which is monotonic in \( \tau \). The types of \( \text{TyF}(\sigma; \tau) \) are those of \( \sigma \) plus those gotten by applying the non-universe type constructors to the types of \( \tau \). We will pass the universes in as base types through \( \sigma \).

We begin the definition by setting out the type formation methods. This is done by defining the relations \( \bar{N}_\tau \), \( \bar{N} \), \( \bar{I} \), \( \bar{+} \), \( \bar{S} \), \( \bar{\Pi} \) and \( \bar{W} \). Each of these is an operator on possible type systems,\(^5\) whose value (a possible type system) has only the types that evaluate to a certain form.

The types \( N_n \) and \( N \) have no constituent types.

\[
\bar{N}_\tau T \phi \text{ iff } \exists n. N_n \rightarrow T \& \forall a b. a \phi b \text{ iff } \exists m < n. m_n \rightarrow a, b.
\]

Define \( N \)-equality by

\[
\text{Neq is the strongest } \phi \text{ such that}
\forall a b. a \phi b \text{ if } 0 \rightarrow a, b \text{ or } \exists a' b'. \text{ suc}(a') \rightarrow a \& \text{ suc}(b') \rightarrow b \& a' \phi b'.
\]

\(^4\)The same sort of definition could be used in the construction of Frege structures in [Aczel 80] (instead of the one given there using classical ordinals) by giving an ordinary inductive definition of

\[ z \text{ is a proposition which is true iff } \Phi. \]

Then, choosing some object \( a \), let “\( z \text{ true} \)” mean that \( z \) is a proposition which is true iff \( a = a \).

\(^5\)We may consider possible type systems themselves to be zero-place operators.
\[ \neg T \phi \text{ iff } N \rightarrow T \land \phi \text{ is Neq.} \]

The rest of the non-universe type constructors have constituent types, and so the type formation operators need, as a parameter, a possible type system from which to get these constituent types. In the definitions of \( \hat{T} \) and \( \hat{\tau} \), \( \alpha \) and \( \beta \) range over two-place relations on terms.

\[ \hat{T}(\tau)T \phi \text{ iff } \exists \, A \, \alpha \, a \, b \, . \, I(A, a, b) \rightarrow T \land \tau \, A \alpha \land \alpha \alpha a \land bab \land \forall \, t \, t'. \, \phi \, t' \text{ iff } t \rightarrow t' \land \alpha \alpha b. \]

\[ \begin{align*}
\hat{\tau}(\tau)T \phi & \text{ iff } \exists \, A \, \alpha \, B \, \beta \, . \, A + B \rightarrow T \land \tau \, A \alpha \land \tau \, B \beta \\
& \land \forall \, t \, t'. \, t \phi \, t' \text{ iff } \exists \, a \, a' \, . \, i(a) \rightarrow t \land i(a') \rightarrow t' \land \alpha \alpha a' \\
& \text{ or } \exists \, b \, b' \, . \, j(b) \rightarrow t \land j(b') \rightarrow t' \land b \beta b'.
\end{align*} \]

Now we proceed with the type constructors having families of constituent types. In the definitions below, \( \alpha \) ranges over two-place relations between terms and \( \gamma \) ranges over three-place relations between terms. The application of \( \gamma \) to terms is indicated by \( t \gamma \).

\[ \text{Fam}(\tau; A; \alpha; x; B; \gamma) \text{ iff } \tau \, A \alpha \land \forall \, a \, a'. \text{ if } a \alpha a' \text{ then } \gamma_a \text{ is } \gamma_{a'} \land \tau \, B[a/x] \gamma_a \land \tau \, B[a'/x] \gamma_{a'}. \]

Note that \( \text{Fam}(\tau; A; \alpha; x; B; \gamma) \) is strictly positive in \( \tau \).

\[ \begin{align*}
\hat{\Sigma}(\tau)T \phi & \text{ iff } \exists \, A \, \alpha \, x \, B \, \gamma. \, (\Sigma x \in A)B \rightarrow T \land \text{Fam}(\tau; A; \alpha; x; B; \gamma) \\
& \land \forall \, t \, t'. \, t \phi \, t' \text{ iff } \exists \, a \, b \, b'. \, (a, b) \rightarrow t \land (a', b') \rightarrow t' \\
& \land a \alpha a' \land b \gamma_{a'b'}. 
\end{align*} \]

\[ \begin{align*}
\hat{\Pi}(\tau)T \phi & \text{ iff } \exists \, A \, \alpha \, x \, B \, \gamma. \, (\Pi x \in A)B \rightarrow T \land \text{Fam}(\tau; A; \alpha; x; B; \gamma) \\
& \land \forall \, t \, t'. \, t \phi \, t' \text{ iff } \exists \, u \, u'. \, (\lambda u)b \rightarrow t \land (\lambda u')b' \rightarrow t' \\
& \land \forall \, a \, a'. \, b[a/u] \gamma_a b'[a'/u'] \text{ if } a \alpha a'. 
\end{align*} \]

Let us define the equality for W types.

\[ \text{Weq}(\alpha; \gamma) \text{ is the strongest } \phi \text{ such that} \]
\[ \forall \, t \, t'. \, t \phi \, t' \text{ if } \exists \, a \, f \, u \, s \, a' \, f' \, u' \, s'. \, \sup(a, f) \rightarrow t \land (\lambda u)s \leftarrow f \\
& \land \sup(a', f') \rightarrow t' \land (\lambda u')s' \leftarrow f' \\
& \land a \alpha a' \land \forall \, b \, b'. \, s[b/u] \phi s'[b'/u'] \text{ if } b \gamma_{a'b'}. \]

\[ \begin{align*}
\hat{\text{W}}(\tau)T \phi & \text{ iff } \exists \, A \, \alpha \, x \, B \, \gamma. \, (Wx \in A)B \rightarrow T \land \text{Fam}(\tau; A; \alpha; x; B; \gamma) \land \phi \text{ is Weq}(\alpha; \gamma). 
\end{align*} \]
In each of these definitions of type formation, the definition of the member
equality of a type depends only on the member equalities of its constituent
types.

We may now define type formation under these constructors plus any
base types.

\[ \text{TyF}(\sigma; \tau) \Theta \phi \iff \sigma \Theta \phi \lor \neg \tau \Theta \phi \lor \neg T \Theta \phi \lor \neg (\tau) \Theta \phi \lor \neg (\tau) \Theta \phi \lor \neg \Sigma(\tau) \Theta \phi \lor \neg \Pi(\tau) \Theta \phi \lor \neg \Phi(\tau) \Theta \phi \]

The relation \( \text{TyF}(\sigma; \tau) \Theta \phi \) is strictly positive, hence monotonic, in \( \tau \). Let us
introduce a convenient notation for closure under \( \text{TyF}(\sigma; \cdot) \).

\[ \text{CTyF}(\sigma; \tau) \Theta \phi \iff \forall \ T \phi. \ \tau \Theta \phi \text{ if } \text{TyF}(\sigma; \tau) \Theta \phi. \]

Now we define \( \mu \).

\[ \mu(\sigma) \text{ is the strongest } \tau \text{ such that } \text{CTyF}(\sigma; \tau). \]

Before discussing the validity of this definition, let us finish up the definition
of the hierarchy. We define our hierarchy using universes \( U_i \) and generator
\( \mu \).

\[ \text{HAN}_n \text{ is } \mu^*(i. U_i)_n. \]

Defining \( \text{spine}_n \) by

\[ \text{spine}_n \Theta \phi \iff \exists m < n. \ U_m \not\equiv T \& \phi \text{ is } =_{\text{HAN}_m}, \]

we may say that \( \text{HAN}_n \) is \( \mu(\text{spine}_n) \).

Returning to the definition of \( \mu \), it is clearly valid set-theoretically (in
a theory with the power set axiom such as ZF or IZF); the terms can be
represented as members of a set \( T \), and for any subset \( \sigma \) of \( T \times \text{Pow}(T \times T) \),
\( \text{TyF}(\sigma; \tau) \) is monotonic in \( \tau \) on the subsets of \( T \times \text{Pow}(T \times T) \), hence

\[ \mu(\sigma) \text{ is } \bigcap \{ \tau \subseteq T \times \text{Pow}(T \times T) \mid \text{TyF}(\sigma; \tau) \subseteq \tau \}. \]

Standard intuitionistic theory of inductive definition directly licenses inductive definitions of

the strongest \( P \) such that \( \forall \overline{x}. \ P \overline{x} \text{ if } \theta(\overline{x}; P) \),

where \( \theta(\overline{x}; P) \) is a relation between individuals and properties of individuals
that is (strictly) positive in \( P \). The definition of \( \mu(\sigma) \) does not quite conform
to this standard since it is not a relation between individuals, but rather,
for each \( \sigma \), \( \mu(\sigma) \) is a relation between individuals and two-place relations
between individuals. Still, the intuitionist might be convinced of the validity
of our definition since \( \text{TyF}(\sigma; \tau) \Theta \phi \) is strictly positive in \( \tau \); it might also
help to note that the equality of each type depends only on the equalities of constituent types.

Notice that the notion of type system, as opposed to possible type system, does not enter into the definition of HANₙ. Although it may already be clear that HANₙ is a reflective hierarchy of type systems, we are in a position to prove it explicitly by using induction on type formation. We shall not carry out that proof here, but the three key lemmas are that μ(σ) is TyF(σ; μ(σ)), that μ(σ) is monotonic in σ, and that if every type of σ evaluates to some Uᵢ, then μ(σ)Tφ defines a partial function in T if σTφ does.

In [HAN] such a hierarchy is not the end result of the type definitions. All the universes are taken as types of a single system whose non-type-theoretic analog would be μ(∪ₙ spineₙ), which we may call HANₙ. For future reference, let us use spineₙ to mean ∪ₙ spineₙ. It turns out that HANₙ is ∪ₙ HANₙ. This would not be so, if, say, there were a term constructor univ(ε), with principle argument ε, such that U₀ = univ(0) and Uᵢ₊₁ = univ(suc(t)); in that case, (Π x ∈ N)univ(x) would be a type of HANₙ but would not be a type of ∪ₙ HANₙ. The argument to be presented here turns on the fact that the Uᵢ are, in a sense, computationally inert. In the course of evaluating a term that has a value, occurrences of Uᵢ are just dragged around or abandoned without any notice taken of which term is being used, and their lineage can always be traced back to occurrences of Uᵢ in the original term. We shall see that if all indices of universes occurring in a type T of HANₙ are less than n, then T is a type of HANₙ.

We shall exploit the fact that there are inert canonical terms that are not types of HANₙ and in which universes do not occur. One such term is I(0, 0, 0). Let t − n be the term gotten from t by replacing each occurrence of Uᵢ₊₁ (for every i) by I(0, 0, 0). Let us say that terms t and s are variants up from n, or t ∼ varₙ s, when t − n is s − n, that is, when t and s differ only in occurrences of I(0, 0, 0) and universes at or above Uₙ. Clearly,

\[ t[s/x] \text{ var}_n t'[s'/x] \text{ if } t \text{ var}_n t' \& \bar{s} \text{ var}_n \bar{s}', \]

where \( \bar{s} \text{ var}_n \bar{s}' \) is the obvious analog of \( t \text{ var}_n t' \). By induction on evaluation,

if \( t \sim s \text{ var}_n s' \) then \( \exists t'. t \sim t' \& t \text{ var}_n t' \).

We shall now characterize a certain kind of immunity to variation up from n which types of HANₙ may have. Define VARₙ Tφ by

\[ \text{VAR}_n T \phi \text{ iff } \forall T'. \text{ HAN}_n T' \phi \text{ if } T' \text{ var}_n T \]
\& \( \forall t s. \text{ if } t \phi s \text{ then } \forall s'. t \phi s' \text{ if } s' \text{ var}_n s. \]

In fact, every type of HANₙ has this immunity (and so, since every term is a variant of itself, HANₙ is VARₙ):
if HAN$_n T\phi$ then VAR$_n T\phi$.

It is enough to show (but we will not) that $\forall n$. CTyF(spine$_n$, VAR$_n$), which may be proved by induction on $n$.\footnote{A helpful observation is that if Fam(VAR$_n$; A; $\alpha$; z; B; $\gamma$) & $A'$ var$_n$ A & $B'$ var$_n$ B then Fam(HAN$_n$; $A'$; $\alpha$; z; $B'$; $\gamma$).} The inductive hypothesis (for numbers less than $n$) is applied only in a certain case of the induction over type formation, namely, when showing that VAR$_n T\phi$ if spine$_n T\phi$. This is also the only point in the proof at which is applied the fact that universes below $U_n$ are left intact under variation up from $n$. If spine$_n T\phi$ then $\exists m < n$. $U_m - T$. Any variation $T'$ of $T$ (up from $n$) must evaluate to a variation of $U_m$. But $U_m$ is the only variant of itself, thus, spine$_n T'\phi$. Since equality in $U_m$ is $\equiv_{\text{HAN}_m}$, the elimination of the inductive hypothesis on $m$ establishes the second conjunct of VAR$_n T\phi$.

It follows from the monotonicity of $\mu(\sigma)$ in $\sigma$ that $\cup_n$ HAN$_n$ is as strong as HAN$_\omega$. Thus, since $T$ is $T - n$ for some $n$, to show that HAN$_\omega$ is $\cup_n$ HAN$_n$ it is enough to show that

if HAN$_\omega T\phi$ & $T$ is $T - n$ then HAN$_n T\phi$,

for which in turn it enough to show that CTyF(spine$_\omega$, BD$_n$), where BD$_n T\phi$ means that HAN$_n T\phi$ if $T$ is $T - n$. The only interesting aspect of the proof is a lemma,

if Fam(BD$_n$; A; $\alpha$; z; B; $\gamma$) & A is $A - n$ & B is $B - n$
then Fam(HAN$_n$; A; $\alpha$; z; B; $\gamma$).

**Proof:**

arb $n$, $A$, $\alpha$, $z$, $B$, $\gamma$ s.t. the antecedent holds.

HAN$_n A\alpha$ since BD$_n A\alpha$ & A is $A - n$.

arb $a$, $a'$ s.t. $aa'\alpha$.

$\gamma_a$ is $\gamma_{a'}$.

**enough to show** HAN$_n B[a/x]$ $\gamma_a$ & HAN$_n B[a'/x]$ $\gamma_{a'}$.

$a \alpha (a' - n)$ & $(a - n) \alpha a'$ since HAN$_n$ is VAR$_n$.

$\gamma_a$ is $\gamma_{a - n}$ & $\gamma_{a'}$ is $\gamma_{a' - n}$ since $a \alpha (a - n)$ & $a' \alpha (a' - n)$.

**enough to show** HAN$_n B[a/x]$ $\gamma_{a - n}$ & HAN$_n B[a'/x]$ $\gamma_{a' - n}$.

HAN$_n B[a - n/x]$ $\gamma_{a - n}$ & HAN$_n B[a' - n/x]$ $\gamma_{a' - n}$ by the first assumption, since $(a - n) \alpha (a' - n)$.

**QED** since HAN$_n$ is VAR$_n$. 
Finally, we shall see that $\text{Fn}_{\text{HAN}_\omega}$ is $\text{HF}_{\text{n} \cdot \text{HAN}_n}$. As was indicated in the previous section, it is enough that, for all $\Delta$, if $\text{Fn}_{\text{HAN}_\omega} \Delta$ then $\exists n \cdot \text{Fn}_{\text{HAN}_n} \Delta$. And this reduces to showing that

$$\text{Fn}_{\text{HAN}_n} \Delta \text{ if } \text{Fn}_{\text{HAN}_\omega} \Delta - n,$$

(where $\Delta - n$ is the obvious analog of $t - n$) since for every $\Delta$ there is an $n$ such that $\Delta$ is $\Delta - n$.

**Proof** by induction on the length of $\Delta$:

We shall abbreviate $\text{HAN}_n$ and $\text{HAN}_\omega$

by $n$ and $\omega$ in subscripts to $\text{Fn}_=, \text{and } \in$.

**arb** $n$:

The base case is trivial.

**arb** $\Delta$ s.t. the inductive hypothesis holds for $|\Delta|$.

**arb** $A, B, x$ s.t. $\text{Fn}_\omega (A, B : x \Delta) - n$.

$(A - n) =_\omega (B - n)$.

$(A - n) =_n (B - n)$ since $\text{HAN}_\omega$ is a subrelation of $\text{BD}_n$.

$A =_n B$ since $\text{HAN}_n$ is $\text{VAR}_n$.

**arb** $a, b$ s.t. $a = b \in_n A$.

**show** $\text{Fn}_n \Delta [a; b/x]$.

**enough** by the inductive hypothesis

**to show** $\text{Fn}_\omega (\Delta - n)[a - n; b - n/x]$.

$a - n = b - n \in_n A - n$ since $\text{HAN}_n$ is $\text{VAR}_n$.

$a - n = b - n \in_\omega A - n$.

**QED**.

### 4.3 Universe Polymorphism

Normally, we may expect to design reflective hierarchies whose constituent type systems are very similar. The principal similarity is that there will be certain type constructors under which every level of the hierarchy is closed. Let us call such type constructors uniform. Beyond this, there is the regularity of universe construction. Often our assertions relating several universes depend not on the particular universes mentioned, but rather, only on certain simple arithmetical relations between the indices of those universes. For example, we know that for any $i$ and $j$,

$$\text{HF}_{\text{n} \cdot \text{HAN}_n} \cup_i : X \cup_j : Y \cup X + Y \cup \text{U}_{\text{max}(i, j)}.$$
But this fact is not expressible by merely predicing $\text{HF}_n(n.\text{HAN}_n)$ of some sequent. In a proof system based on assertions of the form "$\text{HF}_n(n.\text{T}_n) \sigma$", or similarly restrictive forms, this is quite a nuisance, since different instances of rather simple abstractions from universe indices must be proved independently. One approach to this problem would be to schematize proofs by variables ranging over universe levels (numbers) and to require that certain relations between these schema variables, determined by the proof, be met in order to instantiate a schema. One might also permit the use of simple expressions for indices, such as $\text{max}(i, j)$, to reduce the number of index variables needed.

We shall not pursue this purely proof-theoretic approach here. Instead, we shall directly seek a semantic solution by slightly generalizing properties of universe hierarchies (term sequences). Our method will be to introduce new "expressions" to be used, instead of numbers, as universe indices, and then to show how to interpret propositions that use these new expressions as propositions about ordinary sequences of universes. More precisely, we will define a class of new index expressions, which we shall call level expressions, and we will define a relation $\text{UPoly}(i.V_i; U.\Psi(\alpha.\text{U}_\alpha))$ between term sequences $(V_i)$ and properties $(\Psi)$ of families of terms $(\text{U}_\alpha)$ over level expressions $(\alpha)$. In practice, it may be convenient to extend the class of terms to include level expressions. Thus, when our principal assertions are about finitely many terms, e.g., the terms of a sequent, our universe polymorphic assertions may be expected to take the form

$$\text{UPoly}(i.V_i; U.\Phi(m.\text{F}(\alpha.\text{U}_\alpha; t_m))),$$

where

- the $t_m$ are from a class of extended terms resulting from adding the new level expressions as terms, and

- $\text{F}(\alpha.\text{U}_\alpha; s)$ is the result of replacing each (outermost) level expression $\beta$ in $s$ by $U_\beta$, and

- $\Phi(m.\tau_m)$ depends upon only finitely many terms of $\tau_m$.

