CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30. Please read the authorization forms which accompany this thesis.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

A Program Verifier for Total Correctness Based on Intuitionistic Type Theory

Michel Patrick Devine

A Thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

August, 1984

ABSTRACT

A Program Verifier for Total Correctness
based on
Intuitionistic Type Theory

Michel Patrick Devine

A program verifier based on a variant of Intuitionistic Type Theory was developed to prove total correctness of applicative programs. The functional language Brouwer which was devised specifically for ease of verification includes such features as modules, high-order functions, polymorphic data types and abstract specifications. Verification is achieved through the use of proof rules expressed as VML (Verification Meta-Language) programs. Executable code for a traditional Von Neumann computer is obtained from type theoretical expressions by source-to-source transformation.

To Ann, who made it all possible.

Acknowledgments

I wish to thank Professor Hendrik Boom for his supervision of this project. His extensive knowledge of this topic provided the inspiration for many of the ideas.

Thanks to those who proof-read the thesis and suggested needed clarifications: Franco Carlacci, John Opala and Rene Hollan. In particular, I would like to thank Alan Madras and Stephen Spackman for their support, encouragement and tangible help far beyond the call of duty.

I am very grateful to my family for their patience and understanding during the past two years.

I am indebted to the Natural Sciences and Engineering Research Council of Canada for their generous financial support.

CONTENTS

Chapter 1

Introduction

The problem of correctness in computer science is one which has been extensively studied during the last few decades. Since 1959, many efforts have been made by computer scientists to describe the meaning (or semantics) of computer programs. However, the field of program specification and verification has so far seen relatively few major breakthroughs, as compared with other computer-related fields such as analysis of algorithms, computational complexity and compiler construction.

Why has correctness of programs lagged behind other disciplines? The answer lies in the theoretical complexity of the question of correctness itself.

Traditionally, programmers use 'debugging' to ascertain that their programs are working correctly. Having written a program, the programmer tries it out on a set of test cases which he believes will be sufficient to find all possible mistakes. Obviously, the validity of this approach depends heavily on the programmer's choice of test cases and his ability to detect mistakes. It is thus extremely error-prone, since the agent performing the testing has a

biased view of the validity of his/her work.

Large software producers like N.A.S.A. have hierarchical debugging systems involving large numbers of people. A program's specification is written by a team which passes on its requirements to a programming team. When the programming team has coded the program, a new team of people reads the code and attempts to find discrepancies between the specification and the implementation. Once past this stage, the program is sent to the debugging team, which tests it. If any problems are detected, a different team of programmers is called in to fix them, thereby ensuring a fresh outlook, and an entirely new reading and debugging team is used for the next stage. This process is repeated until no further problems are found.

With such an extensive (and expensive) debugging system, it is reasonable to expect the produced program to behave correctly. Unfortunately, this is not the case, as the problems encountered during the warm-up phase of some Space Shuttle launches have demonstrated [Gar81].

An automated program verifier can be described as a computer program whose input consists of two items of data: a specification and a program. The verifier attempts to prove mathematically that the program satisfies its supplied specification. If this can be done, no testing is required

since it has been shown that the program is necessarily correct for all possible sets of input.

This thesis describes the implementation of a program verifier based on constructive logic. The verifier checks for total correctness, i.e., that the program satisfies its specification and that it terminates. The language Brouwer has been developed to enable the formulation of theorems and proofs (specifications and programs) using programming language notation rather than pure logic. Brouwer is translated into logic for the purposes of verification. When the logical translation of the program is verified, it is further translated into a popular programming language (LISP) which may then be compiled. Thus it is possible (and practical) to compile logical formulae and their proofs. It is not suggested that logic be used to express programs (as in Logic Programming); rather, the logical expressions themselves can be seen as programs and consequently compiled.

Chapter 2 defines the correctness problem in computer science, describing briefly the existing verification methods and verification systems. The mathematical foundations for the Brouwer verifier are given in chapter 3, while chapter 4 consists of a description of the specification and implementation language used in our approach (Brouwer). The verification method is detailed in

chapter 5, along with an overview of a complete Brouwer verification system.

The source-to-source transformation scheme used to produce executable code from the implementation's proof is given in chapter 6. Finally, chapter 7 concludes the thesis and describes future work on the Brouwer verification system.

Chapter 2

The Problem of Correctness in Computer Science

This chapter presents a definition of the problem of program correctness and describes briefly the principal existing specification and verification techniques.

2.1 Definition of Specification and Verification

When programming a computer to perform a task, a formal notation is used to describe an algorithm. The problem to be solved can be thought of as a function from input to output values described by a specification. The specification typically includes constraints on input and output data as well as propositional rules detailing the desired behaviour of the solution (algorithm). This solution is called the implementation.

A specification can be weak, strong or complete. For example, a weak specification could be that program P accepts an integer value and returns an integer. This is a weak specification since it does not define how the result is to be derived from the input. Consequently, there is a multitude of programs which would exhibit this behaviour, e.g. the identity function, increment and decrement

functions, etc. Most current compilers include rules for verification of weak specifications which are inherent in the semantics of the source language. These weak specifications are based on data type equality rules, and the process of verification is commonly called type checking. Type checking is used to spot source language expressions which are obviously illegal.

A complete specification is one which describes uniquely the program which the programmer must write, including all implementation details such as the choice of data structures. A strong specification is one which lies in the continuum existing between weak and complete specifications. We define informally a useful specification as a strong specification which describes the semantics of the task in enough detail to describe its properties fully, without relying heavily on a given implementation.

A typical example is the problem of sorting a linear list or array. Given a single sorting specification, we should be able to prove that implementations of insertion sort, quicksort, etc. all satisfy the specification.

The verification process consists of comparing the behaviour of the implementation with its specification. If the implementation behaves as required, we say that it 'satisfies' its specification.

Early verification techniques called for after-the-fact verification: one would propose a detailed specification and then build an implementation. The verification process consisted of assuming that the specification was definitive and checking for correctness. It is believed at present that specifications and implementations should evolve concurrently. A relatively weak specification is built, along with an implementation which satisfies it. The specification's requirements are then described in greater detail and the implementation is modified accordingly. Thus, the construction of specifications and implementations is an incremental process.

2.2 Partial and Total Correctness

A program is partially correct with respect to its specification if it behaves as required whenever it terminates. There is no guarantee that the program will terminate: indeed, a non-terminating program is partially correct. A program is totally correct with respect to its specification if the program is partially correct and it terminates.

Total correctness is a stronger property than partial correctness, and has been shown harder to prove since proof of termination is equivalent to the halting problem for Turing machines. However, there exists a large class of

important problems in computer science which require non-termination, such as operating systems and database managers.

2.3 Verification Methods

Several theories describing the semantics of programming languages exist, and they can be classified as follows: the operational, axiomatic and denotational approaches.

2.31 The Operational Approach

The semantics of programming language constructs are defined in terms of a low-level abstract machine and a collection of translation functions which convert source language programs to equivalent abstract machine programs. The basic assumption is that the abstract machine is extremely simple and that the meaning of its state transitions is intuitively obvious and unambiguous.

Verifying a program using this approach requires applying all of the relevant source language construct translators to the source program, resulting in a series of state transformations on the abstract machine. The low-level program is then executed for specific input data and correctness is determined by tracing through the

execution.

The operational approach is obviously equivalent to traditional program testing at a lower level of semantic complexity. Thus, it does not solve the general verification problem in a satisfactory fashion. The formal system of the operational approach has inherent limitations which inhibit the statement of general mathematical theorems. However, it has been used to generate test data automatically [How76] and can be extended to prove correctness mathematically.

2.32 The Denotational Approach

This approach was invented by Scott and Strachey [Sc70a, Sc70b, SS71, Sto77] and is based on the notion that programming language constructs are denotations for their abstract values. For each construct, a 'semantic valuation function' is defined (usually recursively) which maps its syntax to the abstract values which it denotes. The semantic functions are typically formulated using Church's λ-calculus [Chu51]. Several syntactical transformations (conversion rules) are available, e.g., functional application: $(\lambda x.x + 1)4 = 4 + 1$.