The details of the level expressions and of the definition of $\text{UPoly}$ that are to be proposed here are based on some guesses about what sorts of schemas (over universe levels) are desirable. First, one should be able to specify independently varying choices of arbitrary levels of the hierarchy, that is, one should have variables that range over all universe levels (numbers). In interpreting the rest of the level expressions these variables are held constant. In choosing our other forms of level expression we shall strive to make expressions stand for the lowest levels possible (given assignments to the level variables) since the universes of a reflective hierarchy are cumulative. Next,
we should have an expression for the successor of a (n ambiguously specified) level, in order that, given a universe, we can specify the least universe of which it is a member. Finally, given several (ambiguously specified) levels, one should be able to specify the maximum of them. This is because, when a type is constructed by means of a uniform type constructor and when various levels are specified for its constituent types, we may specify the level of the result type to be the maximum of the levels specified for the constituents. We shall design the level expressions to meet these three demands. Notice that no provision has been made for specifying level constants. At the end of this section, it will be argued that references to particular universes should be systematically avoided.

There are three distinct forms of level expression. Each expression is

- a level variable, or
- a nonempty list \([\alpha_0 \ldots \alpha_k]\), where \(\alpha_0 \ldots \alpha_k\) are level expressions, or
- a pair \(\alpha[n]\), where \(\alpha\) is a level expression and \(n\) is a number.

These conditions should be read as clauses of an inductive definition parameterized by the choice of level variables. We shall assume that identity between level variables is decidable and that there is at least one level variable. Now we define the value of each expression with respect to an assignment to level variables. Define \((\alpha)_f\) to be a number-valued function such that

- \((\alpha)_f\) is \(f(\alpha)\) if \(\alpha\) is a level variable,
- \(((\alpha_0 \ldots \alpha_k))_f\) is \(\max_{i \leq k}(\alpha_i)_f\); and
- \((\alpha[n])_f\) is \((\alpha)_f + n\).

Only the values of \(f\) on level variables matter.

We can now define \(\text{UPoly}\) by

\[
\text{UPoly}(i.V_i; U.\Psi(\alpha.U_{\alpha})) \iff \forall f. \Psi(\alpha.V_{(\alpha)_f}).
\]

The principal means of exploiting universe polymorphic assertions is in the application of the following fact.

If \(\text{UPoly}(i.V_i; U.\Psi(\alpha.U_{\alpha})) \land \gamma_1 \ldots \gamma_k\) are level variables
then \(\text{UPoly}(i.V_i; U.\Psi(\alpha.U_{\alpha[\beta/\gamma]} \ldots \beta_k/\gamma_1 \ldots \gamma_k}))\),

where \(\alpha[\beta/\gamma]\) is the obvious analog of \(a[\beta/\gamma]\). To show this, it is enough to examine the case in which \(\gamma_1 \ldots \gamma_k\) are distinct.\(^7\)

\(^7\)This proof is a correction to that in the original version of this report.
Proof:

\( \text{arb } V, \Psi \text{ s.t. } \text{UPoly}(i.V_1; U.\Psi(\alpha.U_\alpha)) \).

\( \text{arb } k, \gamma_1 \ldots \gamma_k \text{ s.t. } \gamma_1 \ldots \gamma_k \) are distinct level variables.

\( \text{arb } \beta_1 \ldots \beta_k, f : \)

show \( \Psi(\alpha.V(\alpha/\beta_1 \ldots \beta_k/\gamma_1 \ldots \gamma_k))f \).

let \( g \) be the function such that \( \forall i < k. \ g(\gamma_{i+1}) = (\beta_{i+1})f \) and \( \forall \alpha. \ g(\alpha) = f(\alpha) \) if \( \alpha \) is not among \( \gamma_1 \ldots \gamma_k \).

\( \forall \alpha. \ (\alpha)_g = (\alpha/\beta_1 \ldots \beta_k/\gamma_1 \ldots \gamma_k)_f \).

QED by applying the first hypothesis to \( g \).

Note that one trivial but practically valuable fact is that level variables may be uniformly renamed. This facilitates the combination of polymorphic propositions when the independence of level variables is to be preserved. Before we proceed with our study of UPoly, it should be noticed that even if we were not to use UPoly, the use of level expressions would still be a suitable means for schematizing proofs by universes.

The use of universe polymorphic assertions is intended to uniformly supplant the use of non-polymorphic assertions. Non-polymorphic assertions of the form \( \Phi(m.V_g(m)) \), where \( m \) may be thought of as indexing the occurrences of \( V_\gamma \) in the assertion, are supplanted by possibly more general assertions of the form \( \text{UPoly}(i.V_i; U.\Phi(m.U_{\gamma(m)})) \). We shall set down a number of facts that form the basis for adapting knowledge about the non-polymorphic forms to the polymorphic forms.

When the indices of universes are of no special concern in a non-polymorphic proposition, for example, when we are using the facts about HF\( n \) that were adapted from those of chapter 3 about \( F_n \), then our aim is merely to introduce the new level expressions.

If \( \forall g. \ \Phi(m.V_g(m)) \) then \( \text{UPoly}(i.V_i; U.\Phi(m.U_{\gamma(m)})) \).

(Given any \( f \), apply the hypothesis to \( (\gamma(m))_f \) (for \( g(m) \)).) Some non-polymorphic propositions may depend upon some relation between universe indices; for example, the fact given on page 47 relating HF\( n \) to cumulativity depends on one universe being no higher than another. In such cases, the condition on numeric universe levels must be converted to a condition on new level expressions. For any relation \( iRj \) between numbers,

if \( \forall g, j, k. \ \Phi(m.V_g(m); V_j; V_k) \) if \( jRk \)
then \( \forall \gamma, \alpha, \beta. \ \text{UPoly}(i.V_i; U.\Phi(m.U_{\gamma(m)}; U_\alpha; U_\beta)) \) if \( \forall f. \ (\alpha)_f R(\beta)_f \).

The relation \( \forall f. \ (\alpha)_f R(\beta)_f \) is recursively decidable when \( jRk \) is either \( j < k \), or \( j \leq k \), or identity.
4.3 Universe Polymorphism

In order to adapt inference rules about non-polymorphic assertions, we need, in addition to the means just indicated for replacing numeric-indices by the new expressions, the following:

\[ \text{UPoly}(i; V; \Psi) \text{ if } \text{UPoly}(i; V; \Psi') \land \text{UPoly}(i; V; U; \Psi \, (\alpha.U_\alpha) \text{ if } \Psi' \, (\alpha.U_\alpha) \). \]

It is possible to use UPoly to express functionality over all types at all levels of a hierarchy. This is done by specifying the type of the argument to be an ambiguous universe that can be made arbitrarily large independently of other specified universe levels. We shall use HFn and exploit the fact that

\[ 1 \text{ HFn}(n; T_n) \Delta \triangleright A = B \text{ iff } 2 \exists j. \text{ HFn}(n; T_n) \Delta \triangleright A = B \in V_j. \]

Let us say that a level variable \( \gamma \) is reassignable in \( F(\alpha.U_\alpha) \) (via \( \alpha, U_\alpha \))\(^8\) when

\[ \forall U, U'. \text{ if } \forall \alpha. U_\alpha \text{ is } U'_\alpha \text{ if } \gamma \text{ does not occur in } \alpha \text{ then } F(\alpha.U_\alpha) \text{ is } F(\alpha.U'_\alpha). \]

Then, abbreviating \( \Delta(\alpha.U_\alpha) \) to \( \Delta_U \), and so forth,

\[ 2 \text{ if } T_n \text{ is a reflective hierarchy with universes } V_i \& \gamma \text{ is reassignable (via } \alpha, U_\alpha \text{) in } \Delta_U \text{ and } \sigma_U \& \gamma \text{ occurs in } \beta \& \forall U. \text{ free } \sigma_U[AU; BU/X] \text{ if } \forall \alpha. U_\alpha \text{ is closed} \& \text{UPoly}(i; V_i; U. \text{ HFn}(n; T_n) \Delta_U \ U_\beta : X \triangleright \sigma_U) \& \text{UPoly}(i; V_i; U. \text{ HFn}(n; T_n) \Delta_U \triangleright AU = BU) \text{ then } \text{UPoly}(i; V_i; U. \text{ HFn}(n; T_n) \Delta_U \triangleright \sigma_U[AU; BU/X]. \]

**Proof:**

arb \( V, \Delta, A, B, \alpha, \beta, X, \sigma \text{ s.t. the antecedent holds.} \)

arb \( f \):

let \( U_\alpha \) be \( V(\alpha)_f \).

show \( \text{HFn}(n; T_n) \Delta_U \triangleright \sigma_U[AU; BU/X]. \)

\( \text{HFn}(n; T_n) \Delta_U \triangleright AU = BU. \)

let \( j \) be such that \( \text{HFn}(n; T_n) \Delta_U \triangleright AU = BU \in V_j. \)

let \( g \) be such that \( g(\gamma) \) is \( j \& \forall \alpha. g(\alpha) \) is \( f(\alpha) \) if \( \alpha \) isn’t \( \gamma. \)

let \( U'_\alpha \) be \( V(\alpha)_g \).

\( \text{HFn}(n; T_n) \Delta_U \triangleright AU = BU \in U'_\beta \) since \( j \leq (\beta)_g. \)

---

\(^8\)This via clause merely binds \( U. \)
HF\n(n.T_n) \Delta_U, U'_\beta : X \triangleright \sigma_U.

HF\n(n.T_n) \Delta_U \triangleright U'_\beta : X \sigma_U \text{ by the reassignability of } \gamma \text{ in } \Delta \text{ and } \sigma.

\textbf{QED} by composition of sequents with functional expressions.

We can generalize this a bit to handle functionality over type-valued functions. Let \( \Delta \rightarrow T \) and \((\lambda \Delta)t\) be term-valued functions such that (for any \( \Delta, \Delta', s, t \text{ and } T \))

\[ \text{\( \uparrow \text{ HF} \n (n.T_n) \Delta \triangleright (\lambda \Delta')t = (\lambda \Delta')s \in \Delta' \rightarrow T \)} \]

if \( \text{HF}(n.T_n) \Delta \Delta' \triangleright t = s \in T. \)

Such term-valued functions are easily constructed by iteration of \((\Pi x \in A)B \) and \((\lambda x)b\).

\[ \text{\( \downarrow \text{ If } T_n \text{ is a reflective hierarchy with universes } V_i \)} \]

\& \( \gamma \text{ is reassignable (via } \alpha, U_\alpha \text{) in } \Delta_U, \Delta'_U \text{ and } \sigma_U \)

\& \( \gamma \text{ occurs in } \beta \)

\& \( \forall U. \text{ free } \sigma_U[A_U; B_U/X] \text{ if } \forall \alpha. U_\alpha \text{ is closed} \)

\& \( \text{UPoly(i.V_i; U. HF}(n.T_n) \Delta_U \Delta'_U \rightarrow U_\beta : X \triangleright \sigma_U) \)

\& \( \text{UPoly(i.V_i; U. HF}(n.T_n) \Delta_U \Delta'_U \triangleright A_U = B_U) \)

then \( \text{UPoly(i.V_i; U. HF}(n.T_n) \Delta_U \triangleright \sigma_U[(\lambda \Delta'_U)A_U; (\lambda \Delta_U)B_U/X)] \).

The proof is achieved by an obvious modification of the one just given.

Finally, let us consider the possibility of using proof systems built solely on universe polymorphic assertions that make no reference to individual universes. The danger, of course, is that the polymorphic propositions will be stronger than, or at least we may not know them to be as weak as, the propositions about particular universes, thus we would lose the ability to express valuable propositions. It will be suggested that these weaker propositions are not actually valuable and should be shunned.

The weakest polymorphic propositions are those that use only one level variable. If \( \gamma \) is the only level variable occurring in \( \alpha \), then \((\alpha)f\) is

\[ (\gamma[(\alpha)K_0])f, \]

where \( K_0 \) is the constant zero-valued function. Thus, the weakest polymorphic propositions may be put into the form

\[ \text{UPoly(i.V_i; U. } \Phi(i.U_{\gamma[i]})), \]

where \( \gamma \) is a level variable. This is equivalent to \( \forall f. \Phi(i.V_{\gamma[i]}f) \), which is equivalent to

\[ \forall j. \Phi(i.V_{j+1}). \]
Therefore, the argument for exclusive use of universe polymorphic propositions, as the basis of a proof system, reduces to arguing for the use of \( \forall j. \Phi(i.V_{j-1}) \), which we may call universe relative, instead of \( \Phi(i.V_i) \).

As an example, suppose we were considering whether to "go polymorphic" with a proof system based on assertions of the form

\[
\text{HFn}(n.T_n) \sigma(i.V_i).
\]

The new form of assertion would be

\[
\text{UPoly}(i.V_i; U.\text{HFn}(n.T_n) \sigma(\alpha.U_\alpha)),
\]

where \( \sigma \) gets its universes only through its argument. If it should turn out that uniformly incrementing universes throughout a sequent preserves \( \text{HFn}(n.T_n) \), then we would lose nothing at all in moving to polymorphic forms since our assertions would already be universe relative.\(^9\) But there is reason to accept, and even require, universe relative assertions even when they are stronger than those they replace. It concerns the anticipation of adding new types to the type system hierarchy (resulting in a new hierarchy, of course), a matter considered in more detail in chapter 6.

As users of a given type system hierarchy, we are likely to expect that new type constructors will be added in the future, the simplest case being the addition of base types to all levels of the hierarchy. Suppose \( T_n \) is a reflective hierarchy (with universes \( V_i \)), and \( T'_n \) is a reflective hierarchy (with universes \( V'_i \)) that is built from the same (non-universe) type constructors as \( T_n \), but also from new base types \( B_0 \ldots B_{j-1} \). We would prefer that the assertions we prove about \( T_n \) carry over to \( T'_n \). One possible source of difficulty might be the difference between \( V_i \) and \( V'_i \). But, since the universes are introduced merely for the purpose of obtaining a reflective hierarchy, we may feel free to abstract from the particular sequence of terms used for the universes, perhaps by treating the occurrences of universes as variables ranging over term sequences. What we want to do is restrict our assertions about \( T_n \) in such a way that they may be reinterpreted as assertions about \( T'_n \) (preserving truth). Thus, we would be willing to restrict our assertions to propositions of the form

\[
\text{if } B_0 \ldots B_{j-1} \text{ are reasonable base types to add to } T_n, \text{ and } V'_i \\
\text{is a sequence of terms suitable for use as universes of } T'_n, \text{ then} \\
\exists k. \forall n \geq k. \Phi(i.V'_i; T'_n).^{10}
\]

\(^9\)It seems plausible that the hierarchy \( \text{HAN}_n \) (with universes \( U_i \)) has this property, but I have not proved it.

\(^{10}\)We shall assume that, normally, our assertions about hierarchies may be put in the form \( \exists k. \forall n \geq k \ldots T_n \ldots \). This is the form of applications of \( \text{HFn} \) and the other relation, \( \text{HPwFn} \), between hierarchies and sequents which is to be studied in a later chapter.
But since it is difficult to be perfectly precise about what new base types are reasonable we shall have to settle for less discipline.

For any number $j$, some reasonable choices of base types $B_0 \ldots B_{j-1}$ are the universes $V_0 \ldots V_{j-1}$ of the hierarchy $\mathbb{T}_n$, and, given these base types, one reasonable choice for the term sequence $V'_i$ is simply $V_{j+i}$, making the hierarchy $\mathbb{T}'_n$ identical with $\mathbb{T}_{j-n}$. Thus, we should be willing, even eager, to restrict our assertions to propositions that may be put in the form

$$\forall j. \exists k. \forall n \geq k. \Phi(i.V_{j+i}; \mathbb{T}_{j-n}),$$

and this is equivalent to the universe relative form

$$\forall j. \exists k. \forall n \geq k. \Phi(i.V_{j+i}; \mathbb{T}_n).$$

So, we should also be happy to restrict our assertions to universe polymorphic ones that make no reference to particular universes.

Universe relativity will be treated with more precision in chapter 6.
Chapter 5

Respect for Relations

Suppose that \( t \) and \( s \) have the same value if either has a value. There is no need to analyze \( T \) in particular in order to make the inference from \( t \in T \) to \( t = s \in T \). There are many useful instances of this form of inference, for example, when \( t \) is a redex with contractum \( s \). The aim of this chapter is to generalize from this paradigm.

Rather than considering only sameness of value, we shall consider generally the relations, which we shall call respected relations, that would justify this and similar forms of inference. We shall also establish the connection between functionality and respected relations that allows us to generalize this form of inference to judgements using assumptions. In later sections we will discuss particular relations that are useful if respected. Special attention will be given to the replacement of closed subterms by computationally equivalent terms.

As evidence of the applicability of this work, we may mention that the direct computation rules of Nuprl [Constable et al. 86] are based on this work, and they have proved valuable.

5.1 Respected Relations and Functionality

Let \( T \simeq \text{type } S \) be the relation on terms

\[
\forall A. T = A \text{ iff } S = A,
\]

and let \( t \simeq \text{mem } s \) be the relation

\[
\forall A, a. t = a \in A \text{ iff } s = a \in A.
\]

These are equivalence relations. The relations respected, we shall say, by type equality are the subrelations of \( \simeq \text{type } \), while those respected by member equality are the subrelations of \( \simeq \text{mem } \). We shall look at some relations
which are practically useful if respected. But first we will establish some useful properties of respect. The dagger $\dagger$ and double dagger $\ddagger$ prefixes are used as in section 3 and are not yet significant.\footnote{See the footnote on page 40 for explanation of the daggers.}

The similarity between $\simeq$ type and $\simeq$ mem makes it convenient to have an ambiguous reference to them both. We shall use the relational variable $\simeq$ opt and restrict it by implicitly assuming that $\simeq$ opt is $\simeq$ type or is $\simeq$ mem, and we shall use the property-of-relations variable "R is respected" implicitly assuming that to be respected is to be a subrelation of $\simeq$ opt.

To establish the connection between Fn and respected relations we will need the operation $R^o$ on relations such that

$$t \ R^o \ s \ \text{iff} \ \forall \overline{e}, \overline{x}. \ t[\overline{e}/\overline{x}] \ R \ s[\overline{e}/\overline{x}] \ \text{if} \ \overline{e}, \ t[\overline{e}/\overline{x}], \ s[\overline{e}/\overline{x}] \ \text{are closed.}$$

It follows that

$$t \ R^o \ s \ \text{iff} \ \forall \overline{e}, \overline{x}. \ t[\overline{e}/\overline{x}] \ R^o \ s[\overline{e}/\overline{x}] \ \text{if} \ \overline{e} \ \text{closed.}$$

Notice that $R^o$ depends only upon the relation between closed terms related by $R$, and that the operation $R^o$ is monotonic in $R$. The connection between member equality and respect is that

$$\dagger \ Fn \ \Delta \triangleright t = s \in T = S \ \text{if} \ \Fn \ \Delta \triangleright t' = s \in T = S \ \& \ t \simeq \ \text{mem}^o \ t' \ \& \ \text{the free variables of} \ t \ \text{are among} \ \Delta.$$

**Proof** by induction on the number of declarations:

**arb** $R$:

**base case**:

**arb** $t, s, t', T, S$:

**assume** $t' = s \in T = S$.

**assume** $t \simeq \ \text{mem}^o \ t'$.

**assume** $t$ closed.

**show** $t = s \in T = S$.

$t \simeq \ \text{mem} t'$ since $t$ and $t'$ are closed.

qed.

**arb** $\Delta$:

**assume** induction hypothesis for declaration lists as short as $\Delta$.

**arb** $A, B, x, t, s, t', T, S$:
\(\text{assume } \text{Fn } A,B : x \Delta \triangleright t' = s \in T = S.\)

\(\text{assume } t \simeq \text{mem}^0 t'.\)

\(\text{assume } \text{the free variables of } t \text{ are among } x, \tilde{\Delta}.\)

\(A = B.\)

\(\text{arb } a,b \text{ s.t. } a = b \in A.\)

\(\text{enough to show } \text{Fn } (\Delta \triangleright t = s \in T = S)[a;b/x].\)

\(\text{Fn } (\Delta \triangleright t' = s \in T = S)[a;b/x].\)

\(t[a, \tilde{\Delta}/x, \tilde{\Delta}] \simeq \text{mem}^0 t'[a, \tilde{\Delta}/x, \tilde{\Delta}].\)

the free variables of \(t[a, \tilde{\Delta}/x, \tilde{\Delta}]\) are among \(x, \tilde{\Delta}.\)

\text{QED by induction hypothesis.}

Hence,

\[\uparrow \text{Fn } \Delta \triangleright t = s \in T \text{ if } \text{Fn } \Delta \triangleright s \in T \]
\[\& \ t \simeq \text{mem}^0 s \]
\[\& \ \text{the free variables of } t \text{ are among } \tilde{\Delta}.\]

An analogous result holds for respect by type equality, by a similar proof, so,

\[\uparrow \text{Fn } \Delta \triangleright T = S \text{ if } \text{Fn } \Delta \triangleright S \text{ type} \]
\[\& \ T \simeq \text{type}^0 S \]
\[\& \ \text{the free variables of } T \text{ are among } \tilde{\Delta}.\]

These results, in combination with the facts of section 3.1 concerning replacement of type-valued functions by equal functions, give us that

\[\uparrow \text{if } \text{Fn } \Delta \triangleright t = s \in T = S \]
\[\& \ T \simeq \text{type}^0 T' \& S \simeq \text{type}^0 S' \]
\[\& \ t \simeq \text{mem}^0 t' \& s \simeq \text{mem}^0 s' \]
\[\& \ \text{the free variables of } t', s', T', S' \text{ are among } \tilde{\Delta} \]
then \(\text{Fn } \Delta \triangleright t' = s' \in T' = S'\)

and

\[\uparrow \text{if } \text{Fn } \Delta \triangleright A : x \sigma \]
\[\& \ A \simeq \text{type}^0 A' \]
\[\& \ \text{the free variables of } A, A' \text{ are among } \tilde{\Delta} \]
then \(\text{Fn } \Delta \triangleright A' : x \sigma.\)
Here are a few facts that can be helpful in the selection and use of respected relations. If relations $t \mathbin{R} s$ and $t \mathbin{S} s$ are respected then so is their composition, $\exists t \mathbin{R} r \mathbin{S} s$, which we may write as $t \mathbin{R} \circ \mathbin{S} s$. Let $R^*$ be the strongest equivalence relation of which $R$ is a subrelation. If $R$ is a subrelation of $S$ then $R^*$ is a subrelation of $S^*$, hence, $R^*$ is respected if $R$ is. The relation $R^{*^0}$ is $R^{*^0^*}$. And so, $\simeq \circ R^{*^0}$ is $\simeq \circ R^*$.

A more refined notion of respect by member equality may be easily obtained if needed. Instead of $t \simeq \text{mem } s$ we would use $t \simeq [A] s$, meaning

$$\forall a. \, t = a \in A \text{ iff } s = a \in A.$$ 

And we would use an operation $R^\circ(t, s, A)$ on three-place relations similar to the one we defined on two-place relations, viz.,

$$R^\circ(t, s, A) \text{ iff } \forall \bar{e}, \bar{y}. \, R(t[\bar{e}/\bar{y}], s[\bar{e}/\bar{y}], A[\bar{e}/\bar{y}])$$

if $\bar{e}, t[\bar{e}/\bar{y}], s[\bar{e}/\bar{y}], A[\bar{e}/\bar{y}]$ closed.

I do not know whether this refinement is useful, and so in the presentation of respect I have avoided the extra complexity.

When using respected relations in conjunction with a reflective hierarchy, the expectation is that one will use relations that are respected at every level of the hierarchy. Universe polymorphic assertions about relations respected throughout a hierarchy $T_n$ with universes $V_i$ will have the form

$$\text{UPoly}(i, V_i; U, t(\alpha.U_\alpha) R^\circ s(\alpha.U_\alpha),)$$

where $R$ is a subrelation of $\simeq \circ \text{opt}_{T_n}$ for every $n$.

### 5.2 Change of Bound Variables

It would seem unusual for someone to object to respect for $\alpha$-conversion, i.e., change of bound variables (without incurring capture). Indeed, one might be inclined to build it into the very definition of type assignment system; however, I have not done so because it would contribute very little to the nature of type systems, despite its practical convenience. In combination with the fact that $\text{Fn } \sigma$ is preserved under a change of variables declared in $\sigma$, respect for $\alpha$-conversion ensures that $\text{Fn } \sigma$ is preserved under change of any bound variables of $\sigma$. It is very easy to arrange respect for $\alpha$-conversion, and it is a great practical advantage since one is bound to design inference

---

2This condition on $A$ is included to justify the dagger $\dagger$ prefix.

3By $t \mathbin{R} r \mathbin{S} s$ is meant that $t \mathbin{R} r$ and $r \mathbin{S} s$.

4One direction is immediate. The other follows from the fact that $S^{*^0}$ is always a subrelation of $S^{*^0}$ and $R^{*^0}$ is $R^*$. 
rules that run the danger of free variable capture. Of course, to escape capture one must have sufficiently many variables. Let us say that variables are plentiful when
\[ \forall \bar{e}. \exists x: x \text{ is not among } \bar{e}. \]
Normally, variables are plentiful.

Let \( t\alpha s \) mean that \( t \) and \( s \) are \( \alpha \)-convertible. When considering respect for relations in a system that respects \( \alpha \)-conversion, the relation \( \alpha \circ R \circ \alpha \), which we may write as \( R^\alpha \), is valuable because it permits us to sidestep any difficulties with choice of bound variables that might arise in the use of \( R \). Of course, \( R^\alpha \) is respected if \( R \) and \( \alpha \)-conversion are.

It may be helpful to note that \( R^{\alpha^*} \) is \( R^{\alpha \ast \alpha} \) and \( R^{\alpha^*} \) is \( R^{\alpha \ast \alpha} \), and that if \( \alpha \)-conversion is respected then \( \simeq \text{opt} \) is \( \simeq \text{opt}^\alpha \).

It can be shown that type and member equality of \( \text{HAN}_n \) respect \( \alpha \)-conversion. Prove by induction on — that
\[ \text{if } t \simeq s \alpha s' \text{ then } \exists t'. t\alpha t' \simeq s', \]
and then show by induction on \( n \) that \( \text{CTyF}(\text{spine}_n; \alpha_n) \), where
\[ \alpha_n T \phi \text{ iff } \forall T'. \ \text{HAN}_n T' \phi \text{ if } T' \alpha T \]
\[ \& \forall s, t. \text{ if } s \alpha t \text{ then } \forall t'. s \alpha t' \text{ if } t' \alpha t. \]

5.3 Partial Identity of Value

Let \( t \simeq s \) be the relation
\[ t, s \text{ are closed } \& \forall a. a \rightarrow t \text{ iff } a \rightarrow s. \]
This relation is respected. Rules about evaluation that are independent of type may be expressed using \( \simeq^o \). For example, if, as in \( \text{HAN}, \text{Constable et al. 86} \), evaluation is defined in terms of principal argument places and redex/contractum pairs, we get \( r \simeq^o c \) if every closed instantiation of \( r \), say \( r[\bar{a}/\bar{e}] \), is a redex whose contractum is \( c[\bar{a}/\bar{e}] \). And if \( x \) occurs in \( t \) only as a principle argument place and \( a \simeq^o b \) then \( t[a/x] \simeq^o t[b/x] \) if \( \text{free } t[a/x] \) and \( \text{free } t[b/x] \).

5.4 Closed Subterm Variation

Now we shall look at a means for more liberal use of \( \simeq^o \). First, let me mention a relation with which I have had no success. The replacement of any subterms, open or closed, by \( \simeq^o \)-equivalent terms, would be very handy if it were respected. However, with my rather simple proof methods I have not
been able to confirm respect for this relation by either Nuprl or by HAN$_n$. Instead, we shall study the replacement of closed subterms by $\simeq$-equivalent terms.

Terms $t$ and $s$ are closed subterm variants, or $\text{csv } s$, when $t$ can be gotten from $s$ by replacing zero or more (closed) subterms of $s$ by $\simeq$-equivalent (closed) terms. This super-relation of $\simeq$ is reflexive and symmetric, but not necessarily transitive. If variables are plentiful then $t \text{ csv } s$ iff

$$\exists r, n, a_1 \ldots a_n, b_1 \ldots b_n, x_1 \ldots x_n. \forall i < n. a_{i+1} \simeq b_{i+1}$$

$$& t = r[a_1 \ldots a_n/x_1 \ldots x_n]$$

$$& s = r[b_1 \ldots b_n/x_1 \ldots x_n].$$

Respect for closed subterm variation reduces the problem of whether a term $t$ is functional in $x$ (over type $A$) to whether $t[a/x]$ and $t[b/x]$ are equal for all equal (in $A$) canonical terms $a$ and $b$. For example, respect for closed subterm variation by a type system would justify the following rule for cartesian product elimination.

$$\text{Fn } \Delta \triangleright t[s/z] \in T[s/z]$$

if $\text{free } t[s/z] \& \text{free } T[s/z]$

$$& \text{Fn } \Delta \triangleright s \in A \times B$$

$$& \text{Fn } \Delta A : x B : y \triangleright t[(x, y)/z] \in T[(x, y)/z]$$

$$& \text{free } t[(x, y)/z] \& \text{free } T[(x, y)/z]$$

$$& x, y \text{ are not free in } t, T \& x \text{ is not } y.$$

There are several such elimination rules given in [Petersson 82] which omit certain premises implicit in the corresponding rules of [HAN], and whose validity depends on respect for closed subterm variation, or, rather, a relation that is very much like closed subterm variation. It is possible to contrive one's type system so that it disrespects this kind of variation. For example, one could include a type $T$ and a canonical form $\langle e \rangle$ such that

$$t = t' \in T \text{ iff } \exists e, e'. (\langle e \rangle) \leftarrow t \& (\langle e' \rangle) \leftarrow t' \& e \leftarrow e' \text{ iff } e' \leftarrow e'.$$

Thus, $(x)$ would not be $T$ functional over any type that had a noncanonical member, even though for any canonical terms $e$ and $e'$, $(\langle e \rangle) = (\langle e' \rangle) \in T$.

Respect for closed subterm variation is very desirable, but it prohibits certain forms of very strong intensionality. For example, there can be no term $t$ that is $N$ functional in $x$ over $N$ such that $t[e/x]$ gives a running time for evaluation of each member $e$ of $N$. However, the value of such strong intensionality has not been demonstrated, so it seems that we should, prima facie, design our type systems to respect closed subterm variation.

When $\text{csv}$ is respected so is $\text{csv}^*$, and the relation $\text{csv}^{*0}$, which is used to relate $\text{csv}^*$ to functionality, turns out to have properties that are handy in establishing its instances. One useful fact is that $\text{csv}^{*0}$ is $\text{csv}^{*0*}$. Another will be proved with the help of the following lemma:
5.4 Closed Subterm Variation

\[ t[a/x] \text{ csv } t[b/x] \text{ if } a \text{ csv } b, \]

and so

\[ t[a/x] \text{ csv }^* t[b/x] \text{ if } a \text{ csv }^* b, \]

since \( t[a/x] \text{ csv }^* t[b/x] \) is an equivalence relation in \( a, b \). Now we prove that

\[ t[a/x] \text{ csv }^* s[b/x] \text{ if free } t[a/x] \& \text{ free } s[b/x] \]

\[ \& t \text{ csv }^* s \& a \text{ csv }^* b. \]

**Proof:**

**arb** \( t, s, a, b, x : \)

**assume** the antecedent.

**arb** \( \overline{e}, \overline{y} \text{ s.t. } t[a/x][\overline{e}/\overline{y}] \text{ and } s[b/x][\overline{e}/\overline{y}] \) are closed.

**show** \( t[a/x][\overline{e}/\overline{y}] \text{ csv }^* s[b/x][\overline{e}/\overline{y}]. \)

**enough to show** \( t[\overline{e}, a[\overline{e}/\overline{y}]/\overline{y}, x] \text{ csv }^* s[\overline{e}, b[\overline{e}/\overline{y}]/\overline{y}, x], \)

since \( \text{ free } t[a/x] \& \text{ free } s[b/x]. \)

\( x \) is or is not free in \( t \) and is or is not free in \( s. \)

**assume** \( x \) free in neither \( t \) nor \( s. \)

\( t[\overline{e}, a[\overline{e}/\overline{y}]/\overline{y}, x] \text{ is } t[\overline{e}/\overline{y}]. \)

\( s[\overline{e}, b[\overline{e}/\overline{y}]/\overline{y}, x] \text{ is } s[\overline{e}/\overline{y}]. \)

**show** \( t[\overline{e}/\overline{y}] \text{ csv }^* s[\overline{e}/\overline{y}]. \)

**qed** since \( t \text{ csv }^* s. \)

**assume** \( x \) free in \( t \) but not in \( s. \)

\( s[\overline{e}, b[\overline{e}/\overline{y}]/\overline{y}, x] \text{ is } s[\overline{e}, a[\overline{e}/\overline{y}]/\overline{y}, x]. \)

\( a[\overline{e}/\overline{y}] \) is closed since \( \text{ free } t[a/x] \)

and \( t[\overline{e}, a[\overline{e}/\overline{y}]/\overline{y}, x] \text{ is closed. } \)

**show** \( t[\overline{e}, a[\overline{e}/\overline{y}]/\overline{y}, x] \text{ csv }^* s[\overline{e}, a[\overline{e}/\overline{y}]/\overline{y}, x]. \)

**qed** since \( t \text{ csv }^* s. \)

**assume** \( x \) free in \( s \) but not in \( t. \)

\( t[\overline{e}, a[\overline{e}/\overline{y}]/\overline{y}, x] \text{ is } t[\overline{e}, b[\overline{e}/\overline{y}]/\overline{y}, x]. \)

\( b[\overline{e}/\overline{y}] \) is closed since \( \text{ free } s[b/x] \)

and \( s[\overline{e}, b[\overline{e}/\overline{y}]/\overline{y}, x] \text{ is closed. } \)

**show** \( t[\overline{e}, b[\overline{e}/\overline{y}]/\overline{y}, x] \text{ csv }^* s[\overline{e}, b[\overline{e}/\overline{y}]/\overline{y}, x]. \)

**qed** since \( t \text{ csv }^* s. \)
**assume** $x$ free in both $t$ and in $s$.

$a[e/y]$ and $b[e/y]$ are closed.

$a[e/y]$ $\texttt{csv}^* b[e/y]$ since $a \texttt{csv}^* b$.

$t[e, a[e/y]/y, x] \texttt{csv}^* s[e, a[e/y]/y, x] \texttt{csv}^* s, b[e/y]/y, x$.

QED

We may proceed similarly, anticipating respect for $\alpha$-conversion, with $\texttt{csv}^{\alpha*\alpha}$, which is $\texttt{csv}^{\alpha*\alpha*}$.

$t[a/x] \texttt{csv}^\alpha t[b/x]$ if $a \texttt{csv}^\alpha b$,

since $t[r/x] \alpha t[r'/x]$ if $r \alpha r'$, and so

$t[a/x] \texttt{csv}^{\alpha*} t[b/x]$ if $a \texttt{csv}^{\alpha*} b$.

By a proof similar to the earlier one,

$t[a/x] \texttt{csv}^{\alpha*\alpha} s[b/x]$ if $\texttt{free} t[a/x] \& \texttt{free} s[b/x]$

$\& t \texttt{csv}^{\alpha*\alpha} s \& a \texttt{csv}^{\alpha*\alpha} b$.

It can be shown that type and member equality of $\text{HAN}_n$ respect $\texttt{csv}$.

Prove by induction on — that

if $t \rightarrow s \texttt{csv} s'$ then $\exists t'. t \texttt{csv} t' \rightarrow s'$,

and then show by induction on $n$ that $\text{CTyF}(\text{spine}_n; \text{CSV}_n)$, where

$\text{CSV}_n T \phi$ iff $\forall T'. \text{HAN}_n T' \phi$ if $T' \texttt{csv} T$

$\forall s, t$. if $s \phi t$ then $\forall t'. s \phi t'$ if $t' \texttt{csv} t$.

### 5.5 An Example: Induction

Induction need not be expressed using a special form of noncanonical term that effects recursion. Suppose that 0 is a canonical term and that for any term $t$, $\text{succ}(t)$ is canonical. Suppose $N$ is a type whose equality is the strongest relation $t = s \in N$ such that

$e = e' \in N$ if $0 \rightarrow e, e'$ or $a = a' \in N \& \text{succ}(a) \rightarrow e \& \text{succ}(a') \rightarrow e'$.

Then
\[ \text{\textbf{Proof} by induction on } |\Delta|: \]

The induction step is easy.

\textbf{base case :}

\textbf{arb} \( t, t', T, T', x, y, s, s' : \)

\textbf{assume} \( x \) is not \( y \) & \( y \) is not free in \( T, T' \).

\textbf{assume} \( \text{Fn } t[0/x] = t'[0/x] \in T[0/x] = T'[0/x] \).

\textbf{assume} \( \text{Fn } N : x T, T' : y s = s' \in T[suc(x)/x] = T'[suc(x)/x] \).

\textbf{assume} \( \forall d, e. \text{ if } d \leftarrow e \text{ then } t[d/x] = \text{mem}^\circ t[e/x] \)

& \( t'[d/x] = \text{mem}^\circ t'[e/x] \)

& \( t[suc(x)/x] = \text{mem}^\circ s[t/y] \& \text{free } s[t/y] \)

& \( t'[suc(x)/x] = \text{mem}^\circ s'[t'/y] \& \text{free } s'[t'/y]. \)

\textbf{enough to show} \( \text{Fn } N : x \ t = t' \in T \).

We proceed by induction on \( N \) equality.

\textbf{arb} \( e, e' \text{ s.t. } 0 \leftarrow e, e' \).

\textbf{show} \( t[e/x] = t'[e'/x] \in T[e/x] \).

\( t[0/x] = \text{mem } t[e/x] \& t'[0/x] = \text{mem } t'[e'/x] \).

\textbf{enough to show} \( t[0/x] = t'[0/x] \in T[e/x] \).

\( T[e/x] = T[0/x] \) since \( e = 0 \in N \).

\textbf{enough to show} \( t[0/x] = t'[0/x] \in T[0/x] \).

\( \text{qed}. \)

\textbf{arb} \( e, e', a, a' \text{ s.t. } a = a' \in N \& \text{suc}(a) \leftarrow e \& \text{suc}(a') \leftarrow e' \).

\textbf{assume} \( t[a/x] = t'[a'/x] \in T[a/x] \).