There are two possible evaluation (reduction) orders for λ-expressions: applicative order and normal order.

Formal mathematical formulae are strings of symbols and the evaluation process is usually expressed in terms of rewrite rules. Applicative order is 'evaluation from within' and it implies that the value of an expression depends on the values of its parts. Normal order evaluation proceeds 'from without' and it corresponds to call-by-need or lazy evaluation [Sto77]. When a λ-expression cannot be reduced (evaluated) any further, we say that it is canonical or normal. If a λ-expression has a normal form, then applying normal order reduction will always terminate [Sto77]. Applicative order reduction can be much faster (in terms of reduction steps), but there is no guarantee that the normal form can be found. The speed of normal order evaluation can be increased by using graphs to represent the λ-expressions; evaluating an expression then proceeds by graph reduction [Joh83]. Given a λ-expression such as

$$(\lambda x.\lambda y.y)\ ((\lambda z.zz)\ (\lambda z.zz))\ (\lambda z.z + 1)$$

if applicative order is used, the application $(\lambda z.zz)\ \lambda z.zz$ is an infinite loop! However, if normal order is used, then the argument is evaluated only when absolutely needed: in this case, not at all.

As can be seen from the example given, functions are first-class objects in the λ-calculus, in the sense that they can be passed as parameters, evaluated, returned as

function values and built at evaluation time. Since the semantic ° (valuation) function of a construct is a λ-expression, all λ-calculus syntactic transformation rules are available when building the valuation function of a program.

The specification of a program is written in the λ-calculus (with suitable high-level notation). Formally, it is the canonical semantic valuation function exhibiting the desired behaviour. Verifying a program consists of checking that the specification and implementation functions are equivalent λ-expressions, using the conversion and equality rules. The fact that abstract mathematical functions are used to describe the behaviour of programs makes it possible to state the programs' properties independently of specific input values. The disadvantage of the operational approach thus does not appear in the denotational approach.

It has been shown that the λ-calculus has the same expressive power as Turing machines, making it ideal to describe the semantics of programs written for Von Neumman computers. Since general recursion is used in the definition of semantic valuation functions, this method is most appropriate in verifying partial correctness.

2.33 The Axiomatic Approach

The semantics of programming language constructs are described by the general form P { S } Q, where S is a syntactic construct, P is a precondition which must be true before execution of S, and Q is a postcondition which must be true after execution of S. Note that both pre- and postconditions are assertions: logical formulae used to describe the required behaviour of the construct.

Floyd [Flo67] introduced the inductive assertion method where a predicate transformer - a function which maps an input assertion (precondition) and a syntactic construct to an output assertion (postcondition) - is provided for each syntactic construct in the language. In Floyd's work, the important predicate transformers are the strongest verifiable consequent (svc) transformers. Given a particular input assertion P, the strongest (most limiting) condition that can result from evaluating a syntactic construct 'const' is the value of 'svc (P, const)'.

The program to be proved must contain global pre- and postconditions (P, Q), as well as intermediate assertions ($P1 \ldots PN$) which are associated with strategic nodes of the control flow diagram (flowchart) of the program. These intermediate assertions are used to guide the verifier and to help pinpoint any potential errors. Verifying a program

consists of assuming the input assertion to be true and deriving assertions using the predicate transformers, that is, at point P^i in the program, we perform svc (P^i, const) = P^{i+1} where 'const' is the current syntactic construct.

If the derived assertions imply the user-supplied intermediate assertions the program segment processed so far is correct. If the final postcondition matches the derived assertion and the end of the program has been reached, then the program is correct. This method takes its name from the inductive character of deductions performed during the verification process, which can be shown in graphical form:

$$P \rightarrow P^1 \rightarrow P^2 \rightarrow \ldots \rightarrow P^N \rightarrow Q.$$

One important problem with the inductive assertion method is that verifying a program involves a lot of work on the part of the programmer. Since all intermediate assertions must be supplied with the program, it may be hard to localize errors. Furthermore, the verifier might not have the deductive power to derive complex (but true) assertions. This is solved by supplying a large number of intermediate assertions to limit the complexity of the verification process.

The inductive assertion method can be augmented to prove total correctness by adding local information to check

for termination of loops. In this case, the program's output assertion must contain global information about termination.

Hoare's work [Hoa69] is based on Floyd's but he introduces the important difference that program semantics must be theorems in a deductive system. The notation P { S } Q comes from Hoare, and his system includes axioms and rules of inference to prove new theorems. For example, given two statements S1 and S2, it is possible to compose their semantics by the following rule :

$$\frac{P \{ S1 \} Q \qquad Q \{ S2 \} R}{P \{ S1;S2 \} R}$$

No intermediate assertions are necessary since each construct in the language is provided with a set of associated rules of inference. We start with the input assertion and apply rules of inference to derive the final theorem. The semantics of adjacent syntactic constructs are thus coalesced to form the semantics of the program.

One advantage of the axiomatic method (from Hoare) over the inductive assertion method is that less work is involved in correcting a wrong program, since the intermediate assertions are deduced by the program verifier. While the order of verification is irrelevant in the inductive assertion method, in the axiomatic method it is necessary to

process the source program continuously. However, it is still possible for the verifier to be unable to prove a true theorem, due to lack of deductive power.

To prove total correctness, this method is augmented with inference rules for loop termination. These termination rules guide the selection of a single element from a well-ordered set for each pass through a loop. If all loops in a program can be shown to terminate, then the program as a whole terminates.

Dijkstra [Dij76] developed a constructive method for total correctness where the program and its proof of correctness are built concurrently. A predicate transformer which, given a syntactic construct and a postcondition, finds the least limiting precondition, is called a weakest precondition transformer (wp). Dijkstra's method works 'backwards' as compared to Floyd's. The procedure leading to a totally correct program begins with selecting pre- and postconditions P and Q for the program. Starting with Q, we use heuristic search methods to find syntactic constructs 'const' which get semantically 'closer' to P, by performing 'wp (const, Q)'. This is repeated, using Q_{i+1} = wp (const, Q^i) as the new postcondition, until we arrive at a precondition Q^N which can be derived directly from P.

$$P \leftrightarrow Q^N \leftarrow Q^{N-1} \leftarrow \ldots \leftarrow Q^1 \leftarrow Q$$

If there is no syntactic construct which yields a 'closer' precondition to P, then we must backtrack.

While this method has the great advantage of building both a proof and a program, it is not amenable to machine implementation. Generating a correct program from a single pre- and postcondition pair is not practical since it requires an exhaustive search through the tree of all possible syntactic construct sequences for the 'right' sequence. However, it is possible to have a machine check the program produced by using the inductive assertion method on the series of constructs and assertions.

2.4 Verification in Brouwer

In the Brouwer system, a variant of denotational semantics is used to prove total correctness. A Constructive Type Theory (ITT, see chapter 3) forms the basis for verification, rather than λ-calculus. Brouwer is a functional language [Bac78] where data types and functions are first-class objects. Brouwer specifications are abstract data type definitions with an algebraic flavor [Gut77]. The semantic valuation functions map each construct in the source language to equivalent constructs in the type theory.

The power of the type theory is such that the

constraints for total correctness of Brouwer programs can be specified by a data type. The verification process becomes equivalent to strong type checking: a program is totally correct if and only if it has the required type. Unfortunately, type equality is undecidable in general. It is necessary for the programmer to give proofs of all deductive steps that the verifier cannot perform.

Since the value space of Brouwer is a formal type theory, the type checking involved is a purely syntactical process. Well-formedness rules dictate under which assumptions a particular type theoretical construct can be built. Verification of a program then involves checking that all assumptions hold. Brouwer uses 'after-the-fact' verification, but the specification and implementation can be built incrementally.