\textbf{show} \( t[e/x] = t'[e'/x] \in T[e/x] \).
\[ t[\text{suc}(a)/x] \simeq \text{mem} \ t[e/x] \ \& \ t'[\text{suc}(a')/x] \simeq \text{mem} \ t'[e'/x]. \]

**enough to show** \( t[\text{suc}(a)/x] = t'[\text{suc}(a')/x] \subseteq T[e/x]. \)

\[ s[a, t[a/x]/x, y] = s'[a', t'[a'/x]/x, y] \subseteq T[\text{suc}(a)/x]. \]

\[ t[\text{suc}(a)/x] \simeq \text{mem} \ s[a, t[a/x]/x, y]. \]

\[ t'[\text{suc}(a')/x] \simeq \text{mem} \ s'[a', t'[a'/x]/x, y]. \]

**enough to show** \( s[a, t[a/x]/x, y] = s'[a', t'[a'/x]/x, y] \subseteq T[e/x]. \)

**enough to show** \( T[e/x] = T[\text{suc}(a)/x]. \)

**QED** since \( e = \text{suc}(a) \in N. \)

If closed subterm variation is respected by member equality or if \( x \) occurs free only as principal arguments in \( t \) then

\[ t[d/x] \simeq \text{mem}^\circ \ t[e/x] \text{ if } d = e. \]

As a familiar case, if \( t \) is \((R, x, y)(x, b[0/x], s)\) of \([HAN]\), then \( x \) occurs free in \( t \) only as the principal argument and

\[ t[\text{suc}(x)/x] \simeq \text{mem}^\circ \ s[t/y]. \]

By a similar but far simpler proof,

\[ \text{Fn } \Delta \triangleright N: x \ \exists \in T \]

if \( x \) is not \( y \) & \( y \) is not free in \( T \)

\& \ Fn \ \Delta \triangleright \exists \in T[0/x]

\& \ Fn \ \Delta \triangleright N: x \ T: y \ \exists \in T[\text{suc}(x)/x]. \]
Chapter 6

Open-endedness

By open-ended use of a type assignment system or a type system hierarchy, I mean its use with anticipation of the possibility that new terms and types will be added. We shall not treat here the open-endedness of type theory with the thoroughness it deserves, but we shall discuss it in passing. In order that the reader not be disappointed later, I say now that the means we shall consider for open-ended use of type assignment systems is not really satisfactorily general; the failing is that only the addition of new terms and new types, not arbitrary new type constructors, is anticipated. Still, some consideration of open-endedness is essential because of its practical and theoretical importance. At least we should get some idea of what would be involved in a proper development.

On the practical side, users of a type system surely expect to incorporate new types and terms, and normally cannot afford to sacrifice hard won knowledge about the old type system, most of which is probably applicable to the enlarged one. To the extent that the users’ knowledge is expressed in a formal proof system, open-ended use should sanction application of only those formal inferences that remain valid under the expected enlargements. To the extent that our expectations about possible enlargements can be expressed precisely, we can accommodate them in our forms of assertion, i.e., in our semantics. Ideally, our assertions would universally quantify over all expected enlargements, obviating any explicit concern about formal proof as regards open-endedness. But I do not know how to achieve this ideal, and so must continue to rely upon judicious restriction of the formal proof system in order to anticipate addition of new type constructors.

On the theoretical side, it seems that the primary significance of open-endedness lies in the ability to express functionality of terms already introduced over domains that include types or functions not yet introduced. Here we see a contrast between the theoretical force of type theory and the theoretical narrowness of type assignment. A type assignment system speci-
ification sets down completely the terms and their values, and the types and their membership. Our assertions, if they are not to become obsolete, must not be tied to a single type system; thus, we are induced to adopt a rather undesirably sophisticated semantics. Martin-Löf's type theory, on the other hand, is open-ended at its core. One need not define all the types of the theory prior to making judgements about those already defined. Indeed, this is essential to the type-theoretic explication of mathematical propositions, since it is scarcely believable that all possible propositions and propositional functions are even extensionally represented among those types and type-valued functions already given explicitly (in [HAN], for example). Now let us turn to the addition of new terms. It is explicitly stated in [HAN] that, "New primitive forms of expression may of course be added when there is need of them." If this open-endedness is significant then it is also essential to the type-theoretic explication of intuitionistic connectives. For, if it is significant, then this is because new forms of expression may embody means of computation or data structuring that cannot be effectively encoded using the forms of expression already introduced. And if that is the case then, since every possible effectively computable total function from a type $A$ to a type $B$ must be representable as a member of $A \rightarrow B$, we must be able to add new members to $A \rightarrow B$.

Finally, let us reconsider what is left to us when we cannot precisely anticipate the addition of new type constructors generally, but we can anticipate the addition of new terms and types. First, we can still anticipate the addition of arbitrary new families of types in extension. To add a new family, select an appropriate index type, adding it as a new type if necessary; then add a new one-place canonical form and add each instance of this new form (for the members of the index type) as a new type. Second, our anticipation of new types will enable us to make precise the argument offered at the end of section 4.3 for restricting ourselves to universe-polymorphic assertions.

We shall work out forms for assertion about type system hierarchies which anticipate the addition of new types and the enlargement of the evaluation system.

### 6.1 Adding New Types

We shall use the apparatus for reflective hierarchy generation introduced in chapter 4. Recall that the central device for producing universes was the insertion of types through the argument of a hierarchy generator $S(T)$,

---

1. Recall that definition of a type in [HAN] consists in connecting a canonical term with its canonical members and its equality relation on canonical members.

2. When a form of expression is introduced, one also explains how it binds variable occurrences in its subterms, and how one evaluates expressions of that form.
which is a one-place equality skeleton-valued function of equality skeletons. In section 4.1 this device was used only to insert universes, but here we shall use it more generally; in addition to inserting the universes, other types may be inserted by means of skeletons \( R \). Also, the base types inserted through the argument of a hierarchy generator must be chosen so as not to interfere with the proper functioning of the other type constructors.\(^3\) This will be handled in our forms of assertion by allowing the use of some property \( T(t) \) that one chooses in order to characterize some of the terms that may be safely inserted as types. We shall make our open-ended assertions independent of the exact choice of terms to be used as universes. So our open-ended assertions will be about hierarchies

\[
(S; R)^*(i.V_i)_n,
\]

where \( S \) is a particular generator, and the types of \( R \) and the universes satisfy \( T \). But we may well expect not to be interested in all these hierarchies. We are likely to want only hierarchies whose type systems have certain properties, e.g., extensional type equality, or respect for \( c_{sy} \). This restricted interest can be expressed in our forms of assertion. Let

\[
\mathcal{O}(T.S(T); A.\Upsilon(A); T.\Theta(T); V; T. \Phi(i.V_i; n.T_n))
\]

where \( S \) is a skeleton operation, \( \Upsilon \) is a property of terms, \( \Theta \) is some property of skeletons, and \( \Phi \) is a relation between sequences of terms and sequences of skeletons, be the relation

\[
\forall V, R. \Phi(i.V_i; n.(S; R)^*(i.V_i)_n)
\]

if \((S; R)^*(i.V_i)_n\) is a reflective hierarchy with universes \( V_i \)
\& \( \forall n. \Theta((S; R)^*(i.V_i)_n) \)
\& \( \forall A. \Upsilon(A) \) if \( A \) type\( R \) or \( \exists i. V_i \xrightarrow{\sim} A. \)

(We may drop the binding and bound variables from an argument when it is clear what is meant, so we may write \( \mathcal{O}(S; \Upsilon; \Theta; \Phi) \).) The intention is that instead of assertions of form \( \Phi(i.V_i; n.T_n) \) for some particular hierarchy \( T_n \) with universes \( V_i \), we use assertions of the form

\[
\mathcal{O}(S; \Upsilon; \Theta; V; T.\Phi(i.V_i; n.T_n)),
\]

where \( S \) is a reflective hierarchy generator that generates a hierarchy each of whose type systems has property \( \Theta \), and terms satisfying \( \Upsilon \) are admissible to \( S \).

\(^3\)For example, if we wanted to anticipate the addition of base types to \( HAN_n \), we would not care to admit as a new base type a term \( A \) such that \((A \rightarrow A) \xrightarrow{\sim} A\), whose membership and equality we are free to define. This would interfere with the arrow type constructor in that we could no longer infer that the members of arrow types have the properties they were intended to have.
We are now in a position to make precise part of the argument given in section 4.3, page 60, for restricting the assertions of one's proof system to universe polymorphic assertions. Recall that the argument turned on our willingness, when anticipating the addition of new base types, to restrict our assertions to those that remain true under uniform increment of universe indices. Our concern is primarily with propositions of the form \( \exists k. \forall n \geq k. \Phi(i.V_i; T_n) \).

If \( O(S; \gamma; \Theta; V, T. \exists k. \forall n \geq k. \Phi(i.V_i; T_n)) \)
then \( O(S; \gamma; \Theta; V, T. \exists k. \forall n \geq k. \Phi(i.V_{i+1}; T_n)) \).

**Proof:**

arb \( S, \gamma, \Theta, \Phi \):

assume \( O(S; \gamma; \Theta; V, T. \exists k. \forall n \geq k. \Phi(i.V_i; T_n)) \).

arb \( V, R \):

assume \( (S; R)^*(i.V_i)_n \) is a reflective hierarchy with universes \( V_i \).

assume \( \forall n. \Theta((S; R)^*(i.V_i)_n) \).

assume \( \forall A. \gamma(A) \) if \( A \) type \( R \) or \( \exists i. V_i \rightarrow A \).

show \( \exists k. \forall n \geq k. \Phi(i.V_{i+1}; (S; R)^*(i.V_i)_n) \).

**enough to show** \( \exists k. \forall n \geq k. \Phi(i.V_{i+1}; (S; R)^*(i.V_i)_{n+1}) \).

let \( R' \) be \( (V_0 \text{ reflect } S(R)) \cup R \).

\( \forall n. (S; R)^*(i.V_i)_{n+1} = (S; R')^*(i.V_{i+1})_n \).

**enough to show** \( \exists k. \forall n \geq k. \Phi(i.V_{i+1}; (S; R')^*(i.V_{i+1})_n) \).

\( (S; R')^*(i.V_{i+1})_n \) is a reflective hierarchy with universes \( V_{i+1} \).

\( \forall n. \Theta((S; R')^*(i.V_{i+1})_n) \).

\( \forall A. \gamma(A) \) if \( A \) type \( R' \) or \( \exists i. V_{i+1} \rightarrow A \).

**QED** by eliminating the first assumption using sequence \( V_{i+1} \) and \( R' \).

From this fact it is easy to see that

\[
O(S; \gamma; \Theta; V, T. \exists k. \forall n \geq k. \Phi(i.V_i; T_n))
\]

iff

\[
O(S; \gamma; \Theta; V, T. \forall j. \exists k. \forall n \geq k. \Phi(i.V_{j+i}; T_n))
\]

iff

\[
O(S; \gamma; \Theta; V, T. UPoly(i.V_i; U. \exists k. \forall n \geq k. \Phi(i.U_{\gamma[j]}; T_n)))
\]

where \( \gamma \) is a level variable. Thus, open-ended assertions of the form in which we are primarily interested are equivalent to their universe polymorphic versions.
6.2 Adding New Terms

We shall explore the matter of adding new terms by trying an example. It will simplify our work to use a simple evaluation system, so we shall adapt the type system FIN (section 2.1) to our purpose by making a hierarchy generator out of it. We must figure out how to abstract our definition of the generator with respect to evaluation systems, and then come up with criteria for an evaluation system's being an acceptable enlargement of the evaluation system of FIN.

We may parameterize FINbody by equality skeletons, letting

\[ \text{FINbody}(T; \psi'; \psi; \phi'; \phi) \]

be the relation

\[ \text{FINbody}(\psi'; \psi; \phi'; \phi) \land \forall A, a, b. \psi(A; a; b) \text{ if } a = b \in_T A \]

\[ \land \phi(A) \text{ if } A \text{ type}_T. \]

This induces an obvious parameterization of FIN which we shall indicate by FIN(T).\(^4\)

What constants of this definition should we abstract from? Of course we mean to abstract from the evaluation system. But the definition of our example also uses three term-valued functions, namely, the operators of our original evaluation system. I do not see how we can expect to find these very functions in all the enlargements of our evaluation system since, after all, the new terms may be different sorts of objects. So it seems we should abstract from the operators as well. Let us call the parameters replacing these operators distinguished operator parameters. The parameterization propagates up through the definition of FIN(T). We may indicate the result by

\[ \text{FIN}\{ E; a, b. \lambda ab; t, a.t(a); A, B. A \rightarrow B \}(T), \]

where E ranges over evaluation system skeletons. Assertions open-endedly using such a parameterized skeleton operation will have the form: for any evaluation system E, and any term-of-E forming operators \( \lambda ab \), \( t(a) \), and \( A \rightarrow B \), if the roles in E of these operators appropriately resemble the roles of the obvious operators in our particular evaluation system, then . . . .

\(^4\)This is equivalent to parameterizing the recursive definition of FIN by equality skeletons, using skeleton variable T, by adding as clauses to the inductive definition

\[ A \text{ type if } A \text{ type}_T \]

and

\[ a = b \in A \text{ if } a = b \in_T A. \]
But how do we expect to complete this schema? We will see in more detail shortly, but it seems clear already that we will want to mention terms constructed from the distinguished operators and also from some distinguished variables. This is mentioned now because we are about to give a relation between $E$, $\lambda x b$, $t(a)$ and $A \rightarrow B$ that expresses a resemblance to our sample evaluation system, and we should make sure to include in the relation the distinguished variable parameters which, even though they don’t appear in our sample parameterized equality skeleton operator, are likely to appear in our open-ended assertions. Since there are denumerably many variables of the sample evaluation system, we shall use a term sequence variable as the parameter for distinguished variables.

Now we must choose a relation between $E$, $\lambda ab$, $t(a)$, $A \rightarrow B$, and our distinguished variable parameter $q$, that expresses a resemblance to our sample evaluation system. The one I have in mind requires

- that distinct numbers index distinct distinguished variables,
- that $\lambda ab$, $t(a)$ and $A \rightarrow B$ be term forming operations of $E$,
- that $\lambda x b$ and $A \rightarrow B$ be canonical forms,
- that $t(a)$ be a noncanonical form with principal argument place $t$,
- that $\beta$-conversion characterize evaluation of $\lambda$ applications, and
- that “old” terms have the “same” values they “had”.

Or to put it more precisely,
∀i, j. \var(q_i) \& i is j if q_i is q_j

\& \exists k_1, k_2, k_3, k_1, k_2, k_3 are distinct
\& ∀x, b. \form( λxb; k_1 ; x ; b )
\& ∀t, a. \form( t(a); k_2 ; (); () ; t, a )
\& ∀A, B. \form( A→B; k_3 ; (); () ; A, B )

\& ∀x, b, A, B, t, a, e. λxb→λxb if λxb is closed
\& (A→B)→(A→B) if A→B is closed
\& e→t(a) iff \exists r. r→t & e→r(a)
\& e→(λxb)(a) iff e→b[a/x] & a is closed

\& ∀c, s. if ∀Φ. Φ(s) if ∀i, b, t, a, A, B. Φ(q_i)
\& Φ(λq_i b) if Φ(b)
\& Φ(t(a)) if Φ(t) & Φ(a)
\& Φ(A→B) if Φ(A) & Φ(B)

then c→s iff ∀R. c R s

if ∀x, b, A, B, e, t, a, r.
λxb R λxb if λxb is closed
& A→B R A→B if A→B is closed
& e R t(a) if r R t & e R r(a)
& e R (λxb)(a) if e R b[a/x] & a is closed.

The requirement that the order (with regard to \form( ; ; ; )) of subterms in t(a) and A→B be as specified above is unnecessary but harmless. Of course, the criteria for acceptable enlargement are not determined by the original evaluation system; I have merely given what I would regard as a typical example of such criteria.

Our example used only three distinguished operators, but in general we must expect any number. To give a uniform treatment, instead of making each such operator a parameter, we could use as parameters a family of operators, together with functions indicating the number of subterms and bound variables to be supplied to each operator of the family. We also need not use sequences of distinguished variables — any family of distinguished variables will do. The details are not critical; there is nothing subtle here, so we shall take the liberty of ambiguously indicating by a simple variable all the distinguished operator and distinguished variable parameters.

Now we are ready to give the form of assertions that are open-ended with regard to both evaluation and type systems. The form is

∀E, O. if Γ(E; O) & E is an evaluation system
then \mathcal{O}(E)\{T. S\{E; O\}(T); A. Y\{E; O\}(A);
T. Θ\{E; O\}(T); V, T. Φ\{E; O\}(i. V_i ; n. T_n )\).
S, Y, Θ, and Φ are essentially as before (in O), but now they may be further parameterized. The extra argument to O just makes explicit the parameterization implicit in the definition of O.
Chapter 7

Propositions and Types

The correspondence discovered by Curry [Curry, Feys, & Craig 58] and Howard [Howard 80] between intuitionistic propositions and types in combinatory logic was elevated by Martin-Löf [Martin-Löf 73, HAN, Padova] to an identity: every proposition is a type and every type is a proposition. This identity, under which truth is inhabitation, is the key to a philosophical analysis that assimilates logic to type theory. Judgements of the forms "A is a proposition" and "A is true" are assimilated to type-theoretic judgements of the forms "A type" and "a ∈ A." The computational content of the proofs that a proposition is true is supposed to be effectively represented by the members of the type that corresponds to that proposition. The paradigms for assimilation are the identification of the connective "if A then B" with the type constructor A → B, and the identification of the quantifier (∀ x ∈ A)B with the type constructor (Π x ∈ A)B. We shall see that this correspondence is not suited to the use of non-type-theoretic type systems, and we shall work out an appropriate weaker correspondence.

The correspondence, let us call it strong, upon which is based the identification of types with propositions is at least as strong as the following correspondence between propositions (P) and types (A):

- the proposition P is true just when the type A is inhabited, and
- the computational content of the proofs that P is true is effectively equivalent to the computational content of the members of A.

We shall not try to determine whether there is more to the strong correspondence than this. A type A is supposed to be identified, under the strong correspondence, with the proposition that A is inhabited. In practice, one sets down a correspondence between the forms of propositions and the forms of types, which is used to express conventional forms of propositions; let us call such a correspondence formal. The standard formal correspondence used with the types defined in [HAN] is as follows:

81
\( A \) is inhabited
\((\forall x \in A)B\)
\(B \text{ if } A\)
\((\exists x \in A)B\)
\(A \& B\)
\(\bot\)
\(A \text{ or } B\)
\(a = b \text{ (in } A)\)

\( A \)
\((\Pi x \in A)B\)
\(A \to B\)
\((\Sigma x \in A)B\)
\(A \times B\)
\(N_0\)
\(A + B\)
\(I(A, a, b)\).

The justification for using this correspondence would be that whenever it holds, the strong correspondence holds.

Let us look at some circumstances under which the standard formal correspondence is of no use. By the weak correspondence between propositions \((P)\) and types \((A)\) we shall mean

- \(P\) is true just when \(A\) is inhabited.

Obviously, when the strong correspondence holds, so does the weak one. Non-constructivists, whether or not they accept the significance of the strong correspondence,\(^1\) would reject even the claim that the standard formal correspondence is justified by the weak correspondence, since, for example, they would not believe that the existence of an effective procedure taking each \(e \in \mathbb{N}\) to a member of \(A[e/x]\) always follows from the inhabitation of \(A[e/x]\) for every \(e \in \mathbb{N}\).\(^2\) Similarly, if used with non-type-theoretic type systems, the validity of the standard formal correspondence with respect to the weak correspondence may be doubted by intuitionists because they might doubt that every effectively computable function is represented in a fixed computation system.

Now we turn to a different objection. It may be doubted, even within type theory, that the standard formal correspondence is valid with respect to the strong correspondence. While the standard formal correspondence may be valid when restricted to types definable exclusively from the type constructors defined in [HAN], the open-endedness of the theory permits the definition of types which undermine the standard correspondence. The correspondence between the type \(A \to B\) and the proposition that "\(B\) is inhabited if \(A\) is" depends upon the computational content of proofs of \(A\) inhabitation being completely represented in the members of \(A\). This is because the procedure by which one attempts to evaluate the application

---

\(^1\)In the case of one who accepts the law of the excluded middle, every provable proposition would have a proof by double negation elimination with no computational content. Hence, there would be no computational content to the body of proofs (of the proposition) as a whole since a procedure that was applicable to every proof (of the proposition) must not require any computationally useful information from such proofs.