2.5 Existing Verification Systems

Most existing verification systems are based on Floyd's inductive assertion method or Hoare's axiomatic method. The Stanford Pascal Verifier is an axiomatic verification system for the language Pascal: assertions are inserted in the program text and proved by a powerful theorem prover. This system has seen extensive use and has been the basis for a proof of a compiler [Pol81], the largest verified program so far. Gordon and Milner [MMN75, LCF79] developed an

interactive system founded on denotational semantics with typed λ-calculus. This system (Edinburgh LCF) and its accompanying programming language ML allows one to derive theorems from terms and expressions written in ML.

The AUTOMATH group [vaD73, deB74] developed a proof checker which successfully checked formal proofs of the theorems of chapter 1 of Landau's 'Foundations of Analysis' [Lan55]. The AUTOMATH language greatly influenced the design of Martin-Lof's Intuitionistic Type Theory (ITT), which is the underlying framework of the Brouwer verifier.

Two groups are currently working on ITT as a basis for synthesizing correct programs: the PRL group at Cornell [CB83, CB84, CZ84] and the Programming Methodology group at Goteborg [Nor81, Dyb83, NP83, NS83, Pet83]. The PRL (Program Refinement Language) project is a direct descendant of the PL/CV projects [CJE82, CZ84], and its aim is to produce an interactive specification environment. Specifications are written in a mathematical notation and the verifier automatically generates verification goals and subgoals (theorems and lemmas). It is the task of the programmer to guide the verifier in proving the theorems. The program verifier includes powerful verification tactics and only calls for guidance from the user when it is unable to prove a lemma or theorem. Once a specification is proved, it is entered into a knowledge base and can be used

by subsequent specifications.

The existing PRL system allows the user to reason about integers and lists of integers in a subset of ITT. This subset does not include higher-order functions. The Nearly Ultimate PRL system described in [CB84] is being developed presently and will contain all of the features of ITT (see Chapter 3).

The system developed at Goteborg is written entirely in ML and uses ML's type checking facility (augmented to deal with ITT) to prove programs correct. The result is an interactive system with similar verification strategies to the ones used in Brouwer. The reliance on ML and LCF leads to a bottom-up approach to verification which makes the construction of proofs highly unnatural [Pet83].

Chapter 3

Constructive Logic and Type Theory

3.1 An Introduction to Constructive Mathematics

This section describes the salient points of Constructive Logic (CL) which provides the foundation for the Brouwer verifier. Differences between Classical and Constructive Logic are outlined.

In Classical mathematics, the mathematical objects are assumed to exist independently of our ability to perceive them, and the basic premise is that a proposition must be either true or false. Classical mathematical research thus consists of *discovering* objects and their properties.

Georg Cantor [Can55] introduced the notion of *set*: a set is a collection of elements of a given universe which share a common property. Many paradoxes emerged, the best known one being the paradox of the set of all sets. The formalist school led by Hilbert [Hil25] proposed to *formalize* mathematics completely by abstracting away the meaning of the mathematical terms and symbols. When this approach is taken, mathematics becomes a purely syntactical science, where the truth of a mathematical statement can be ascertained using *rewrite* rules.

In the monumental classic 'Principia Mathematica', Russell and Whitehead [RW96] use formal classical logic and the concept of classes in an attempt at eliminating several set theoretical paradoxes.

In 1907, Brouwer [Bro07] invented Intuitionistic mathemathics (also called Constructive). Brouwer holds that mathematical objects are constructions of the human mind rather than independent objects. the primitive objects are given by the basic mathematical intuition, and they include natural numbers, the principle of induction and finite subsets of the natural numbers.

In Intuitionistic mathematics, the existence of an object depends on either concrete evidence for the object or a rule which yields the desired object when applied. An existential statement (proposition) is thus true if such a rule is found, false if it is proved that there can be no such rule. The halting problem for arbitrary Turing machines is an example of an undecidable problem: there exists no general decision rule which can predict whether a Turing machine will halt. A constructive proposition is thus a statement whose truth-value depends on the existence of a construction.

There are also undecided propositions: statements which could be true, false or undecidable, but for which no

evidence is currently available. An example of an undecided proposition is 'there are ten consecutive 7's in the decimal expansion of π'. Since this question cannot be answered currently, it is undecided and no further comment may be made on its truth-value.

3.2 Constructive Logic

Constructive falsity is '0 = 1', a proposition for which no construction exists. The statement 'P implies Q', written P -> Q, is proved by supplying a rule which, given a proof (construction) of P yields a proof of Q. The proof of an implication is a function whose domain and range are P and Q, respectively. The negation of a proposition P can then be formulated as 'P -> 0 = 1'.

The statement 'P and Q', written P & Q, is proved by giving both a proof of P and a proof of Q. The statement 'P or Q', written P | Q, is proved by producing either a proof of P or a proof of Q.

3.3 Constructive Proof Methods

The fact that a proposition is not necessarily true or false means that the Law of Excluded Middle does not hold in Constructive logic. Thus traditional proof methods such as indirect proof and proof by contradiction cannot be used.

The proof of a theorem must rely entirely on constructing arguments as needed.

It may seem that the prospects of constructive mathematics are dismal. Certainly, some theorems are much harder to prove constructively. However, Heyting [Hey56] and Bishop [Bis67] have shown that it is possible to formulate most of classical mathematics using only constructive arguments.

3.4 Propositions as Types

In light of the explanation of Constructive Logic given above, a constructive proposition can be associated with the type of its proofs. A proposition is true if its type is non-empty. This identification of propositions and types comes from Howard [How69]. It is possible to supplement this logic with universal and existential quantification by allowing free variables in type expressions.

Heyting [Hey56] expresses universal quantification as: 'for all x in A, B [x] holds' where B [x] is a proposition which depends on the value of x. A proof of this proposition is a total function over the domain A which, given an object a of type A, yields a proof of B [a/x] (here 'a/x' denotes substituting a for x in B, see section 3.5). If no object of type A can be built (A is a false

proposition), then the universal quantification is true vacuously, even though it is impossible to build the proof of B.

Existential quantification is equivalent to: 'there exists some x in A, for which B [x] holds'. A proof of such a proposition consists of an element a of type A and a proof of B [a/x]. Since both a and B [a/x] are needed simultaneously, it is natural to insist that they be supplied as a pair.

3.5 Intuitionistic Type Theory

Intuitionistic Type Theory (ITT) is a constructive theory of types invented by Per Martin-Löf [ML76, ML79]. His intention was to develop a formal theory with enough expressive power to formulate the whole of Constructive mathematics. A type in ITT is a set with equality: it describes how to build elements of the type, and how to tell when two elements of a given type are equal. There are several statements (judgments) that can be made about types and their elements:

1) A is a type

2) A and B are equal types

3) a is an element of A

4) a and b are equal elements of A

The judgments are denoted by 'A type', 'A=B type', 'a : A' and 'a = b : A', respectively. The first judgment is a meta-predicate which checks for type well-formedness. The second judgment tests for type equality (which is undecidable in general), while the third one verifies that element a belongs to type A. The fourth judgment tests for equality of elements a and b, both of which must be of type A.

3.51 ITT Notation

The ITT notation used in this thesis adheres to the following conventions:

1. A string of ITT symbols is collectively called an expression, which can be a variable, a constant (0, 1, ...) or an abstraction. A lower-case letter is an element and an upper-case letter is a type. Since types are first-class objects, it is possible to have type-valued elements. An abstraction has the general form '(x : A)e', where x is a bound variable of type A and e is an arbitrary expression.

2. A variable occurring in a given ITT expression can be bound or free. The specific binding rules will be given for each construct but some general definitions can be made:

a) x is free in x.

b) x is free in the abstraction (y : A)e iff x is a different variable from y, is free in A and free in e.

c) x is bound in (x : A)e.

A variable occurring in an expression is bound if it is not free. The fact that variable x may be free in expression e is written 'e [x]'. The sub-expression '(x : A)' denotes an explicit binding.