\(^2\)even when \(A\) is type functional in \(x\) over \(\mathbb{N}\).
Propositions and Types

\( f(a) \), given \( f \in A \to B \) and \( a \in A \), is supposed to depend only upon \( f \) and \( a \), and not any further upon proofs that \( a \in A \); and yet, proofs that "\( B \) is inhabited if \( A \) is" may yield procedures using any information available in proofs that \( A \) is inhabited. So, if for every type \( B \), proofs of "\( B \) is inhabited if \( A \) is" are to yield members of \( A \to B \), then all the computationally useful information contained in the proofs that \( A \) is inhabited must be contained in the members of \( A \) themselves.\(^3\) Let us call types such as \( A \) member-complete.\(^4\)

It is plausible that member-completeness would be a feature of the types defined using only the type constructors that are already defined in \([HAN]\). However, in \([Constable \& Zlattin 84, Constable 83b]\), Constable defined two new type constructors, the set type and quotient type constructors, that meet the criteria for definition in \([HAN]\), but that can produce incomplete types from complete ones. The so-called set type constructor produces a type \( \{ x \in A \mid B \} \) when \( B \) is a family of types (in \( x \)) over type \( A \). The members are the \( a \in A \) such that \( B[a/x] \) is inhabited, and equality is just the obvious restriction of \( A \) equality. One might even consider the main purpose of this constructor to be the construction of certain incomplete types. This is because the principal difference between \((\Sigma x \in A)B\) and \( \{ x \in A \mid B \} \) is that a procedure over \( \{ x \in A \mid B \} \) may be applied as such only when an \( a \in A \) and a \( b \in B[a/x] \) are effectively available, and yet this procedure must not use \( b \). A procedure over \((\Sigma x \in A)B\) is applicable as such under exactly the same circumstances, but, of course, it may use \( b \). The standard formal correspondence (page 81) fails (to be valid with respect to the strong correspondence) at the first clause, viz., that a type \( A \) corresponds to the proposition that \( A \) is inhabited, because the members of a type might not contain all the computationally useful content of the proofs that the type is inhabited.

Perhaps we have been too liberal in our interpretation of \([HAN]\) in the matter of defining type membership. Let us suspend judgement and proceed to the quotient type constructor. This constructor produces a type \( x, y \in A/\!\!/B \) when \( B \) is a family of types over \( A \) in \( x \) and \( y \), and also a certain condition is met which forces the inhabitation of \( B[a, a'/x, y] \) to be an equivalence relation in \( a \) and \( a' \) over \( A \). The members are the members of \( A \), and equality is (in \( a, a' \in A \)) the inhabitation of \( B[a, a'/x, y] \). According to \([HAN]\),

A canonical type \( A \) is defined by prescribing how a canonical object of type \( A \) is formed as well as how two equal canonical objects of type \( A \) are formed. There is no limitation on this

---

\(^3\)What we mean here by computationally useful information in the objects of some kind is the information usable by procedures that are applicable to every object of that kind.

\(^4\)It will not be necessary here to be more explicit about what member-completeness is.
prescription except that the relation of equality which it defines between canonical objects must be reflexive, symmetric and transitive.

Even if, in the definition of the set type constructor, we have misunderstood what is meant by prescription, it would be difficult to believe that the relations referred to, and of which reflexivity, symmetry and transitivity may be predicated, are not to include the relations expressed by two-place type-valued functions.

Quotient types may be incomplete in a slightly more subtle sense than we have so far considered. There may be computationally useful information in the proofs of \( a = a' \in x, y \in A // B \) that is not in the proofs of \( a \in A \) and \( a' \in A \). This happens, of course, when proofs that \( B[a, a'/x, y] \) is inhabited contain computationally useful information, which may happen even when \( B[a, a'/x, y] \) is member-complete since proofs of inhabitation yield members. Let us call a type \( A \) equality-complete when, for \( a, a' \in A \), the proofs of \( a = a' \in A \) have no computationally useful content beyond that of the proofs of \( a \in A \) and \( a' \in A \).

We shall now see how, by means of type constructors already defined in [HAN], member-incomplete types can be constructed from equality-incomplete types. And so, if we are still to reject the possibility of defining member-incomplete types, which appears to be the basis for rejecting the set-type constructor, then we must eliminate the offending type constructors. If a type \( A \) is equality-incomplete and \( a, a' \in A \), then \( I(A, a, a') \) is a member-incomplete type. The \( I(, ,) \) constructor was defined simply in order to provide a propositional representation for type equality, so we might be willing to sacrifice it if some other means presented itself. But \( \Pi \), too, can be used to build member-incomplete types from equality-incomplete types. For example, suppose \( T \) is \( x, y \in N // A \), and \( S \) is \( x, y \in N // B \), and \( T \) and \( S \) are types. Then from a proof that \( f \in T \rightarrow S \), and a proof that \( e = e' \in N \), and a proof that \( a \in A[e, e'/x, y] \), one can effectively obtain a \( b \) with a proof that \( b \in B[f(e), f(e')/x, y] \). But what intuitionist would believe that, for every such \( A \) and \( B \), such a \( b \in B[f(e), f(e')/x, y] \) is effectively obtainable from \( f \), \( e \) and \( e' \) alone (which would be so if \( T \rightarrow S \) were member-complete)? Abandoning \( \Pi \) is out of the question, so it seems that we must not read [HAN] as both excluding the definition of member-incomplete types and permitting the definition of equality-incomplete types. And since it appears that equality-incomplete types are definable, we shall, henceforth, assume that incomplete types are definable. Clearly, the formal correspondence must be modified if it is to be justified by strong correspondence; but, we shall not pursue the matter here.

\(^5\)Recall that any canonical member of \( I(A, a, a') \) is \( r \) and that \( I(A, a, a') \) is inhabited just when \( a = a' \in A \).
It is possible, using incomplete types, to find a nonstandard formal correspondence that is valid with respect to the weak correspondence both classically and constructively, and both type-theoretically and non-type-theoretically. The only clauses of the standard formal correspondence that we need to change are the ones about universal quantification and implication. The other clauses obviously entail weak correspondence. Let fact be a canonical term, and let $\|A\|$ be a type constructor such that, given constituent type $A$,

- $\|A\|$ is a type that is inhabited just when $A$ is, and
- if $\|A\|$ is inhabited then $\text{fact} \in \|A\|$, and
- if $A = B$ then $\|A\| = \|B\|$.

Such a type constructor can be defined using the set type constructor by letting $\text{fact}$ be 0, and defining $\|A\|$ to be $\{N \mid A\}$\(^6\). The use of a similar constructor is discussed in [Constable et al. 86], where it is called the squash operator. If $B$ is a family of types in $x$ over $A$, then $(\Pi x \in A)\|B\|$ is inhabited (by $\lambda x \text{fact}$) just if, for every $a \in A$, $B[a/x]$ is inhabited. Thus, we may include in our formal correspondence

\[
(\forall x \in A)B \equiv (\Pi x \in A)\|B\| \quad B \text{ if } A \quad A -\|B\|.
\]

Retaining the standard clauses about the other forms of proposition leaves us with a nonstandard formal correspondence that is valid with respect to the weak correspondence.

This formal correspondence shows one way of representing the usual connectives and quantifiers, but one might sometimes prefer to use the weak correspondence in other ways in order to avoid introducing $\|B\|$ unnecessarily. For example, given a family of types $B_x (x \in A)$, when there is a $b$ such that

\[\forall a \in A. b \in B_a \text{ if } \exists b \in B_a,\]

then $(\Pi x \in A)B_x$ is inhabited (by $\lambda x b$) if and only if $(\forall x \in A)(\exists b \in B_x)$. A simple way to exploit this would be to identify an easily recognized class of type constructors, let us call them plain, for which we have obvious candidates for membership:

- $N_n$, $N$, $I(A,a,b)$, $\|A\|$, and $U_k$ are plain.
- $A \times B$ is plain iff $A$ and $B$ are.

\(^6\)which is a type equal to $\{ x \in N \mid A \}$ for any $x$. 

• \((\Pi x \in A)B\) is plain iff \(B\) is.

• \(\{ A \mid B \}\) and \(x, y \in A\) if \(B\) are plain iff \(A\) is.

• No other terms are plain.

Let \((\ast B)\) be \(B\) if \(B\) is plain and \(\|B\|\) otherwise; we may add the following to our nonstandard correspondence:

\[
\begin{align*}
(\forall x \in A)B & \quad (\Pi x \in A)(\ast B) \\
B \text{ if } A & \quad A \rightarrow (\ast B).
\end{align*}
\]

Although the weak correspondence cannot do the full work of the strong correspondence in type theory, it is still useful when the computational content of a proposition is not explicitly needed. And, of course, when using non-type-theoretic type systems, we can make no use of the strong correspondence anyway, and so we must then resort to something weaker. It should be noted that the weak correspondence directly permits the expression of classical propositions and the use of classical axioms, rather than requiring some sort of reinterpretation of classical into constructive propositions such as is described in [Constable et al. 86]. For example, those who wish to use the law of the excluded middle need only use the inference rule

\[
\begin{array}{c}
A \text{ type} \\
\hline
\text{fact} \in \|A + (A \rightarrow N_0)\|
\end{array}
\]

or, using the sequent form \(\exists \in T\),

\[
\begin{array}{c}
A \text{ type} \\
\hline
\exists \in A + (A \rightarrow N_0)
\end{array}
\]

The rest of this chapter consists of a discussion of whether propositions ought to be represented by types when non-type-theoretic type systems are used. One reason would be to make a large body of work amenable to both type-theoretic and non-type-theoretic interpretation, but we shall concentrate on reasons internal to non-type-theoretic practice. Two other approaches to treating propositions occur to me. The first, open only to non-constructivists, is to select two terms to be used as truth values, say, \(0_2\) and \(1_2\), and represent propositions by terms that evaluate to one of these, that is, represent propositions by members of \(N_2\). One may then represent properties by their characteristic functions, and so forth. But, of course, effective evaluation must be abandoned as soon as predicates are introduced that are not effectively decidable. This is probably sufficiently discouraging.

The second approach would be to introduce propositional apparatus directly. Special propositional forms would be introduced, while predicates
would be represented as proposition-valued functions. Besides the usual quantifiers and connectives, the forms “\(A\) inhabited” and \(I(A,a,b)\) would be added, with \(I(,\,\,)\) no longer needed among the type constructors. Besides removing \(I(,\,\,)\) from the roster of type constructors, the set and quotient type constructors would be reformed as \(\{ x \in A \mid P_x \}\) and \(x,y \in A//R_{x,y}\), where \(P_x\) and \(R_{x,y}\) must be proposition-valued expressions over type \(A\). Further, we would add new simple sequents

\[
P = \text{prop} Q
\]

to express an appropriate equality between propositions, which is needed for the concept of proposition-valued function. And we would add simple sequents of the form

\[
P, Q \text{ true}
\]

to express the truth of (equal) propositions, and a new form of complex sequent,

\[
P, Q \sigma
\]

to express the hypothesis that \(P\) is true (and equal to \(Q\)). The extension of \(\text{Fn}\) to express these things is obvious, and we may abbreviate “\(P = \text{prop} P\)” as “\(P \text{ prop}\)”, “\(P, Q \text{ true}\)” as “\(P \text{ true}\)”, and “\(P, Q \sigma\)” as “\(P \sigma\)”. In a reflective hierarchy \(T_n\), we would reflect the propositional equality of \(T_n\) by means of a type \(\text{Prop}_n\) (of type system \(T_{n+1}\)).

Can the nonstandard correspondence given above, justified by the weak correspondence, be applied in order to make all this extra propositional machinery redundant? The answer depends upon whether the equality between propositions is easily expressible as a relation between the corresponding types (under the nonstandard formal correspondence). Supposing that we were to use types instead of propositions, then propositional equality would become a relation between types. If we can design \(\|A\|\) so that \(\|A\| = \|B\|\) just when \(A = \text{prop} B\), then we can eliminate the new propositional machinery:

- Let \(A = \text{prop} B\) be \(\|A\| = \|B\|\).
- Let \(A, B \text{ true}\) be \(\exists \in \|A\| = \|B\|\) or \(\text{fact} = \text{fact} \in \|A\| = \|B\|\).
- Let \(A, B \sigma\) be \(\|A\|, \|B\| : z \sigma\) for some \(z\) not free in \(\sigma\).
- Let \(\text{Prop}_i\) be \(X, Y \in V_i//I(V_i, \|X\|, \|Y\|)\).

Perhaps the clearest case in which, by proper design of \(\|A\|\), this elimination may be accomplished is that in which propositional equality is simply biconditionality.

It should be noted that, in practice, the elimination of the special propositional sequent forms need not always introduce \(\|A\|\). For example,
if \( \text{Fn } \Delta \triangleright A \text{ type} \) then \( \text{Fn } \Delta \triangleright A \text{ true} \) iff \( \text{Fn } \Delta \triangleright \exists \in A \),

and

if \( \text{Fn } \Delta \triangleright A \text{ type} \) & \( z \) not free in \( \sigma \)
then \( \text{Fn } \Delta \triangleright A \triangleright \sigma \) iff \( \text{Fn } \Delta \triangleright A : z \triangleright \sigma \).

Another possible approach to avoiding the new sequent forms would be to refrain altogether from explicitly formulating propositional equality and proposition-valued functionality in the formal system, leaving their proper expression to the user.
Chapter 8

Pointwise Functionality

We shall consider some properties of sequents intended for use instead of Fn and HFn in proof systems, especially Nuprl. The notion of pointwise functionality was introduced for the semantics of Nuprl [Constable et al. 86] (see section 1.2). It will be generalized here in order to make it more useful and to make thinking about it in relation to Fn easier. We shall use a hybrid form that can include both negative and positive assumptions. Much of this chapter consists of rather uninteresting proofs aimed at establishing several basic theorems about pointwise functionality. These proofs are carried out here only because it would be too unkind to ask even the interested reader to do it. That these basic theorems are much more difficult to establish than the corresponding theorems about simple functionality, Fn, may well be considered a criticism against using pointwise functionality. The principal known advantage of pointwise functionality seems to be that induction is more expressive (see page 101).

Consider the problem of showing that \((\Pi x \in A)B\) is a type. This reduces to showing \(Fn\; A : x\; B\) type, which reduces to finding a natural deduction style proof of the form

\[
\vdots \\
A \text{ type} \\
\vdots \\
\text{assume } x \in A \\
\vdots \\
B \text{ type}
\]

where the parts indicated by dots contain no undischarged assumptions. This reduces to the independent problems of (1) producing the proof of \(Fn\; A\) type, and (2) showing how to proceed from an arbitrary proof of \(Fn\; A\) type to a whole proof of \(Fn\; A : x\; B\) type. More generally, the problem of showing \(Fn\)
\( \Delta \triangleright \sigma \) reduces to giving a proof of \( \text{Fn} \, \Delta \) and showing how to proceed from an arbitrary proof of \( \text{Fn} \, \Delta \) to a proof of \( \text{Fn} \, \Delta \triangleright \sigma \).

One principle underlying the so-called refinement style of Nuprl proofs [Constable et al. 86] is that the problem of "proving a sequent" is to be reduced to zero or more independent problems of proving other sequents. This principle precludes construing each refinement proof as a proof that the sequent it proves has property \( \text{Fn} \), because the important method of reduction just discussed is not available; that one can proceed from an arbitrary proof of \( \text{Fn} \, \Delta \) to a proof of \( \text{Fn} \, \Delta \triangleright \sigma \) is not generally captured in any obvious way by the provability of \( \text{Fn} \, \tau \) for some \( \tau \). One approach to explaining a refinement style proof system would be to first set down a natural deduction style system, and then construe a refinement proof of \( \Delta \triangleright \sigma \) as a formal description of how to proceed from any natural deduction proof of \( \text{Fn} \, \Delta \) to a natural deduction proof of \( \text{Fn} \, \Delta \triangleright \sigma \). Another, more direct, approach would be to use assertions of the form

\[
\text{Fn} \, \Delta \triangleright \sigma \text{ if } \text{Fn} \, \Delta
\]

as the unit of proof.

Nuprl takes another approach. One rather natural way of reading an assumption \( x \in A \) is as "suppose \( x \) is an arbitrary member of \( A \)" without any presuppositions. Thus we might naturally read

\[
t \in T \left( x \in A \quad y \in B \right)
\]

as

\[
\text{for any } x \text{ in } A \text{ and any } y \text{ in } B, \text{ } t \text{ is in } T,
\]

or more precisely as

\[
\forall a, b, \text{ if } a \in A \& b \in B[a/x] \text{ then } t[a, b/x, y] \in T[a, b/x, y].
\]

This is almost Nuprl's criterion for the truth of the sequent

\[
A : x \quad B : y \quad t \in T.
\]

All that is missing is a certain kind of respect for replacement of \( a \) and \( b \) by equal terms. The real criterion is that

\[
\forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)

\[
& \forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)

\[
& \forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)

\[
& \forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)

\[
& \forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)

\[
& \forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)

\[
& \forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)

\[
& \forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)

\[
& \forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)

\[
& \forall a, b, \text{ if } a \in A \& b \in B[a/x] \& \forall a', B[a, x] = B[a'/x] \text{ if } a = a' \in A
\]

then \( \forall a', b', \text{ if } a = a' \in A \& b = b' \in B[a/x]
\]

then \( t[a, b/x, y] = t[a', b'/x, y] \in T[a, b/x, y] \)

& \( T[a, b/x, y] = T[a', b'/x, y], \)
for which we may say that \( t \) is pointwise \( T \) functional in \( x \) and \( y \) over \( A \) and \( B \). If, in addition to this condition, \( B \) should be type functional over \( A \) in \( x \), then \( t \) would be \( T \) functional in \( x \) and \( y \) over \( A \) and \( B \).

Let \( a = b = c \in A \) mean \( a = c \in A \) and \( b = c \in A \). We shall define pointwise functionality by means of a relation \( \text{Fn}@() \sigma \) defined such that

\[
\text{Fn}@() \sigma \text{ iff Fn } \sigma
\]

and

\[
\text{Fn}@() A,B : x \sigma \text{ iff } A = B \& a \in A \\
& \& \forall b,b'. \text{Fn}@() \sigma[b;b'/x] \text{ if } b = b' = a \in A
\]

and

if \( \sigma \) is simple then \( \text{Fn}@() \sigma \) iff \( \text{Fn } \sigma \).\(^1\)

It will be handy to have the relation \( \text{Fn}@() \Delta \gg \sigma \) such that

\[
\text{Fn}@() \Delta \gg \sigma \text{ iff } \text{Fn}@() \Delta > \sigma \& |\vec{\epsilon}| \text{ is } |\Delta|.
\]

The effect of \( \text{Fn}@() \) could be gotten through \( \text{Fn} \) if there were a type constructor \( A@() \) such that

\[
A@()B@() \text{ iff } A = B \& a = b \in A
\]

and

\[
t = s \in A@() \text{ iff } t = s = a \in A.\(^2\)
\]

Then

\[
\text{Fn}@() B : x \gg \sigma \text{ iff } \text{Fn}@() \Delta \gg \Delta@()a,B@()a : x \sigma.
\]

Now we define pointwise functionality \( \text{PwFn}(m;\sigma) \) as follows.

\[
\text{PwFn}(|\Delta|; \Delta > \sigma) \text{ iff } \forall \vec{\epsilon}. \text{Fn}@() \Delta \gg \sigma \text{ if } \text{Fn}@() \Delta \gg
\]

and

if \( \sigma \) is simple then \( \text{PwFn}(|\Delta| + m; \Delta > \sigma) \) iff \( \text{PwFn}(|\Delta|; \Delta > \sigma) \).

It will be convenient if we can explicitly indicate the break between the negative and positive assumptions of a sequent. Let

\[
\text{PwFn } \Delta \gg \sigma \text{ iff } \text{PwFn}(|\Delta|; \Delta > \sigma).
\]

So,

\(^1\)This turns out to be convenient.