3. If e [x] (x is possibly free in e), then we can substitute an arbitrary expression for x into e. For example, given the function $\lambda x.x + 1$, the application $(\lambda x.x + 1)4$ proceeds by substituting the constant 4 for x in 'x + 1', resulting in '4 + 1'. Substituting a for x in e is written 'e [a/x]'. If x does not occur in e, then e [a/x] = e.

4. The type theoretical inference rules are of the form

$$\frac{A_1 \quad A_2 \quad \cdots \quad A_n}{C}$$

where the A_i (i=1,n) are <u>assumptions</u> which must be satisfied to enable us to deduce the <u>conclusion</u> C.

5. The fact that a variable x is potentially free in an assumption is indicated by:

$$\frac{\begin{array}{c}[x : A]\\ b : B\end{array}}{C}$$

which states that b [x], B [x] and x : A. This notation is equivalent to:

$$\frac{\begin{array}{ccc}x : A\\ \cdot \quad \cdot \quad \cdot \quad \cdot & b[x] : B[x]\end{array}}{C}$$

<u>3.52 Substitution Rules</u>

$$\frac{a : A \qquad \begin{array}{c}[x : A]\\ B \text{ type}\end{array}}{B\,[a/x] \text{ type}}$$

This rule states under which conditions we can substitute the element a of type A into the type-valued expression B. Clearly, if B depends on some x of type A, then substituting a for x in B results in a valid ITT type. Obviously, if a : C and C is not equal to A, then the substitution is invalid. The substitution rule for elements is:

$$\frac{a : A \qquad \begin{array}{c}[x : A]\\ b : B\end{array}}{b[a/x] \; : \; B[a/x]}$$

As explained above, it is possible for both element and type expressions to depend on the same variable. This is useful when formulating quantification, since both the proposition B and the proof must depend on x (see section 3.4).

There are rules rule for the subsitution of equal elements of types and equal types, and these are given in the section devoted to equality in ITT.

3.53 Evaluation of ITT Expressions

In ITT, normal order reduction is used to evaluate expressions. The fact that an expression is in normal form can be ascertained by looking at its outer-most construct. For example, even if the body b of the function $\lambda x.b$ is unevaluated, the function is in normal form. An expression in normal form is canonical: it has itself as value.

3.54 ITT Types

A type is described by providing rules for type formation, element introduction, element elimination and equality.

The type formation rules describe the construction of valid type expressions, while the introduction rules define the allowable elements of a type. The elimination rules

show the mechanism used to evaluate noncanonical expressions and the equality rules describe the actual evaluation process. For each construct, the LISP representation used in the Brouwer verifier is given.

All of Martin-Löf's types are included, except for the well-ordering type. The well-ordering type defines a tree-like data structure with potentially infinite branching factor and an iteration construct on such trees. All of the existing program construction systems based on ITT have ignored well-orderings for pragmatic reasons. The PL/CV3 type theory of Constable and Zlatin [CZ84] uses well-orderings extensively, but PL/CV3 was never put into practical use: the project was abandonned due to technical difficulties. The PRL program synthesis system [CB79] is based on a subset of ITT including naturals and lists of naturals, but excluding high-order functions.

It is important to note that my version of the type theory differs from Martin-Lof's: modifications include explicit binding of variables in constructs, instead of the implicit bindings of pure ITT.

3.541 Functions

I. Function Formation

$$\frac{A \text{ type} \qquad [x : A]\ B \text{ type}}{(x:A) \rightarrow B \text{ type}}$$

An object of type (x:A) -> B takes an object a of type A and returns the corresponding element b : B [a/x]. If x is not free in B, then (x:A) -> B can be abbreviated A -> B, which is the traditional function type. The function type constructor is right-associative.

$$A \rightarrow B \rightarrow C \iff A \rightarrow (B \rightarrow C)$$

Unless otherwise stated, all ITT constructs are right-associative. The LISP representation of the function type constructor is as follows:

```
(defrec /binding type variable)
(defrec /abstract binding expression)
(defrec /Pi domain range)
```

where domain is a binding record and the pair 'domain range' forms an abstraction. For a description of dialect of LISP used, see the Appendix A. Briefly, the function 'defrec' defines a LISP record, with functions for selecting fields by name, such as binding_type. The type (x:A) -> B is represented as the hunk:

[Pi [abstract [binding A x] B]].

The function type constructor is called Pi for historical reasons: Martin-Löf initially used the notation $(\Pi x \in A)B$. The generalized function type (x:A) -> B is equivalent to universal quantification (see section 3.4) and A -> B is constructive implication.

II. Function Introduction

$$\frac{\begin{array}{c}[x : A] \\ b : B\end{array}}{(x{:}A).b \;:\; (x{:}A) \to B}$$

The original function object of Martin-Lof is of the form $\lambda x.b$, with the binding (x : A) implicit. A function is denoted in LISP by:

(defrec /lambda binding body)

where the pair 'binding body' forms an abstraction.

III. Function Elimination

$$\frac{c : (x{:}A) \to B \qquad a : A}{c\ (a) \;:\; B\ [a/x]}$$

This is the application of function c to the argument a. Here c must be of form $\lambda(x : A).b$. Functional application is left-associative in ITT. Application is represented by:

(defrec /apply function argument).

IV. Function Equality

$$\frac{\begin{array}{cc}[x : A] & \\ a : A & b : B\end{array}}{(\lambda(x{:}A).b)\ (a) = b\ [a/x] \ : B\ [a/x]}$$

As can be seen, the equality-rule contains information relevant to reduction of noncanonical objects. If x is not free in b, then $(\lambda(x : A).b)\ (a) = b$, i.e. this function is constant.

3.542 Tuples

I. Tuple Formation

$$\frac{\begin{array}{cc}[x : A] & \\ A \text{ type} & B \text{ type}\end{array}}{(x{:}A) \times B \text{ type}}$$

The variable x is bound in $(x{:}A) \times B$. If x is free in B, $(x{:}A)$ B can be seen as a free-union in the Pascal sense [JW78]. Indeed, the tuple type corresponds exactly to a variant case where the element x of type A is the tag. This construct is richer than the Pascal free union since the tags are not limited to elements of scalar types: objects of any type can be used as tags, including functions and types.

If x is not free in B, then $(x{:}A) \times B$ can be abbreviated as $A \times B$, which is the product of types of standard denotational semantics. The tuple constructor is

represented in LISP by:

(defrec /sigma domain range)

where domain is a binding and range is an arbitrary expression. Martin-Löf originally used the notation $(\Sigma x \in A)B$ for the tuple type.

If x is free in B, then B [x] defines a family of types and the type $(x:A) \times B$ is equivalent to existential quantification over B. The type $A \times B$ corresponds to constructive conjunction.

II. Tuple Introduction

$$\frac{a : A \qquad b : B\,[a/x]}{(a,b) \;:\; (x:A) \times B}$$

To build an object of type (x:A) B, both a : A and b : B [a / x] must be supplied. If an a can be found such that b exists, then the existential quantification is proved, and if a and b are independent, then supplying both amounts to a proof of A & B. The object (a,b) is called a pair or tuple. Tuples in LISP are:

(defrec /tuple first second).

III. Tuple Elimination

$$\frac{c : (x{:}A) \times B \qquad d : C[(x,y)/z] \quad [x : A], [y : B]}{\text{split } (\ c, (x{:}A)\ (y{:}B)\ d) \ : \ C[c\ /\ z]}$$

In this case, c is a tuple and C is a type-valued expression with a free variable z of type $(x{:}A) \times B$. The explicit abstraction '(x:A) (y:B) d' indicates that d can have x and y as free variables. LISP defines split as:

(defrec /split pair firstvar secondvar exp)

where pair must be a tuple, and firstvar and secondvar are bindings. In LISP, a split object has form

```
[split pair [abstract [binding type1 var1]
                      [abstract [binding type2 var2]
                                exp]]].
```

IV. Tuple Equality

$$\frac{a : A \qquad b : B\ [a\ /\ x] \qquad d : C\ [(x,y)\ /\ z] \quad [x : A], [y : B]}{\text{split } ((a,b), (x{:}A)\ (y{:}B)\ d) = d\ [a,b/x,y] \ : \ C\ [(a,b)/z]}$$

A typical example of such a splitting operation is the selection of either component of a pair. The function head is defined as accepting a pair of type $(x{:}A) \times B$ as parameter and returning the first component. Similarly, tail returns the second component:

```
head :== λ(pair : (x:A) X B)
                .split (pair, (x:A) (y:B) x)
```

```
tail :== λ(pair : (x:A) × B)
                .split (pair, (x:A) (y:B) y).
```

3.543 Free Unions

I. Free Union Formation

$$\frac{A \text{ type} \qquad B \text{ type}}{A + B \text{ type}}$$

The type A + B is the disjunction of the propositions A and B. The two disjuncts must be independent. The free union type constructor is represented by:

```
(defrec /or type1 type2).
```

The disjunction type can be simulated with the tuple type (x:boolean) C, where C can yield either A or B, since it is simply a free union (variant case) with two alternatives.

II. Free Union Introduction

$$\frac{a : A}{or1(a) : A + B}$$

$$\frac{b : B}{or2(b) : A + B}$$

The functions or1 and or2 are used here as tags for the elements of the different types A and B. This is represented by:

(defrec /first expression)

(defrec /second expression).

III. Free Union Elimination

$$\frac{\begin{array}{ccc}[x : A] & & [y : B]\end{array}}{\begin{array}{ccc}c : A + B & d : C[or1(x)/z] & e : C\ [or2(y)/z]\end{array}}$$
$$\text{case } (c,\ (x{:}A)\ d,\ (y{:}B)\ e)\ :\ C\ [c\ /\ z]$$

The type-valued expression C can have a free object z of type A + B. The name of this free variable is not known in the case expression. The corresponding LISP construct is:

(defrec /case criterion first second)

(defrec /alt assumption expression)

where first and second must be alts (alternatives). The alternative construct corresponds to an abstraction. Translating the case expression into LISP records gives:

```
[case crit [alt [abstract [binding A x] d]]
           [alt [abstract [binding B y] e]]
].
```

IV. Free Union Equality

$$\frac{a : A \qquad d : C[or1(x)/z] \quad e : C[or2(y)/z] \quad [x : A] \quad [y : B]}{case\ (or1(a),\ (x{:}A)\ d,\ (y{:}B)\ e) = d\ [a/x] : C[or1(a)/z]}$$

If the criterion indicates a value of the first type (A), then d is evaluated with the proper substitutions. The case expression is similar to

```
case crit of
     (x : A) +-> d,
     (y : B) +-> e
esac,
```

since the value of the criterion gets bound to x or y (as the case may be) in the expressions d and e.

$$\frac{b : B \qquad d : C[or1(x)/z] \quad e : C[or2(y)/z] \quad [x : A] \quad [y : B]}{case\ (or2(b),\ (x{:}A)\ d,\ (y{:}B)\ e) = e[b/y] : C[or2(b)/z]}$$

Similar to the rule above.

3.544 Finite Types

I. Finite Type Formation

$$N_n \text{ type}$$

Here n must be a natural number. This is not stated as an assumption since the finite type can not be built in the ITT system: it is primitive.

II. Finite Type Introduction

$$m_n : N_n \quad (m = 0, \ldots, n-1)$$

This type is equivalent to the traditional scalar type. For example, N_0 is the type with no elements; N_1 has exactly one element (0) and N_2 is the same as boolean. Note that the various elements of a finite type are not related to one another in any way. The LISP data structure for finite type elements is the same as that for natural numbers (see below).

III. Finite Type Elimination

$$\frac{c : N_n \quad \overset{[x_m : N_n]}{c_m : C\,[m_n / z]} \quad (m=0, \ldots, n-1)}{\text{ncase}\,(c, (x_0 : N_n)\, c_0, \ldots (x_{n-1} : N_n)\, c_{n-1}) : C\,[c/z]}$$

The criterion c in the ncase expression can have values between 0 and n-1, and alternatives (abstractions) are provided for each potential value. In LISP, this is represented as:

(defrec /ncase criterion alts)

where alts is a collection of alternatives.

IV. Finite Type Equality

$$\frac{[x_m : N_n]}{c_m : C\,[m_n / z] \quad (m=0, \ldots, n-1)}{\text{ncase}\,(c, \ldots\ (x_m : N_n)\ c_m\ \ldots) = c_m\,[c/x_m] : C\,[m_n / z]}$$

3.544 Natural Numbers

I. Natural Formation

$$N\ \text{type}$$

Once again, no inference rule is given since the natural number type N is primitive.

II. Natural Introduction

$$0 : N$$

$$\frac{a : N}{a' : N}$$

The notation a' represents the successor of a. The following are the LISP data structures for natural numbers:

(defrec /zero)

(defrec /succ argument).

For example, the number 4 is represented by

[succ [succ [succ [succ [zero]]]]].

III. Natural Elimination

$$\frac{c : N \qquad d : C[0/z] \qquad [x : N],\ [y : C\ [x/z]] \qquad e : C\ [x'/z]}{\text{loop } (c,\ d,\ (x{:}N)\ (y{:}C\ [x/z])\ e) \ :\ C\ [c\ /\ z]}$$

The loop construct corresponds to mathematical induction. The variable x is bound to the current index of the loop, y is bound to the result of the previous iteration, and c is the upper bound on the index. The expression d is the induction basis, and e represents the induction step. The LISP version of the loop is:

```
(defrec /loop count first index result next_val)
```

which gets translated into

```
[loop count first [abstract [binding N index]
                            [abstract [binding C[index/z] result]
                                      next_val]]].
```

IV. Natural Equality

$$\frac{d : C\,[0/z] \qquad [x : N],\ [y : C\,[x/z]]\ e : C\,[x'/z]}{\text{loop}\ (0, d, (x:N)\ (y : C\,[x/z])\ e) = d : C\,[0/z]}$$

This is the rule for the induction basis. The evaluation rule for the induction step is:

$$\frac{a : N \quad d : C\,[0/z] \qquad [x : N],\ [y : C\,[x/z]]\ e : C\,[x'/z]}{\text{loop}\ (a', d, (x:N)\ (y : C\,[x/z])\ e) = e\,[a, \text{loop}\ (a, d, (x:N)\ (y : C[x/z])\ e)\ /\ x,y] : C[a'/z]}$$

An example will make this clear:

$$\begin{aligned}
&\text{loop}\ (2, 3, (x:N)\ (y:N)\ y') \\
&= (\text{loop}\ (1, 3, (x:N)\ (y:N)\ y'))' \\
&= ((\text{loop}\ (0, 3, (x:N)\ (y:N)\ y')')' \\
&= ((3)')' \\
&= 5
\end{aligned}$$

Here, the index variable x is not free in y' and does not contribute to the computation. It is easy to see, that writing

$$\text{loop}\ (a, b, (x:N)\ (y:N)\ y')$$

where a and b are natural is a definition of addition.

3.546 Equality Type

I. Equality Formation

$$\frac{A \text{ type} \quad a : A \quad b : A}{Eq\ (A,\ a,\ b)\ \text{type}}$$

The equality type, although admittedly synthetic, is very important: it permits the unification of type theory and logic.

II. Equality Introduction

$$\frac{a = b : A}{eq\_proof : Eq\ (A,\ a,\ b)}$$

The special element eq_proof only exists if it can be shown that a = b : A. The usual reflexive, symmetric and transitive properties of equality can be shown as inference rules. The rules for determining whether two ITT expressions are equal can be found in Appendix B. LISP uses the following data structures for equality types:

```
(defrec /Eq type exp1 exp2)
(defrec /eq_proof).
```

III. Equality Elimination

$$\frac{\text{eq\_proof} : \text{Eq}(A, a, b)}{a = b : A}$$

Given an element of Eq (A, a, b), we can conclude that a and b are equal elements of type A. This is used extensively in the verification process along with the substitution rules, since substitution of equals preserves meaning.