\(^2\)Indeed, there is such a constructor in Nuprl, viz., \( \{ z \in A | I(A,z,a) \} \).
PwFn $\Delta \gg \sigma$ iff $\forall \vec{e}. \text{Fn@}(\vec{e}) \Delta \gg \sigma$ if \text{Fn@}(\vec{e}) \Delta \gg$.

The basic connection, proved below, between negative and positive assumptions is that

PwFn $\Delta \gg T : z \sigma$ iff PwFn $\Delta T : z \gg \sigma \&$ PwFn $\Delta \gg T$ type.

This justifies the following fact about Nuprl:

PwFn $\Delta \gg (\Pi x \in A)B$ type iff PwFn $\Delta \gg A$ type
\[
\& \text{PwFn } \Delta A : x \gg B \text{ type.}
\]

### 8.1 Fn@ Facts

We shall need a number of lemmas about Fn@ to establish the theorems about PwFn of the next section (page 97), to which the reader may prefer to skip ahead. The statements from section 3.1, pages 39–43, to which the dagger $\dagger$ is prefixed remain true when the property Fn is replaced by \text{Fn@}(\vec{e})$\textsuperscript{3}$, while those to which the double dagger $\ddagger$ is prefixed remain true when Fn is replaced by \text{Fn@}(\vec{e}) such that $|\vec{e}| \leq |\Delta|$. We will not bother to refer to these facts explicitly in the proofs below. The remainder of our lemmas will be labeled. First we shall list them without proof.

(A) If \text{Fn@}(\vec{e}, \vec{s}) \Delta \gg \sigma \& |\vec{e}| = |\Delta|$ then \text{Fn@}(\vec{s}) \sigma[\vec{e}/\vec{\Delta}].

(A') If \text{Fn@}(a, \vec{e}) \Delta A : x \gg t \in T
\[
\begin{align*}
\forall b, b'. \ t'[b, \vec{e}/x, \vec{\Delta}] &= t'[b', \vec{e}/x, \vec{\Delta}] \\
&= T[b', \vec{e}/x, \vec{\Delta}] = T[b', \vec{e}/x, \vec{\Delta}]
\end{align*}
\]

if $b = b' = a \in A$.

(B) \text{Fn@}(\vec{e}, t) \Delta T : z \gg$ iff $t$ is closed & \text{Fn@}(\vec{e}) \Delta \gg t \in T$.

(C) \text{Fn@}(\vec{e}, t) \Delta T : z \gg$ iff $t \in T[\vec{e}/\vec{\Delta}]$ & \text{Fn@}(\vec{e}) \Delta \gg T$ type.

(D) If $|\vec{e}| = |\Delta|$ \& $t = s \in T[\vec{e}/\vec{\Delta}]$
\[
\text{then } \text{Fn@}(\vec{e}, t, \vec{r}) \Delta T : z \gg \sigma \text{ iff } \text{Fn@}(\vec{e}, s, \vec{r}) \Delta T : z \gg \sigma.
\]

$\textsuperscript{3}$Most of these follow easily from statements of the form

$\forall \vec{e}, \Delta, \sigma, \vec{r}. \text{Fn@}(\vec{e}, \vec{r}_0) \Delta \gg \sigma_0$ if $|\vec{e}| \leq |\Delta|
\[
\& \text{if } |\vec{e}| < |\Delta|$ then $\forall i < k + 1. \vec{r}_i$ is empty
\[
\& \Psi(i, \sigma_i ; i, \vec{r}_i)
\]

for some $k$ and some $\Psi$ such that

$\forall \sigma, \vec{r}, a, b, z. \Psi(i, \sigma_i ; a; b/z ; i, \vec{r}_i)$ if $\Psi(i, \sigma_i ; i, \vec{r}_i)$ \& $a, b$ are closed.

By induction on $\vec{e}$, such a statement follows from the base case (empty $\vec{e}$).
(E) If $\text{Fn}(@\bar{e}, t[\bar{e}/\tilde{\Delta}], \bar{r}) \Delta T : z \triangleright \sigma$
& $\text{Fn}(@\bar{e}) \Delta \gg s = t \in T$
& $\text{Fn}(@\bar{e}) \Delta \gg t = s' \in T$
& free $\sigma[s; s'/z]$
then $\text{Fn}(@\bar{e}, \bar{r}) \Delta \triangleright \sigma[s; s'/z]$.

(F) If $|\bar{e}|$ is $|\Delta|$
then $\text{Fn}(@\bar{e}, t, \bar{r}) \Delta T : z \triangleright \sigma$ iff $\text{Fn}(@\bar{e}, t) \Delta T : z$
& $\forall s, s'$. $\text{Fn}(@\bar{e}, \bar{r}) \Delta \triangleright \sigma[s; s'/z]$
if $s = s' = t \in T[\bar{e}/\tilde{\Delta}]$.

(G) $\text{Fn}(@\bar{e}) \Delta \gg T : z \sigma$ iff $\text{Fn}(@\bar{e}) \Delta \gg T$ type
& $\forall t$. $\text{Fn}(@\bar{e}, t) \Delta T : z \gg \sigma$ if $t \in T[\bar{e}/\tilde{\Delta}]$.

Now we consider why these are true. The proof of (A) proceeds easily by induction on $\bar{e}$; (A') follows easily from (A). After (B) is generalized slightly by doubling the type occurrences\(^4\), the proof by induction on $\bar{e}$ is easy.

To prove (C) it is sufficient, in light of (B), to prove that

$t$ is closed & $\text{Fn}(@\bar{e}) \Delta \gg t \in T$ iff $t \in T[\bar{e}/\tilde{\Delta}]$ & $\text{Fn}(@\bar{e}) \Delta \gg T$ type.

One direction is immediate in light of (A), so it is enough to show that

$\text{Fn}(@\bar{e}) \Delta \gg t = t \in T = T'$ if $t \in T[\bar{e}/\tilde{\Delta}]$ & $\text{Fn}(@\bar{e}) \Delta \gg T = T'$.

**Proof** by induction on $\bar{e}$:

base case : trivial ;

arb $a, \bar{e}$ s.t. the induction hypothesis holds for $\bar{e}$.

arb $A, A', x, \Delta, T, T', z, t :$

assume $t \in T[a, \bar{e}/x, \tilde{\Delta}]$.

assume $\text{Fn}(@a, \bar{e}) A, A' : x \Delta \gg T = T'$.

arb $b, b'$ s.t. $b = b' = a \in A$.

show $\text{Fn}(@\bar{e}) (\Delta \triangleright t = t \in T = T')[b; b'/x]$.

$\text{Fn}(@\bar{e}) (\Delta \triangleright T = T')[b; b'/x]$.

enough by ind hyp to show $t \in T[b, \bar{e}/x, \tilde{\Delta}]$.

enough to show $T[a, \bar{e}/x, \tilde{\Delta}] = T[b, \bar{e}/x, \tilde{\Delta}]$.

QED by (A').

To prove (D) it is enough to show that

\(^4\)that is by using $T, T' : z$ instead of $T : z$ and $t = t \in T = T'$ instead of $t = t \in T$,
if $|\tilde{e}|$ is $|\Delta|$ & $t = s \in T[\tilde{e}/\tilde{\Delta}]$

& $\text{Fn@}(\tilde{e}, t, \tilde{r}) \Delta T, T': z \triangleright \sigma$

then $\text{Fn@}(\tilde{e}, s, \tilde{r}) \Delta T, T': z \triangleright \sigma$.

**Proof** by induction on $\tilde{e}$:

**base case**: trivial ;

**arb** $a, \tilde{e}$ s.t. the induction hypothesis holds for $\tilde{e}$.

**arb** $A, A', x, \Delta, t, s, T, T', z, \tilde{r}, \sigma$ :

**assume** $|\tilde{e}|$ is $|\Delta|$.

**assume** $t = s \in T[a, \tilde{e}/x, \tilde{\Delta}]$.

**assume** $\text{Fn@}(a, \tilde{e}, t, \tilde{r}) A, A': x \Delta T, T': z \triangleright \sigma$.

**arb** $b, b'$ s.t. $b = b' = a \in A$.

**show** $\text{Fn@}(\tilde{e}, s, \tilde{r}) (\Delta T, T': z \triangleright \sigma)[b; b'/x]$.

$\text{Fn@}(\tilde{e}, t, \tilde{r}) (\Delta T, T': z \triangleright \sigma)[b; b'/x]$.

**enough** by ind hyp to show $t = s \in T[b, \tilde{e}/x, \tilde{\Delta}]$.

**enough to show** $T[a, \tilde{e}/x, \tilde{\Delta}] = T[b, \tilde{e}/x, \tilde{\Delta}]$.

$\text{Fn@}(a, \tilde{e}) A, A': x \Delta T$ type by (C).

**QED** by $(A')$.

To prove (E) it is enough to show that

if $\text{Fn@}(\tilde{e}, t[\tilde{e}/\tilde{\Delta}], \tilde{r}) \Delta T, T': z \triangleright \sigma$

& $\text{Fn@}(\tilde{e}) \Delta \gg s = t' \in T = T'$

& $\text{Fn@}(\tilde{e}) \Delta \gg t = t' \in T = T'$

& $\text{Fn@}(\tilde{e}) \Delta \gg t = s' \in T = T'$

& free $\sigma[s; s'/z]$

then $\text{Fn@}(\tilde{e}, \tilde{r}) \Delta \triangleright \sigma s; s'/z]$.

**Proof** by induction on $\tilde{e}$:

**base case**: trivial ;

**arb** $a, \tilde{e}$ s.t. the induction hypothesis holds for $\tilde{e}$.

**arb** $A, A', x, \Delta, T, T', z, t, t', s, s', \tilde{r}, \sigma$ :

**assume** $\text{Fn@}(a, \tilde{e}, t[a, \tilde{e}/x, \tilde{\Delta}], \tilde{r}) A, A': x \Delta T, T': z \triangleright \sigma$.

**assume** $\text{Fn@}(a, \tilde{e}) A, A': x \Delta \gg s = t' \in T = T'$.

**assume** $\text{Fn@}(a, \tilde{e}) A, A': x \Delta \gg t = t' \in T = T'$.

**assume** $\text{Fn@}(a, \tilde{e}) A, A': x \Delta \gg t = s' \in T = T'$. 
assume \(\text{free} \ \sigma[s; s'/z]\).

arb \(b, b' \text{ s.t.} \ b = b' = a \in A\).

show \(\text{Fn}(\bar{\varepsilon}, \bar{\tau})(\Delta \triangleright \sigma[s; s'/z])[b; b'/x]\).

\(\text{Fn}(\bar{\varepsilon})(\Delta \triangleright s = t' \in T = T')[b; b'/x]\).

\(\text{Fn}(\bar{\varepsilon})(\Delta \triangleright t = t' \in T = T')[b; b'/x]\).

\(\text{Fn}(\bar{\varepsilon})(\Delta \triangleright s' \in T = T')[b; b'/x]\).

\(\text{free} (\sigma[b, \bar{\Delta}, z; b', \bar{\Delta}, z/x, \bar{\Delta}, z]) [s[b, \bar{\Delta}/x, \bar{\Delta}] ; s'[b', \bar{\Delta}/x, \bar{\Delta}]/z]\).

enough by ind hyp to show \(\text{Fn}(\bar{\varepsilon}, t[b, \bar{\varepsilon}/x, \bar{\Delta}], \bar{\tau})(\Delta T, T': z \triangleright \sigma)[b; b'/x]\).

enough to show \(\text{Fn}(\bar{\varepsilon}, t[b, \bar{\varepsilon}/x, \bar{\Delta}], \bar{\tau}) A, A': x \Delta T, T': z \triangleright \sigma\).

enough by (D) to show \(t[a, \bar{\varepsilon}/x, \bar{\Delta}] = t[b, \bar{\varepsilon}/x, \bar{\Delta}] \in T[a, \bar{\varepsilon}/x, \bar{\Delta}]\).

QED by \(A'\).

As a corollary to (E), in case \(t\) is closed, we get

if \(|\bar{\varepsilon}| \leq |\Delta| \& \text{Fn}(\bar{\varepsilon}, t, \bar{\tau}) \Delta T : z \triangleright \sigma\)
then \(\forall s, s'. \text{Fn}(\bar{\varepsilon}, \bar{\tau}) \Delta \triangleright \sigma[s; s'/z] \text{ if } s = s' = t \in T[\bar{\varepsilon}/\bar{\Delta}]\).

This gives us one direction of (F), so to prove (F) it is enough to show that

if \(\text{Fn}(\bar{\varepsilon}, t) \Delta T, T': z \gg\)
\(\& \forall s, s'. \text{Fn}(\bar{\varepsilon}, \bar{\tau}) \Delta \triangleright \sigma[s; s'/z] \text{ if } s = s' = t \in T[\bar{\varepsilon}/\bar{\Delta}]\)
then \(\text{Fn}(\bar{\varepsilon}, t, \bar{\tau}) \Delta T, T': z \triangleright \sigma\).

Proof by induction on \(\bar{\varepsilon}\):

base case: trivial;

arb \(a, \bar{\varepsilon} \text{ s.t.} \) the induction hypothesis holds for \(\bar{\varepsilon}\).

arb \(A, A', x, \Delta, T, T', z, \bar{\tau}, \sigma\):

assume \(\text{Fn}(\bar{\varepsilon}, t, \bar{\tau}) A, A': x \Delta T, T': z \gg\).

assume \(\forall s, s'. \text{Fn}(\bar{\varepsilon}, \bar{\tau}) A, A': x \Delta \triangleright \sigma[s; s'/z] \text{ if } s = s' = t \in T[a, \bar{\varepsilon}/x, \bar{\Delta}]\).

arb \(b, b' \text{ s.t.} \ b = b' = a \in A\).

show \(\text{Fn}(\bar{\varepsilon}, t, \bar{\tau})(\Delta T, T': z \triangleright \sigma)[b; b'/x]\).

\(\text{Fn}(\bar{\varepsilon}, t)(\Delta T, T': z)[b; b'/x] \gg\).

arb \(s, s' \text{ s.t.} \ s = s' = t \in T[a, \bar{\varepsilon}/x, Dv]\).

enough by ind hyp to show \(\text{Fn}(\bar{\varepsilon}, \bar{\tau})(\Delta \triangleright \sigma[s; s'/z])[b; b'/x]\).

enough to show \(\text{Fn}(\bar{\varepsilon}, \bar{\tau}) A, A': x \Delta \triangleright \sigma[s; s'/z]\).

enough to show \(s = s' = t \in T[a, \bar{\varepsilon}/x, \bar{\Delta}]\).
enough to show $T[a, \bar{e}/x, \tilde{\Delta}] = T[b, \bar{e}/x, \tilde{\Delta}]$.

enough by (A') to show $\text{Fn@}(a, \bar{e}) A, A' : x \Delta \triangleright T$ type.

QED by (B).

One direction of (G) follows from

$\text{Fn@}(\bar{e}, t) \Delta T, T' : z \triangleright \sigma$ if $t \in T[\bar{e}/\tilde{\Delta}] \land \text{Fn@}(\bar{e}) \Delta \triangleright T, T' : z \sigma$.

Proof by induction on $\bar{e}$:

base case:

arb $t, T, T', z, \sigma$:
assume $t \in T$.
assume $\text{Fn } T, T' : z \sigma$.
arb $s, s'$ s.t. $s = s' = t \in T$.
show $\text{Fn } \sigma[s; s'/z]$.
qed since $s = s' \in T$.

arb $a, \bar{e}$ s.t. the induction hypothesis holds for $\bar{e}$.

arb $A, A', x, \Delta, T, T', z, t, \sigma$:
assume $t \in T[a, \bar{e}/x, \tilde{\Delta}]$.
assume $\text{Fn@}(a, \bar{e}) A, A' : x \Delta \triangleright T, T' : z \sigma$.
arb $b, b'$ s.t. $b = b' = a \in A$.
show $\text{Fn@}(\bar{e}, t) (\Delta \triangleright T, T' : z \triangleright \sigma)[b; b'/x]$.
$\text{Fn@}(\bar{e}) (\Delta \triangleright T, T' : z \sigma)[b; b'/x]$.
enough by ind hyp to show $t \in T[b, \bar{e}/x, \tilde{\Delta}]$.
enough to show $T[a, \bar{e}/x, \tilde{\Delta}] = T[b, \bar{e}/x, \tilde{\Delta}]$.
$\text{Fn@}(a, \bar{e}) A, A' : x \Delta \triangleright T$ type by (C).
QED by (A').

To prove (G) it is enough to show that

$\text{Fn@}(\bar{e}) \Delta \triangleright T, T' : z \sigma$ if $\text{Fn@}(\bar{e}) \Delta \triangleright T = T'$
& $\forall t. \text{Fn@}(\bar{e}, t) \Delta T, T' : z \triangleright \sigma$ if $t \in T[\bar{e}/\tilde{\Delta}]$.

Proof by induction on $\bar{e}$:

base case:

arb $T, T'$ s.t. $T = T'$. 
arb σ, z s.t. ∀t. Fn@ (t) T, T': z ≫ σ if t ∈ T.

arb t, t' s.t. t = t' ∈ T.

show Fn σ [t; t'/z].

Fn@ (t) T, T': z ≫ σ since t ∈ T.

qed since t = t' = t ∈ T.

arb a, e s.t. the induction hypothesis holds for e.

arb A', x, Δ, T, T', z, σ, t :

assume Fn@ (a, e) A, A': x Δ ≫ T = T'.

assume ∀t. Fn@ (a, e, t) A, A': x Δ T, T': z ≫ σ if t ∈ T[a, e/x, Δ].

arb b, b' s.t. b = b' = a ∈ A.

show Fn@ (e) (Δ ⊢ T, T': z σ)[b; b'/x].

Fn@ (e) (Δ ⊢ T = T')[b; b'/x].

arb t s.t. t ∈ T[b, e/x, Δ].

enough by ind hyp to show Fn@ (e, t) (Δ T, T': z σ)[b; b'/x].

enough to show Fn@ (a, e, t) A, A': x Δ T, T': z σ.

enough to show t ∈ T[a, e/x, Δ].

enough to show T[a, e/x, Δ] = T[b, e/x, Δ].

QED by (A').

8.2 PwFn Facts

One fact useful in showing facts about PwFn is that

if ( ∀e. Fn@ (e) Δ ≫ σ0 if ∀i < k + 1. Fn@ (e) Δ ≫ σi+1 )
then PwFn Δ ≫ σ0 if ∀i < k + 1. PwFn Δ ≫ σi+1.

Also, the statements from section 3.1, pages 39–43 to which the dagger † is prefixed remain true when the property Fn σ is replaced by PwFn(m; σ), while those to which the double dagger ‡ is prefixed remain true when Fn is replaced by PwFn(m; σ) such that m ≤ |Δ|. We shall strengthen a number of those theorems, mostly ones with double dagger ‡ prefixes. Proofs will be given for the facts below that are labeled.

PwFn Δ ≫ .