IV. Equality Evaluation

$$\frac{a = b : A \qquad d : C[\text{eq\_proof} / z]}{\text{Test}(\text{eq\_proof}, d) = d : C[\text{eq\_proof}/z]}$$

This is a conditional test on the existence of the equality proof for the assertion (proposition) a = b : A. The test construct controls the evaluation of expressions whose meaning depend on finding an equality proof.

3.547 Universes

I. Universe Formation

$$U_n \text{ type}$$

Universes are primitive types and are indexed by natural numbers e.g. U_0. LISP represents universes by:

(defrec /universe size)

where size must be a LISP integer.

II. Universe Introduction

Universes are finite types forming an ascending chain such that all types in U_n are also in U_{n+1}.

$$U_0 : U_n$$

$$\vdots$$

$$U_{n-1} : U_n$$

The types N and N_n are elements of U_0. The type constructors (function, tuple and free union) are elements of the universe for which the following inference rules apply:

$$\frac{A : U_n \qquad [x : A]\ B : U_n}{(x:A) \to B : U_n}$$

$$\frac{A : U_n \qquad [x : A]\ B : U_n}{(x:A) \times B : U_n}$$

$$\frac{A : U_n \qquad B : U_n}{A + B : U_n,}$$

and the equality type Eq (A, a, b) is an element of U_n if $A : U_n$, a : A and b : A.

III. Universe Elimination

$$\frac{A : U_n}{A \text{ type}}$$

If $A : U_n$, then A is a type, and A is automatically a member of U_{n+1}, as stated above. There is no equality rule for universes since we cannot evaluate them.

3.6 Modifications to the Type Theory

Experience with ITT as a programming language led to the discovery of limitations and desirable additions to the type theory for the purpose of verification.

3.61 Finite Types

Finite types are primitive in ITT. While this may seem desirable at first glance, it invalidates many useful applications. For example, the type of an array containing 3 naturals can be described as: N_3 -> N. Thus an array is a function and indexing the array involves a function call. An advantage of this description is that consistency with array bounds is ascertained at compile time: since the index expression must be of type N_3, no run-time checks are necessary. The general array constructor is of type

$$(\text{type} : U_4) \times (n : N) \to (N_n \to \text{type}).$$

So, array (N, 10) should yield a function of type N_{10} -> N.

Unfortunately, N_n cannot be built for an arbitrary natural number n. The finite type formation rule must be changed to:

$$\frac{n : N}{N_n \text{ type}}$$

The array function can be formulated as:

$$\lambda((type, n) : (U_4 \times N))$$
$$.\lambda(idx : N_n)$$
$$.ncase^n(idx, \ldots)$$

where the ncase expression has precisely idx case clauses. The current loop construct only iterates on expressions of type N. Since finite types and N are disparate, arbitrary arrays cannot be built in this type theory.

3.62 Let Expression

A lexically-scoped 'let' construct has been added to the type theory to facilitate the production of programs. The let construct is not a type but an evaluation mechanism, whose effect is to define an evaluation context.

II. Let Elimination

$$\frac{\begin{array}{c}[x : A] \\ a : A \quad exp\ [x] : B\end{array}}{let\ (a, (x : A)\ exp) : B}$$

III. Let Equality

$$\frac{\begin{array}{c}[x : A] \\ a : A \quad \exp\,[x] : B\end{array}}{\text{let}\ (a, (x : A)\ \exp) = \exp\,[a/x] : B}$$

Chapter 4

Description of the Brouwer language

4.1 Brouwer, the language

Brouwer is a language designed to be readily verifiable. While most verification systems have different notations for specifications, implementations and proofs, Brouwer uses a single notation. This is possible because Brouwer is based on Intuitionistic Type Theory: a specification is a data type, a program is a proof, and each is a valid Brouwer expression.

The Brouwer source language is a programming language notation for ITT, with syntactic abbreviations for standard operations. A Brouwer program is scanned, parsed and translated into equivalent ITT constructs by semantic action routines. Since it is impossible for any ITT expression to produce side-effects, Brouwer is a functional language [Bac78]. The complete grammar for Brouwer can be found in Appendix C.

4.2 Brouwer Types

4.21 Primitive Types

Brouwer contains several primitive types which have exact equivalents in the type theory. These are **void**, the type with no elements; **unity**, the type with exactly one element (0); **boolean**, the type with elements **true** and **false**; and **scalar**, which is defined either with a list of identifiers:

scalar (RED, GREEN, BLUE)

or with a natural number which indicates the number of elements in the scalar type, e.g. **scalar** 4 has elements 0,1,2 and 3. In particular, **void**, **unity** and **boolean** are identical to **scalar** 0, **scalar** 1 and **scalar** 2, respectively. These scalar types are equivalent to the finite types of chapter 3.

The type **natural** corresponds to the type of all natural numbers, and the type **universe** corresponds exactly to the ITT universe type. A particular ITT universe is selected by supplying a natural number, e.g. **universe** 3. As in the case of scalar types, the parameter defining the universe must be an explicit constant at verify-time.

4.22 Type Constructors

A type constructor is provided corresponding to each of the (constructed) ITT types. Their respective syntax is:

```
eq_type     ::= @type expression1 = expression2
func_type   ::= domain -> range
tuple_typ   ::= domain &  range
union_typ   ::= type1  |  type2
```

The grammar of Brouwer does not restrict the expressions allowable: the program verifier (see chapter 5) checks for well-formedness of all expressions and types according to the rules of ITT.

4.23 The Context Meta-Type

The special type **context** asserts that it the type of an expression can be extracted from context. For example, if a function BIG expects another function SMALL as argument, and the parameter's domain and range types can be deduced, we can write:

```
BIG : (SMALL : context T -> T) ->...
```

instead of the usual Curried function:

```
BIG : (T : universe 0) -> (SMALL : T -> T) -> ...
```

4.3 Objects and Evaluators

The objects in Brouwer are functions, tuples, free unions, natural numbers, equality proofs and the elements of the scalar types. The syntax of a function is

```
function ::= 'function' '('par_list')' expression 'end'
par_list ::= param par_list | param
param    ::= ident_list ':' type
```

Each parameter must consist of at least one identifier and a type. The function is transformed into a Curried λ-expression and components of the parameter list - which denotes an implicit tuple - are translated into splitting constructs, which are substituted, when necessary, in the body of the function. A function call is of the form:

```
func (argument)
```

Tuples are represented by '[expO;...; expN]' and are translated into '(expO, ...,(expN-1,expN)...)' by the semantic action routines. Tuples are evaluated by the 'split' construct, whose semantics are identical to ITT's split.

```
split TUPLE into [x : A;y : B] in expression end
```

It is also possible to reference a tuple's components by name. If a tuple type has the form:

TUP :== (N1 : T1) & (N2 : T2) &...& (Nn : Tn)

then TUP.Nm is equivalent to the required projection expression.