(H) PwFn Δ ≫ T : z σ iff PwFn Δ T : z ≫ σ & PwFn Δ ≫ T type.
It follows (by induction on $|\Delta'|$) that

$$\text{PwFn } \Delta \Rightarrow \Delta' \Rightarrow \sigma \text{ iff PwFn } \Delta \Delta' \Rightarrow \sigma \& \text{ PwFn } \Delta \Rightarrow \Delta'.$$

(I) $\text{PwFn } \Delta \Delta' \Rightarrow \sigma[t; t'/z] \text{ if free } \sigma[t; t'/z]$

& $z$ is not free in $\Delta'$

& $z$ is not among $\Delta'$

& $\text{PwFn } \Delta \Rightarrow t = t' \in T$

& $\text{PwFn } \Delta \ T : z \Delta' \Rightarrow \sigma$.\(^5\)

(I') $\text{PwFn } \Delta \Delta' \Rightarrow \sigma$ if $z$ not free in $\Delta' \Rightarrow \sigma$

& $\text{PwFn } \Delta \Rightarrow \exists\in T$

& $\text{PwFn } \Delta \ T : z \Delta' \Rightarrow \sigma$.\(^6\)

(J) $\text{PwFn } \Delta \Delta' \Rightarrow \sigma$ if $\text{PwFn } \Delta \Delta'' \Rightarrow \sigma \& \Delta'$ not free in $\Delta'' \Rightarrow \sigma$.\(^7\)

(K) $\text{PwFn } \Delta A : x B : y \Delta' \Rightarrow \sigma$ if $\text{PwFn } \Delta \Delta'' \Rightarrow \sigma$

& $x$ not free in $B$

& $y$ not free in $A$

& $(x$ is not $y)$ or $(x, y$ not free in $\Delta' \Rightarrow \sigma)$.\(^8\)

One direction of (H) follows from

$\text{PwFn } \Delta \ T : z \Rightarrow \sigma$ if $\text{PwFn } \Delta \Rightarrow T : z \sigma$.

Proof:

arb $\Delta, T, z, \sigma$ s.t. $\text{PwFn } \Delta \Rightarrow T : z \sigma$.

arb $\bar{e}, t$ s.t. $\text{Fn@}(\bar{e}, t) \Delta \ T : z \Rightarrow$.

show $\text{Fn@}(\bar{e}, t) \Delta \ T : z \Rightarrow \sigma$.

$\text{Fn@}(\bar{e}) \Delta$.

$\text{Fn@}(\bar{e}) \Delta \Rightarrow T : z \sigma$.

enough by (G) to show $t \in T[\bar{e} / \bar{\Delta}]$.

QED by (C).

To show (H) it is enough to show that

\(^{5}\text{See page 42.}\)

\(^{6}\text{See page 42.}\)

\(^{7}\text{See page 42.}\)

\(^{8}\text{See page 43.}\)
8.2 PwFn Facts

\[ \text{PwFn } \Delta \triangleright T : z \sigma \text{ if PwFn } \Delta T : z \triangleright \sigma \land \text{PwFn } \Delta \triangleright T \text{ type.} \]

**Proof:**

\begin{itemize}
  \item arb \( \Delta, T, z, \sigma : \)
  \item assume PwFn \( \Delta T : z \triangleright \sigma \).
  \item assume PwFn \( \Delta \triangleright T \) type.
  \item arb \( \bar{e} \) s.t. \( \text{Fn@}(\bar{e}) \Delta \triangleright \).
  \item show \( \text{Fn@}(\bar{e}) \Delta T : z \sigma \).
  \item \( \text{Fn@}(\bar{e}) \Delta \triangleright T \) type.
  \item arb \( t \) s.t. \( t \in T[\bar{e}/\tilde{\Delta}] \).
  \item enough by (G) to show \( \text{Fn@}(\bar{e}, t) \Delta T : z \triangleright \sigma \).
  \item enough to show \( \text{Fn@}(\bar{e}, t) \Delta T : z \triangleright \).
\end{itemize}

**QED** by (C).

Now we prove (I).

**Proof:**

\begin{itemize}
  \item arb \( \Delta, \Delta', \sigma, T, z, t, t' : \)
  \item assume free \( \sigma[t; t'/z] \).
  \item assume \( z \) not free in \( \Delta' \).
  \item assume \( z \) not among \( \tilde{\Delta}' \).
  \item assume PwFn \( \Delta \triangleright t = t' \in T \).
  \item assume PwFn \( \Delta T : z \Delta' \triangleright \sigma \).
  \item arb \( \bar{e}, \bar{e}' \) s.t. \( |\bar{e}| \text{ is } |\Delta| \land |\bar{e}'| \text{ is } |\Delta'| \).
  \item assume \( \text{Fn@}(\bar{e}, \bar{e}') \Delta \Delta' \).
  \item show \( \text{Fn@}(\bar{e}, \bar{e}') \Delta \Delta' \triangleright \sigma[t; t'/z] \).
  \item \( \text{Fn@}(\bar{e}) \Delta \).
  \item \( \text{Fn@}(\bar{e}) \Delta \triangleright t = t' \in T \).
  \item let \( r \) be \( t[\bar{e}/\tilde{\Delta}] \).
  \item enough by (E) to show \( \text{Fn@}(\bar{e}, r, \bar{e}') \Delta T : z \Delta' \triangleright \sigma \) since \( (\Delta' \triangleright \sigma)[t; t'/z] \) is \( \Delta' \triangleright \sigma[t; t'/z] \).
  \item enough to show \( \text{Fn@}(\bar{e}, r, \bar{e}') \Delta T : z \Delta' \).
  \item \( \text{Fn@}(\bar{e}) \Delta \triangleright T \) type since \( \text{Fn@}(\bar{e}) \Delta \triangleright t = t' \in T \).
  \item \( r \in T[\bar{e}/\tilde{\Delta}] \) by (A).\
\end{itemize}
\[ \text{Fn@}(\bar{e}, r) \succeq T : z \quad \text{by (C).} \]

**enough** by (F) to show \( \text{Fn@}(\bar{e}, \bar{e}') \succeq \Delta'. \)

**QED**.

The proof of (I') is similar.

The proof of (J) proceeds easily by induction on \(|\Delta'|\) from

\[ \text{PwFn} \Delta T : z \Delta'' \gg \sigma \quad \text{if} \quad \text{PwFn} \Delta \Delta'' \gg \sigma \& \ z \text{ not free in } \Delta'' \gg \sigma. \]

**Proof**:

arb \( \Delta, T, z, \Delta'', \sigma : \)

assume \( \text{PwFn} \Delta \Delta'' \gg \sigma. \)

assume \( z \) not free in \( \Delta'' \gg \sigma. \)

arb \( \bar{e}, t, \bar{e}'' \) s.t. \( \bar{e} \) is \( |\Delta| \& |\bar{e}''| \) is \( |\Delta''|. \)

assume \( \text{Fn@}(\bar{e}, t, \bar{e}'') \succeq T : z \Delta''. \)

show \( \text{Fn@}(\bar{e}, t, \bar{e}'') \succeq T : z \Delta'' \gg \sigma. \)

**enough** by (F) to show \( \text{Fn@}(\bar{e}, \bar{e}'') \succeq \Delta'' \gg \sigma. \)

**enough to show** \( \text{Fn@}(\bar{e}, \bar{e}'') \succeq \Delta'' \).

**QED** by (F) since \( t \in T[\bar{e}/\bar{\Delta}]. \)

The proof of (K) proceeds easily from

\[ \text{Fn@}(\bar{e}, a, b, \bar{e}') \succeq A, A' : x B, B' : y \Delta' \gg \sigma \]

if \( \text{Fn@}(\bar{e}, a, b, \bar{e}') \succeq B, B' : y \ A, A' : x \Delta' \gg \sigma \)

& \( |\bar{e}| \) is \( |\Delta| \& |\bar{e}| \) is \( |\Delta| \)

& \( x \) not free in \( B, B' \)

& \( y \) not free in \( A, A' \)

& \( (x \) is not \( y \) or \( (x, y \) not free in \( \sigma \)), \)

which may be proved by induction on \( \bar{e}. \) The induction step is easy, so to show (K) it is enough to show the base case.

**Proof**:

arb \( A, A', x, B, B', y, \Delta', \sigma : \)

assume \( \text{Fn@}(b, a, \bar{e}') \succeq B, B' : y A, A' : x \Delta' \gg \sigma. \)

assume \( x \) not free in \( B, B' \& y \) not free in \( A, A'. \)

assume \( (x \) is not \( y \) or \( (x, y \) not free in \( \Delta' \gg \sigma). \)

show \( \text{Fn@}(a, b, \bar{e}') \succeq A, A' : x B, B' : y \Delta' \gg \sigma. \)

\( a \in A \& b \in B. \)
\textbf{8.3 PwF}n and Respect

\textbf{arb} \, r, r', s, s' \text{ s.t.} \, r = r' = a \in A \land s = s' = b \in B. \\
\textbf{enough to show} \, \text{Fn@}(\bar{e}') (\Delta' \gg \sigma)[r, s; r', s'/x, y]. \\
\text{Fn@}(\bar{e}') (\Delta' \gg \sigma)[s, r; s', r'/y, x]. \\
\textstyle(\Delta' \gg \sigma)[r, s; r', s'/x, y] \text{ is } (\Delta' \gg \sigma)[s, r; s', r'/y, x]. \\
\textbf{QED}.

\textbf{8.3 PwF}n and Respect

The results in chapter 5 about Fn that are prefixed with a dagger \dag or a double dagger \ddag carry over to Fn@ and PwF}n in the way already described.

\textbf{8.4 Inductive Under PwF}n

Inductive elimination schemes using PwF}n can be made more expressive, in a sense, than the analogous schemes using Fn. We shall use N elimination as our paradigm.

Part of the work in showing that \text{Fn } \Delta \triangleright N : x \land t = t' \in T \text{ is to show that } \text{Fn } \Delta \triangleright N : x \land T \text{ type. But no such work is done in the example on page 70 of section 5.5 since from the premise that } \text{Fn } \Delta \triangleright N : x \land T : y \land s = s' \in T[\text{suc}(x)/x] \text{ it follows immediately that } \text{Fn } \Delta \triangleright N : x \land T \text{ type. Establishing the } T \text{ functionality over } N \text{ of some term by the means provided in that example requires that type functionality of } T \text{ over } N \text{ already be established.}

Using PwF}n, however, sometimes permits simultaneous demonstration, by induction over \text{N}, of the \text{T} functionality of a term and the type functionality of \text{T}.

\text{PwF}n \Delta \gg N : x \land t = t' \in T \land \\
\text{if } z \text{ is not } y \land y \text{ is not free in } T \land \\
\text{& } \text{PwF}n \Delta \gg t[0/x] = t'[0/x] \in T[0/x] \land \\
\text{& } \text{PwF}n \Delta N : x \land T : y \gg s = s' \in T[\text{suc}(x)/x] \land \\
\text{& } \forall d, e \text{. if } d \leftrightarrow e \text{ then } T[d/x] \simeq \text{type}^\circ T[e/x] \land \\
\text{& } t[d/x] \simeq \text{mem}^\circ t[e/x] \land \\
\text{& } t'[d/x] \simeq \text{mem}^\circ t'[e/x] \land \\
\text{& } t[\text{suc}(x)/x] \simeq \text{mem}^\circ s[t/y] \land \text{free } s[t/y] \land \\
\text{& } t'[\text{suc}(x)/x] \simeq \text{mem}^\circ s'[t'/y] \land \text{free } s'[t'/y].

\textbf{Proof}:

\textbf{arb} \, \Delta, t, t', T, x, y, s, s' \text{ s.t. the antecedent holds.}

\textbf{arb} \, \bar{e} \text{ s.t. } \text{Fn@}(\bar{e}) \Delta \gg .
show $\text{Fn@}(\varepsilon) \Delta \gg N : x \quad t = t' \in T$.

$\text{Fn@}(\varepsilon) \Delta \gg N$ type since $N$ type.

enough by (G)
to show $\forall b. \text{ if } b \in N \text{ then } \text{Fn@}(\varepsilon, b) \Delta N : x \gg t = t' \in T$.

First, we shall prove a lemma.

show the free variables of $t, t', T$ are among $\tilde{\Delta}, x$.

$\text{Fn@}(\varepsilon) \Delta \gg t[0/x] = t'[0/x] \in T[0/x]$.

$\text{Fn} \ (t = t' \in T)[\varepsilon, 0/\tilde{\Delta}, x]$ by (A).

$(t = t' \in T)[\varepsilon, 0/\tilde{\Delta}, x]$ is closed.

qed.

We shall proceed by induction on $N$ membership.

base case :

arb $b$ s.t. $0 \leftarrow b$.

show $\text{Fn@}(\varepsilon, b) \Delta N : x \gg t = t' \in T$.

arb $q, q'$ s.t. $q = q' = b \in N$.

enough by (F) to show $\text{Fn@}(\varepsilon) \Delta \gg (t = t' \in T)[q; q'/x]$.

$0 \leftarrow q, q'$.

$T[0/x] \simeq \text{type}^0 T[q/x]$.

$T[0/x] \simeq \text{type}^0 T[q'/x]$.

$t[0/x] \simeq \text{mem}^0 t[q/x]$.

$t'[0/x] \simeq \text{mem}^0 t'[q'/x]$.

enough to show $\text{Fn@}(\varepsilon) \Delta \gg (t = t' \in T)[0/x]$.

enough to show $\text{Fn@}(\varepsilon) \Delta$.

qed.

arb $a, b$ s.t. $\text{suc}(a) \leftarrow b \& \ a \in N$.

assume $\text{Fn@}(\varepsilon, a) \Delta N : x \gg t = t' \in T$.

enough to show $\text{Fn@}(\varepsilon, b) \Delta N : x \gg t = t' \in T$.

$b \in N$.

arb $q, q'$ s.t. $q = q' = b \in N$.

enough by (F) to show $\text{Fn@}(\varepsilon) \Delta \gg (t = t' \in T)[q; q'/x]$.

arb $p, p'$ s.t. $\text{suc}(p) \leftarrow q \& \text{suc}(p') \leftarrow q' \& \ p = p' \in N$. 
\[ T[\text{suc}(p)/x] \simeq \text{type}^0 T[q/x]. \]
\[ T[\text{suc}(p')/x] \simeq \text{type}^0 T[q'/x]. \]
\[ t[\text{suc}(p)/x] \simeq \text{mem}^0 t[q/x]. \]
\[ t'[\text{suc}(p')/x] \simeq \text{mem}^0 t'[q'/x]. \]

**enough to show** \( \text{Fn}\bar@\bar(\bar e) \Delta \gg (t = t' \in T)[\text{suc}(p); \text{suc}(p')/x]. \)
\[ t[\text{suc}(x)/x][p/x] \simeq \text{mem}^0 s[t/y][p/x]. \]
\[ t'[\text{suc}(x)/x][p'/x] \simeq \text{mem}^0 s'[t'/y][p'/x]. \]

**enough to show** \( \text{Fn}\bar@\bar(\bar e) \Delta \gg (s[t/y] = s'[t'/y] \in T[\text{suc}(x)/x])[p;p'/x]. \)

**enough by** (F)
**to show** \( \text{Fn}\bar@\bar(\bar e, p) \Delta N:x \gg s[t/y] = s'[t'/y] \in T[\text{suc}(x)/x]. \)

Prepare to use (E).
\( \text{Fn}\bar@\bar(\bar e, p) \Delta N:x \gg t = t' \in T \) by (D) since \( a = p \in N. \)

**free** \( (s = s' \in T[\text{suc}(x)/x])[t; t'/y]. \)

**enough by** (E)
**to show** \( \text{Fn}\bar@\bar(\bar e, p, t[\bar e, p/\bar \Delta, x]) \Delta N:x \quad T:y \gg s = s' \in T[\text{suc}(x)/x]. \)

**enough to show** \( \text{Fn}\bar@\bar(\bar e, p, t[\bar e, p/\bar \Delta, x]) \Delta N:x \quad T:y \gg . \)
\[ t[\bar e, p/\bar \Delta, x] \in T[\bar e, p/\bar \Delta, x] \) by (A).

**QED by** (C).

The scheme is simpler if our aim is merely to show inhabitation.

\[ \text{PwFn } \Delta \gg N:x \ \exists \in T \]
if \( x \) is not \( y \) & \( y \) is not free in \( T \)
\& \( \text{PwFn } \Delta \gg \exists \in T[0/x] \)
\& \( \text{PwFn } \Delta N:x \quad T:y \gg \exists \in T[\text{suc}(x)/x] \)
\& \( \forall d,e. \) if \( d - e \) then \( T[d/x] \simeq \text{type}^0 T[e/x]. \)

The proof is roughly similar to the previous one, but much simpler.

### 8.5 PwFn and Type System Hierarchies - HPwFn

What is an appropriate analog to HFn for use with type system hierarchies (chapter 4)? The relation \( \exists n. \text{PwFn}_{\mathbf{T}_n}(m; \sigma) \) is not suitable since for certain sequents PwFn could hold at one level of the hierarchy and yet at no higher
level.\(^9\) That is, $\text{PwFn}_{T_n}(m; \sigma)$ is not monotonic in $n$. For example, if $T_n$ is a reflective hierarchy with universes $V_i$, and $A$ has no value, then

$$\text{PwFn}_{T_n}(A : x \gg A \text{ type iff } n = 0).$$

To damp out this non-monotonicity we may use the relation $\text{HPwFn}$ such that

$$\text{HPwFn}(n.T_n ; m ; \sigma) \iff \exists k. \forall n \geq k. \text{PwFn}_{T_n}(m; \sigma).$$

Let $\text{HPwFn}(n. T_n) \Delta \gg \sigma$ be $\text{HPwFn}(n.T_n ; | \Delta | ; \Delta \gg \sigma )$.

It is useful when trying to carry properties of PwFn over to HPwFn that

if ( $\exists k. \forall n \geq k. \text{PwFn}_{T_n}(m_0; \sigma_0)$ if $\forall i < k. \text{PwFn}_{T_n}(m_{i+1}; \sigma_{i+1})$ )
then $\text{HPwFn}(n.T_n ; m_0 ; \sigma_0)$ if $\forall i < k. \text{HPwFn}(n.T_n ; m_{i+1}; \sigma_{i+1})$.

Also, all the statements from chapter 4 about HFn that are prefixed with a dagger $\dagger$ remain true when the relation $\text{HFn}(n.T_n ; \sigma)$ is replaced by the relation $\text{HPwFn}(n.T_n ; m ; \sigma)$, while those prefixed by a double dagger $\ddagger$ remain true when the relation $\text{HFn}(n.T_n ; \sigma)$ is replaced by $\text{HPwFn}(n.T_n ; m ; \sigma)$ such that $m \leq | \Delta |$.

I do not know of another informative characterization of the equivalence of $\text{HPwFn}(n.T_n)$ and $\text{PwFn}_{\bigcup n T_n}$ for a reflective hierarchy $T_n$, but a sufficient condition is that

$$\forall \sigma. \exists n. \forall \bar{e}. \text{Fn}_{T_n}(\bar{e}) \sigma \text{ if } \text{Fn}_{\bigcup n T_n}(\bar{e}) \sigma.$$

The results concerning universe polymorphism apply to HPwFn because it has the form $\exists k. \forall n \geq k. \Phi(T_n)$.

---

\(^9\)To make explicit the parameterization of PwFn by type systems we use a subscript for the type system argument.
Bibliography

[Aczel 80]
Peter Aczel.
Frege structures and the notions of proposition, truth, and set.
*The Kleene Symposium*,

[Aczel 82]
Peter Aczel.
The type theoretic interpretation of constructive
set theory: choice principles.
*The L. E. J. Brouwer Centenary Symposium*,
A. S. Troelstra and D. van Dalen, eds.

[Allen 87]
Stuart Allen.
A non-type-theoretic definition of Martin-Löf’s types.
*Proceedings of 1987 IEEE Symposium on Logic in Computer Science.*
Computer Society Press of the IEEE, Washington, DC, 1987,
pages 215–221.
(also available as Tech. Report, TR 87–832, C.S. Dept., Cornell)

[Bates 79]
Joseph L. Bates.
*A Logic for Correct Program Development.*
Doctoral Dissertation, Computer Science Department
Cornell University, 1979.