Free union objects are denoted by **first** (expression) and **second** (expression). The evaluator for free unions is:

```
case expression of
     (y : typ1) +-> expression,
     (z : typ2) +-> expression
esac
```

It may seem that both scalars and free unions can use the case expression unambiguously. This is usually true, since the number of alternatives is always two for disjoint unions. However, if the free union is of the form A + A, it is not possible to decide the type of the expression at translation time. Thus, the ncase expression with similar format for the alternatives is provided for scalars. A useful shorthand notation for the boolean type is

```
if expression then
   true_exp
else
   false_exp
fi
```

The natural numbers are represented by 0,1,2,... and the

expression a' corresponds to ITT's a'. The evaluator for natural is the iterative construct 'loop'.

```
loop to COUNTER  giving (LAST_VALUE : LPTYPE)
   zero +->  FIRST_VAL
 succ I +->  NEXT_VAL
pool
```

and its meaning is identical to

```
loop (COUNTER, FIRST_VAL , (I : natural)
                           (LAST_VALUE : LPTYPE)
                           NEXT_VAL
   )
```

where the giving clause is optional. The let expression is formulated as

```
let ident : type
          :== expression
in
    expression
```

4.4 Brouwer's encapsulation mechanism: the Module

The encapsulation mechanism used in Brouwer is the **module.** A module consists of several sections: the **module header**, the **import** and **export** lists, the **declaration** section and the **property** section.

```
module      ::= module_head in_out_lists decls
                properties 'eludom'

module_head ::= 'module' id_with_type
```

The module header consists of a module name and, optionally, a module type. The type is used to distinguish between specification and implementation modules. Specification modules must have universe types and the instantiation of such a module is closely analogous to a function call. For example, a stack specification module could have the following header:

```
module STACK_SPEC : (T : universe 4)
```

and the implementation module could then have the header

```
module STACK : (ELEMENT_TYPE : universe 4) ->
                                STACK_SPEC (ELEMENT_TYPE)
```

The interface between a module and its environment is controlled by the import and export lists.

```
in_out_lists ::= imports exports
imports      ::= 'import' ident_list | ϵ
exports      ::= 'export' ident_list | ϵ
```

Declaration before use is strictly enforced; in this case, names on the import list must have been exported by a previously seen module. All of the exported definitions of a module whose name appears on an import list are visible.

The export list controls precisely which local declarations are visible from outside the module. Names on the export list are made visible only to the next module level unless the current module is global, in which case all modules can access the name. This mechanism makes it possible to hide implementation-dependent information.

```
module Small_module
   export A,B,C

   define A : natural :== 0

   module B
      export C,D

      define C : ... :== ...

      module D
         .
         .
         .
      eludom  {  D  }
   eludom  {  B  }

   define E : boolean
            :== ... C ...
eludom  {  Small_module  }
```

In this example, A, B and C are exported to the global level by module Small_module. Note that since C is defined in a nested module, it must be exported to Small_module.

The declaration section of a module has the following syntax:

```
dec_list  ::= dec_list  decl   |  decl
decl      ::= module   |  'define' typed_id decl_val
```

```
decl_val  ::= ':==' expression | ϵ
```

The declaration list contains local definitions with type declarations and the value is optional. The type of a variable even if its value is given in a different module. Any nested module can access definitions from a lexically enclosing module provided that they are explicitly imported and they have been already defined. The syntax for the property section is:

```
properties ::= 'property' prop_list | ϵ
prop_list  ::= prop_list property   | property
property   ::= 'prove'   typed_id  decl_val
```

The property section of a module has syntax identical to that of the declaration section (except for the choice of keywords) and the semantics are very similar. In a specification module, this section holds the properties which the local programs must have. These properties are proven by the programmer in the matching property section of the implementation module.

The property section is redundant since any theorem can be defined as a declaration, but it contributes to the algebraic flavor of specifications by making a distinction between objects and their properties.

The module does not have an analog in the logic : it is a language feature which does not survive the translation of the program into ITT. However, it is possible to view the result of the instantiation of a module as a tuple of 'let' constructs defining the various values and types.

4.5 Recursion in Brouwer

General recursion is not allowed in Brouwer and this would seem to be an important disadvantage, rendering the potential usage of the language extremely limited. However, the facts that functions and types are first-class objects and that there exist operators on both, permits us to compute a subtantially larger class of functions than the restriction would tend to imply. Consider the task of '^', an iteration operator on functions. Given a function f of type T -> T, where T is some type, and a natural number n, we can build the function f (f (... f (x)...)) with n terms 'f (..)', in which x is a formal parameter of type T. The operator '^' is defined to be of type

```
'^' : ((context T -> T) & natural) -> (T -> T)
```

and is built with the inductive construct

```
function (F : context; N : natural)
   loop to N giving (PREDFUN : T -> T)
      zero +-> Id (T),
```

```
        succ I  +->  function (Y : T)
                           F (PREDFUN (Y))
                     end
    pool
end
```

where the function Id is defined as

```
define Id : (T : universe 4) -> (T -> T)
          :== function (X : T)
                  X
              end
```

It would, of course, have been more palatable to formulate '^' as

```
function (F : context; N : natural)
   if N = 0 then
      Id (T)
   else
       function (Y : T)
         F ((F ^ (N - 1)) Y)
       end
   fi
end
```

The main problem here is that '^' is recursively defined which cannot be allowed because of the possibility of non-termination inherent in general recursion. However, it can be argued that the inductive definition is clear since it exposes some of the details of the implementation.

4.6 Boolean Connectives

This section presents simple Brouwer programs for the boolean connectives **and**, **or** and **not**.

```
and : (boolean & boolean) -> boolean
    :==
        function (a,b : boolean)
           if a then
              b
           else
              false
           fi
        end

or  : (boolean & boolean ) -> boolean
    :==
        function (a,b: boolean)
           if a then
              true
           else
              b
           fi
        end

not : boolean -> boolean
    :==
        function (a : boolean)
           if a then
              false
           else
              true
           fi
        end
```

Note that since boolean objects are computable in all cases, the definitions of these logical connectives are essentially identical to the classical formulations. Any expression can be returned as the value of a function.

4.7 Finite Lists

While generalized arrays are not available in Brouwer, finite lists can be formulated readily. The list constructor 'list' is a function which takes 3 parameters: the type of the elements, the number of elements and the initializing value. The type of a finite list is computable and is defined as

```
define nil : unity
           :== 0

define lis_typ : (universe 4) & (natural)
                      -> universe 4

          :== function (T : universe 4; n : natural)
                 loop to n giving (ltyp : universe 4)
                   zero  +-> unity,
                   succ I  +-> T & ltyp
                 pool
               end
```

The list type depends on the desired number of items; the empty list marker nil is also used as a list terminator. The general list constructor is

```
define list : (T : universe 4; n : natural;
                                first : T)
                  -> (natural & lis_typ (T,n))

          :==
             function (T : universe 4;
                       n : natural;
                       first : T)

               loop to n giving (l : lis_typ (T,i))
                 zero  +-> nil,
                 succ  I +-> (first, l)
                 pool
```

```
end
```

All of the usual operations on lists can be defined: for example, the extraction of an element corresponds to a single loop on its index. Changing the value of a component of a list results in the creation of a new list, since there is no concept of store in Brouwer. Clearly, objects of any type can be organized in lists.

4.8 The Ackermann Function

The Ackermann function demonstrates the power of high-order functions, since it is the traditional example of a non-primitive recursive function. The Ackermann function can be stated as:

$$\begin{aligned} &Ack\ (0,\ n) = n + 1 \\ &Ack\ (m,\ 0) = Ack\ (m-1,\ 1) && \text{if } m > 0 \\ &Ack\ (m,\ n) = Ack\ (m-1,\ Ack\ (m,n-1)) && \text{if } m,n > 0 \end{aligned}$$

The Brouwer formulation of this function is actually built with three distinct functions.

```
first  :== function (n : natural)
             n'
           end

next   :== function (g : natural -> natural)
             function (x : natural)
               (g^x) (g (1))
             end
           end
```

```
Ack    :==  function (m,n : natural)
              ((next^m) (first)) (n)
            end
```

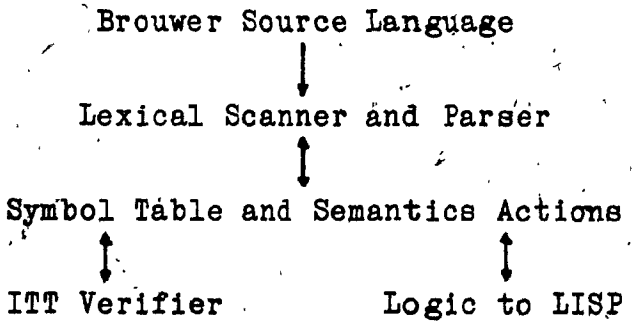
Chapter 5

The Brouwer Program Verifier

An overview of the Brouwer verification system is given and the choice of the application language is justified. A Verification Meta-Language (VML) is employed to present the verification rules corresponding to the various ITT constructs.