[Bates & Constable 81]
The Definition of $\mu$-PRL.
Computer Science Department Technical Report,
TR 82–492.
Cornell University, Ithaca, NY, 1981.
[Bates & Constable 83]
The Nearly Ultimate PRL.
Computer Science Department Technical Report,
TR 83-551.
Cornell University, Ithaca, NY, 1983.

[Beeson 82]
M. J. Beeson.
Recursive models for constructive set theories.

[Beeson 85]
M. J. Beeson.
*Foundations of Constructive Mathematics.*

[Bishop 67]
Errett Bishop.
*Foundations of Constructive Analysis.*

[Bishop 70]
Errett Bishop.
Mathematics as a numerical language.
*Intuitionism and Proof Theory,*
J. Myhill, et al., eds.

[de Bruijn 68a]
N. G. de Bruijn.
AUTOMATH, a Language for Mathematics.
T.H.-Report 68-WSK-05,
Eindhoven University of Technology, 1968.

[de Bruijn 68b]
N. G. de Bruijn.
The mathematical language of AUTOMATH,
it's usage and some of its extensions.
*Symposium on Automatic Demonstration, 1968*  
[de Bruijn 80]
N. G. de Bruijn.
A survey of the project AUTOMATH.
*Essays in Combinatory Logic, Lambda Calculus, and Formalism,*
J. P. Seldin and J. R. Hindley, eds.

[Constable 71]
Robert L. Constable.
Constructive mathematics and automatic programs writers.
*Proceedings of IFIP Congress,* Ljubljana,

[Constable 82]
Robert L. Constable.
Intensional Analysis of Functions and Types.
Computer Science Department Technical Report,
CSR 118–82.
University of Edinburgh, 1982.

[Constable 83a]
Robert L. Constable.
Constructive mathematics as a programming logic I:
some principles of theory.

[Constable 83b]
Robert L. Constable.
Mathematics as Programming.
*Proceedings of Workshop on Logics of Programs,*
*Lecture Notes in Computer Science,* Vol. 164.

[Constable 83c]
Robert L. Constable.
Partial functions in constructive formal theories.
*Proceedings of 6th G.I. Conference,*
*Lecture Notes in Computer Science,* Vol. 45.
Springer-Verlag, New York, 1983.
(also available as Tech. Report, TR 83-549, C.S. Dept., Cornell)

[Constable et al. 86]
Robert L. Constable, et al.
Implementing Mathematics with the Nuprl Proof Development System.

[Constable, Johnson, & Eichenlaub 82]
Introduction to the PL/CV2 Programming Logic,
Lecture Notes in Computer Science, Vol. 135.

[Constable, Knoblock, & Bates 85]
Robert L. Constable, Todd B. Knoblock, and Joseph L. Bates.
Writing programs that construct proofs.
J. Automated Reasoning, Vol. 1, n. 3.

[Constable & Mendler 85]
Robert L. Constable and N. P. Mendler.
Recursive definitions in type theory.
Proceedings of Logics of Programs Conference,
Lecture Notes in Computer Science, Vol. 193.

[Constable & O'Donnell 78]
Robert L. Constable and Michael J. O'Donnell.
A Programming Logic.

[Constable & Smith 87]
Robert L. Constable and Scott. F. Smith.
Partial objects in constructive type theory.
Computer Society Press of the IEEE, Washington, DC, 1987,
pages 183–193.
(also available as Tech. Report, TR 87–822, C.S. Dept., Cornell)

[Constable & Zlatin 84]
Robert L. Constable and Daniel R. Zlatin.
The type theory of PL/CV3.
ACM Transactions on Programming Languages and Systems,

[Conway & Gries 75]
R. Conway & D. Gries.
[Curry, Feys, & Craig 58]
H. B. Curry, R. Feys, and W. Craig.

[Dijkstra 76]
Edsger W. Dijkstra.
A Discipline of Programming.

[Girard 72]
J. Y. Girard.
Interprétation fonctionelle et élimination des
coupures dans l'arithmétique d'ordre supérieur.
Thèse de Doctorat d'Etat,
University of Paris, France, 1972

[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.

[HAN] see Martin-Löf.

[Harper 85]
Robert W. Harper.
Aspects of the Implementation of Type Theory.
Doctoral Dissertation, Computer Science Department,
Cornell University, June 1985.
(also available as Tech. Report, TR 85-675, C.S. Dept., Cornell)

[Howard 80]
W. Howard.
The formulas-as-types notion of construction.
To H. B. Curry: Essays on Combinatory Logic,
Lambda-Calculus, and Formalism,
J. P. Seldin and J. R. Hindley, eds.

[Knoblock 87]
Todd B. Knoblock.
Metamathematical Extensibility in Type Theory.
Doctoral Dissertation, Computer Science Department
Cornell University, 1987 (expected).
[Knoblock & Constable 86]
Todd B. Knoblock and Robert L. Constable.
Formalized metareasoning in type theory.
Computer Society Press of the IEEE, Washington, DC, 1986,
pages 237–248.
(also available as Tech. Report, TR 86–742, C.S. Dept., Cornell)

[MacQueen 86]
David MacQueen.
Using dependent types to express modular structure.
Proceedings of 13th ACM Symposium on
Principles of Programming Languages.

[Martin-Löf 72]
Per Martin-Löf.
An intuitionistic theory of types.
Mimeographed manuscript (apparently a draft of Martin-Löf 73).
1972.

[Martin-Löf 73]
Per Martin-Löf.
An intuitionistic theory of types: predicative part.
Logic Colloquium '73,
H. E. Rose and J. C. Shepherdson, eds.
North–Holland, Amsterdam, 1975, pages 73–118.

[Martin-Löf 75]
Per Martin-Löf.
Syntax and semantics of mathematical language.
Notes written by Peter Hancock on half of a lecture given
by Per Martin-Löf at Oxford University in Michaelmas Term 1975.

[HAN]
Per Martin-Löf.
Constructive mathematics and computer programming.
Proceedings of Sixth International Congress for
Logic, Methodology, and Philosophy of Science.

[Padova]
Per Martin-Löf.
Intuitionistic type theory.

[Martin-Löf 83]
Per Martin-Löf.
On the Meanings of the Logical Constants and the Justifications of the Logical Laws.
Scuola di Specializzazione in Logica Matematica,
Università di Siena, 1985.

[Mendler 87]
N. P. Mendler.
Inductive Definition in Constructive Type Theory.
Doctoral Dissertation, Computer Science Department
Cornell University, 1987 (expected).

[Mendler, Panangaden, & Constable 86]
Infinite objects in type theory.
Computer Society Press of the IEEE, Washington, DC, 1986,
pages 249–257.

[Mitchell & Plotkin 85]
John Mitchell and Gordon Plotkin.
Abstract types have existential type.
Proceedings of 12th ACM Symposium on
Principles of Programming Languages.

[Nordström, Petersson, & Smith]
B. Nordström, K. Petersson, and J. Smith.
Introduction to Type Theory.
(expected)

[Padova] see Martin-Löf.

[Petersson 82]
Kent Petersson.
A Programming System for Type Theory.
Department of Computer Science,
LPM Memo 21.
University of Göteborg, Sweden, 1982.
[Russell 08]
Bertrand Russell.
Mathematical logic as based on a theory of types.
1908, pages 222–262.

[Scott 70]
Dana Scott.
Constructive validity.
*Symposium on Automatic Demonstration*,

[Smith 84]
Jan Smith.
An interpretation of Martin-Löf’s type theory
in a type-free theory of propositions.
1984, pages 730–753.

[Stenlund 72]
Sören Stenlund.
*Combinators, λ-terms, and Proof Theory*.

[Wirth 73]
N. Wirth.
*Systematic Programming: An Introduction*.
Index

assumption, 7, 11
  positive versus negative, 15
  see also declaration

Beeson's models, 21

canonical form, 5, 27
change of variables, 42, 66
closed subterm variation (csx), 67
composition of sequents with terms, 42
computation system, 27
  see also evaluation system
consequent, 36
constituent type, 6
contractum, 67
correspondence,
  formal, 81
  nonstandard, 85
  standard, 81
  validity of, 82, 83, 85
strong, 81, 82
weak, 82, 85, 86
dagger † ‡, 40, 92, 101, 104
declaration, 36
  see also assumption
declaration lists, merging and permuting, 42
equality-complete type, 84
equality,
  see member equality
  see type equality
evaluation, 5, 27
evaluation system, 27, 28, 32
excluded middle, 86
expression, 5
  canonical, 5
  untyped, 2
  see also term
extensional type equality,
  see type equality
family of types, see types
finite types (FIN), 31, 77
Fn, 36, 37, 47
  and respect, 64
  definition of, 37
  facts about, 39
  HFn, 47
Fn@,
  definition of, 91
  facts about, 92
formal correspondence,
  see correspondence
free, 28, 37
Frege structures, 22, 50
functionality, 8, 35, 59
  see also Fn
  see also pointwise functionality
functions over universes, 11

HAN_n, 52
HAN_w, 53
hierarchy,
  see reflective hierarchy
incomplete type, 85
incomplete type (continued):

see also equality-complete type

see also member-complete type

induction, 70

half-positive, 49

on type formation, 53

under PwFn, 89, 101, 103

inference rule, 9, 18

general, 39

omitting premises, 68

see also natural deduction

intensional type equality,

see type equality

intuitionistic type theory, 2

judgements,

of HAN, 7, 35

of Nuprl, 13, 15, 17

lazy evaluation, 30

level expression, 56, 57

level variable, 57

reassignable, 59

logical type, 3

member equality, 6, 30, 40

member-complete type, 83

membership, 6

multi-type, 12

instance of, 12

N, 70, 101

natural deduction, 42

Nuprl, 2, 89

type theory of, 10

open-endedness, 6, 20, 27, 48, 73

adding new terms, 77

adding new types, 74

operator, 27

distinguished, 77

PL/CV, 1

plentiful variables, 67

pointwise functionality, 16, 19, 20, 89

see also PwFn

see also functionality

possible type system, 50, 52

presupposition, 7, 23

PRL (program refinement logic), 1

see also Nuprl

program specification, 2

programming, 4

proof (constructive), 1, 3

propositions, 4

and types, 1, 3, 6, 20, 81, 86

ramification, 6

see also correspondence

classical, 86

direct expression of, 86

PwFn, 40

and induction, 101

and respect, 101

definition of, 91

facts about, 97

HPwFn, 104

relation to Fn, 92

see also pointwise functionality

quotient type, 11, 14, 20, 83

unification with set type, 15

ramified theory, 6

reassignable level variable, 59

recursive realizability, 21

redex, 67

refinement, 90

see also PRL

reflection principle, 44
reflective hierarchy, 45, 46, 52, 53
HF'n, 47
HPwFn, 104
transfinite, 48
reinterpretation of type-theoretic concepts, 17, 21
respected relation, definition of, 63
sequent, 18, 35
complex, 36
simple, 35
set type, 11, 14, 20, 83
skeleton,
equality, 30, 45
evaluation system, 28
type system, 30
Smith's interpretation, 22
spine, 46, 52, 53
strong correspondence,
see correspondence
substitution, 27
parallel, 36
subtype, see set type
symmetry of equality, 40
term, 18, 27
canonical, 27
see also expression
transitivity of equality, 40
truth, 9
type (assignment) system, 3, 19, 27, 29
comparison with type definitions, 30
type constructor, 6
type equality, 30, 40
extensional, 8, 12, 30, 41
intensional, 11, 17
of Nuprl, 12
type formation, 50, 52
induction, 53
type formation (continued):
TyF, 52
type theory, 3, 5, 27
intuitionistic, 2, 3
of Nuprl, 10
types, 3, 4
defining, 2, 5, 22, 84
see also open-endedness
family of, 6, 11, 13
Fam(τ; A; α; x; B; γ), 51
logical, 3
use as objects, 44
universe polymorphism, 19, 55, 60, 76
and HPwFn, 104
relativity, 61, 76
UPoly, 56, 57
universe relativity, 61, 76
universes, 6, 9, 11, 19, 44, 46
functions over, 11
"Tarskian", 9, 22
value, 27
see also evaluation
variable, 27
distinguished, 78
weak correspondence,
see correspondence
# Symbol Index

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N_n)</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>(N)</td>
<td>6</td>
<td>FINbody</td>
</tr>
<tr>
<td>(I(A, a, b))</td>
<td>6</td>
<td>FIN</td>
</tr>
<tr>
<td>(A + B)</td>
<td>6</td>
<td>(\bullet)</td>
</tr>
<tr>
<td>((\Sigma x \in A)B)</td>
<td>6</td>
<td>(T = S)</td>
</tr>
<tr>
<td>((\Pi x \in A)B)</td>
<td>6</td>
<td>(\exists \in T = S)</td>
</tr>
<tr>
<td>((W x \in A)B)</td>
<td>6</td>
<td>(t = s \in T = S)</td>
</tr>
<tr>
<td>(U_n)</td>
<td>6</td>
<td>(A, B: x \sigma)</td>
</tr>
<tr>
<td>(T \text{ type } (x_1 \in A_1 \ldots x_n \in A_n))</td>
<td>7</td>
<td>(\exists \in T)</td>
</tr>
<tr>
<td>(T = S (x_1 \in A_1 \ldots x_n \in A_n))</td>
<td>7</td>
<td>(t = s \in T)</td>
</tr>
<tr>
<td>(t \in T (x_1 \in A_1 \ldots x_n \in A_n))</td>
<td>7</td>
<td>(t \in T)</td>
</tr>
<tr>
<td>(t = s \in T (x_1 \in A_1 \ldots x_n \in A_n))</td>
<td>7</td>
<td>(A: x \sigma)</td>
</tr>
<tr>
<td>{x \in A</td>
<td>B _z}</td>
<td>14</td>
</tr>
<tr>
<td>(x, y \in A/ E_{x,y})</td>
<td>14</td>
<td>(\Delta)</td>
</tr>
<tr>
<td>(x, y \in A/ E_{x,y})</td>
<td>15</td>
<td>(\Delta)</td>
</tr>
<tr>
<td>(x_1 \in A_1 \ldots x_n \in A_n \gg t \in T)</td>
<td>15</td>
<td>(\sigma[a; b/\vec{x}])</td>
</tr>
<tr>
<td>(x_1: A_1 \ldots x_n: A_n \gg T \text{ ext } t)</td>
<td>15</td>
<td>(\sigma[a/\vec{x}])</td>
</tr>
<tr>
<td>(t \leftarrow s)</td>
<td>27</td>
<td>(\text{free } \sigma[a; b/\vec{x}])</td>
</tr>
<tr>
<td>(t[s_1 \ldots s_n/x_1 \ldots x_n])</td>
<td>28</td>
<td>(\text{free } \sigma[a/\vec{x}])</td>
</tr>
<tr>
<td>(</td>
<td>s</td>
<td>)</td>
</tr>
<tr>
<td>(\text{free } t[s/\vec{x}])</td>
<td>28</td>
<td>(t = s \in T = S)</td>
</tr>
<tr>
<td>(\text{var}(t))</td>
<td>28</td>
<td>(t = s \in T)</td>
</tr>
<tr>
<td>(\text{form}(t; k; \vec{x}; \vec{e}))</td>
<td>28</td>
<td>(t \in T)</td>
</tr>
<tr>
<td>#\text{subt}(k)</td>
<td>28</td>
<td>(\uparrow, \downarrow)</td>
</tr>
<tr>
<td>#\text{bv}(k, i)</td>
<td>28</td>
<td>(A \models B)</td>
</tr>
<tr>
<td>(T = S)</td>
<td>29</td>
<td>a = b \in _T A)</td>
</tr>
<tr>
<td>(t = s \in T)</td>
<td>29</td>
<td>(A \text{ type}_T)</td>
</tr>
<tr>
<td>(T \text{ type})</td>
<td>30</td>
<td>(\text{T ext S})</td>
</tr>
<tr>
<td>(t \in T)</td>
<td>30</td>
<td>(\perp)</td>
</tr>
<tr>
<td>(\exists \in T)</td>
<td>30</td>
<td>(_T \cup _S)</td>
</tr>
<tr>
<td>(r \leftarrow t, s)</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

116
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$ reflect $T$</td>
<td>46</td>
</tr>
<tr>
<td>$(T.S(T))^{*}(i.V_i)_n$</td>
<td>46</td>
</tr>
<tr>
<td>spine$_n$</td>
<td>46</td>
</tr>
<tr>
<td>$(T.S(T); R)^{*}(i.V_i)_n$</td>
<td>46</td>
</tr>
<tr>
<td>$F_nT$</td>
<td>47</td>
</tr>
<tr>
<td>$HF_n(n.T_n)$</td>
<td>47</td>
</tr>
<tr>
<td>$\cup_n T_n$</td>
<td>48</td>
</tr>
<tr>
<td>$\hat{N}_\tau T\phi$</td>
<td>50</td>
</tr>
<tr>
<td>$\hat{N}T\phi$</td>
<td>51</td>
</tr>
<tr>
<td>$\hat{I}(\tau)T\phi$</td>
<td>51</td>
</tr>
<tr>
<td>$\hat{\tau}(\tau)T\phi$</td>
<td>51</td>
</tr>
<tr>
<td>$\text{Fam}(\tau; A; \alpha; x; B; \gamma)$</td>
<td>51</td>
</tr>
<tr>
<td>$\hat{\Sigma}(\tau)T\phi$</td>
<td>51</td>
</tr>
<tr>
<td>$\hat{\Pi}(\tau)T\phi$</td>
<td>51</td>
</tr>
<tr>
<td>$\hat{W}(\tau)T\phi$</td>
<td>51</td>
</tr>
<tr>
<td>$TyF(\sigma; \tau)T\phi$</td>
<td>52</td>
</tr>
<tr>
<td>$CTyF(\sigma; \tau)$</td>
<td>52</td>
</tr>
<tr>
<td>$\mu(\sigma)T\phi$</td>
<td>52</td>
</tr>
<tr>
<td>$HAN_n T\phi$</td>
<td>52</td>
</tr>
<tr>
<td>spine$_\alpha T\phi$</td>
<td>52</td>
</tr>
<tr>
<td>HAN$_\omega T\phi$</td>
<td>53</td>
</tr>
<tr>
<td>spine$_\omega T\phi$</td>
<td>53</td>
</tr>
<tr>
<td>$t \text{ var } s$</td>
<td>53</td>
</tr>
<tr>
<td>$\text{VAR}_n T\phi$</td>
<td>53</td>
</tr>
<tr>
<td>$[\alpha_0 \ldots \alpha_k]$</td>
<td>57</td>
</tr>
<tr>
<td>$\alpha[n]$</td>
<td>57</td>
</tr>
<tr>
<td>$(\alpha)_f$</td>
<td>57</td>
</tr>
<tr>
<td>$\text{UPoly}(i.V_i; U.\Psi(\alpha.U_\alpha))$</td>
<td>57</td>
</tr>
<tr>
<td>$\Delta - T$</td>
<td>60</td>
</tr>
<tr>
<td>$(\lambda \Delta)t$</td>
<td>60</td>
</tr>
<tr>
<td>$T \simeq \text{type } S$</td>
<td>63</td>
</tr>
<tr>
<td>$t \simeq \text{mem } s$</td>
<td>63</td>
</tr>
<tr>
<td>$t \simeq \text{opt } s$</td>
<td>64</td>
</tr>
<tr>
<td>$t R^o s$</td>
<td>64</td>
</tr>
<tr>
<td>$t R \circ S s$</td>
<td>66</td>
</tr>
<tr>
<td>$tR^* s$</td>
<td>66</td>
</tr>
<tr>
<td>$t \alpha s$</td>
<td>67</td>
</tr>
<tr>
<td>$t R^\alpha s$</td>
<td>67</td>
</tr>
</tbody>
</table>