5.1 System Overview

The Brouwer system is an interactive specification and verification environment for proving programs totally correct.



The syntax of a Brouwer source file is analyzed by the lexical scanner and parser. When a production is reduced in

the parser, a semantic routine is executed to update the symbol table or generate ITT expressions representing the semantics of the source constructs. The semantic action routines are the concrete realizations of the semantic valuation functions which map Brouwer constructs to ITT expressions.

When a declaration is found, the verifier is invoked by the semantic routines. If the declaration is verified to be correct, i.e., the program satisfies its specification, the ITT expression is translated into executable LISP code and installed in the global working environment. Currently, LISP is the command language for the execution of Brouwer programs.

The Brouwer verifier is written in LITHP, a dialect of LISP [McC80] developed by H.J. Boom for the VAX 11/780 computer, running under VAX/VMS. The LISP language is ideal for implementing the Brouwer verifier because it allows the manipulation of symbolic information in a very natural fashion. However, LISP is ill-suited for lexical scanning and parsing, and the advantages of traditional imperative languages like Pascal [JW78] for these tasks are obvious. This problem is solved in a rather unorthodox fashion by linking LISP with Pascal, thereby benefiting from the best of both worlds.

5.2 Lexical Scanner and Parser

The Brouwer grammar (see Appendix C) is fed to H.J. Boom's parser generator ANAL, resulting in a set of SLR(1) parsing tables. A Pascal program performs scanning actions and interprets the tables. Since LISP needs the symbol table during the verification process, a communication scheme is set up to permit information transmission between LISP and Pascal.

Two variants were implemented: linking the object code produced by the Pascal compiler directly into the LISP system as 'resident' code, and passing messages through VAX/VMS mailboxes.

5.21 Linking Pascal and LISP

The parsing executive function PARSE is written in MACRO (VAX assembler) and is linked with the LISP system. Calling Pascal involves hiding the Pascal stack frame from the LISP garbage collector by faking a large reserved area on the stack for all of Pascal's local variables. The parser is invoked by calling PARSE with the semantic action routine SLR_ACTION as parameter. The parsing schedule follows:

```
Initialize parser;
while not (end-of-file or error) do
   wait for reduction;
   if reduction found then
      hide Pascal stack frame;
      call SLR_ACTION with production number;
      return SLR_ACTION status code to Pascal
   fi
od
```

The problem with this approach is that whenever a change needs to be made to the parser, the LISP system must be relinked, meaning that the verifier system - which is a LISP world - must be regenerated.

5.22 Satellite Communication

The second solution is to run the Pascal parser as a detached process under VMS (a satellite to the LISP system). Two mailboxes are created to allow inter-process communication, and to avoid the problem of a process reading its own message. The following protocol defines the parser-semantics interface:

Parser -> Brouwer	Brouwer Action
integer > 0	accept production number
0	syntax error, stop semantics
-1	end of file reached
-2	empty source file
integer > -2, string	lexical action with symbol

Brouwer -> Parser	Parser Action
integer (s) > 0	semantic error number(s)
0	no action (all OK)
-1	initialize parser
-2	halt parser process

At the beginning of a Brouwer session, the parser process is created and it remains available until explicitly aborted. This arrangement allows cheap revision of both the parser and the semantic action routines, since the LISP system's object code is stable: we need not regenerate the verifier 'world' (see Appendix A).

5.3 Brouwer System Organization

The verifier consists of several functionally separate modules. These verification tools are described in this section.

5.31 The Assumption List

All information deduced by the verifier is represented by assumptions. The collection of all knowledge is organized into an assumption list, where each cell has the form:

(defrec /assumption type name value).

The assumption list is passed as a parameter to the various verification modules, enforcing lexical scoping on the visible information.

5.32 Verification Tools

The main executive is VERIFY which is invoked as:

(verify assump_list exp spec)

where assump_list is the system assumption list of the preceding section. Such a call to VERIFY is equivalent to 'exp : spec' in type theoritical terms. Note that 'spec' is not always a type: the verifier uses internal meta-types (see section 5.33). VERIFY returns the actual type of exp if exp satisfies the specification spec. If the verification fails, an error message is emitted. FREEIN is

a routine which is called as in

(freein var context),

and which returns a status code indicating whether 'var' is free in 'context'. The 'context' can be an ITT expression or the system assumption list. The call (freein var assump_list) corresponds to a search of the knowledge base for a previous definition of var. This search is performed when installing new information in the knowledge base: if var is free, then we must change the name to avoid variable capture.

The module REDUCE evaluates an expression by normal order reduction based on the Equality rules of Chapter 3. An invocation of REDUCE must be of the form:

(reduce assump_list exp).

This function is an <u>interpreter</u> for ITT. One possible source of problems is the reduction of a function taking an element from N_0, since this amounts to a counterfactual hypothesis:

false -> anything.

Reducing such a function can cause the interpreter to loop

forever. A global variable REDUCE_CYCLES is arbitrarily set to 1000 to control the maximum number of reduction steps allowed for a single expression. Counterfactual hypotheses have not proved particularly useful in formulating specifications or programs.

As seen in chapter 3, evaluation depends on rewrite rules or substitution. The module SUBST, invoked by

(subst new old exp),

replaces all free occurrences of 'old' in 'exp' by 'new'. The SUBST routine renames variables when necessary, to avoid variable capture. The module ITT_EQ tests for equality of two ITT expressions. The call

(itt_eq assump_list exp1 exp2)

constitutes a request to the ITT_EQ module to prove that exp1 = exp2. Two expressions are equal if they are syntactically identical. The REDUCE module is used to simplify the two expressions, one construct layer at a time. The assumption list can contain clues useful in proving equality. If it is known that A = B, then, by symmetry, B = A. There exist rules for symmetry and transitivity of expressions, as well as rules for term equality for each construct in ITT. For example, the rule

$$\frac{a = b : N}{a' = b' : N}$$

can be interpreted as: to prove that a' = b', prove that a = b. The complete list of term equality rules can be found in Appendix B. The strategy employed in proving term equality of a and b can be given by the following algorithm. Each step in the table is executed only if equality has not yet been proven.

1. Is a = b (trivially)?
2. Search assumption list for a proof of a = b (an entry with value eq_proof)
3. If not equal, search for a proof of b = a (symmetry)
4. If not equal, search for a proof of a = c and c = b (also allow for symmetry)
5. Look for applicable term equality rule to simplify the problem
6. Repeat 1. to 5. until either equal or no rule is found
7. Reduce both a and b (only one layer)
8. If a <> b then
   if a and b are in normal form,
   a <> b
   else
   goto 1.

5.33 Verification Meta-Types

Two internal type-like objects are used during verification: **type** and **unknown**. **Type** is used to test type

well-formedness (first judgment of ITT), whereas checking that an expression satisfies the **unknown** type corresponds to extracting the expression's type from context. If this is not possible, the verification fails. The unknown type is equivalent to the union of all universes, a concept which can not be formulated since ITT cannot generalize over universes.

The verification meta-types are not allowed in the source language (Brouwer) since this would compromise the constructive character of the theory. However, using them during verification is valid because they replace the judgments of ITT.

5.4 General Verification Strategy

The inference rules presented in chapter 3 have the form:

$$\frac{A_1 \quad A_2 \quad \cdot \quad \cdot \quad A_n}{C.}$$

Verifying that C is valid consists of checking that all A_i are true. For example, consider the function type formation rule.

$$\frac{A\ \text{type} \qquad B\ \text{type}}{(x{:}A) \rightarrow B\ \text{type}}$$

Given an expression (x:A) -> B, both A and B must be valid type expressions. The verification rules for types, object and evaluators are derived from the rules for type formation, introduction and elimination, respectively. However, verification is not simply a matter of 'reversing' the inference rules. Since substitution of equal terms preserves meaning, it is necessary to check equivalent type and element expressions when verifying the inference conditions. Finally, an expression might not be in normal form. Expressions should be evaluated until the judgment is proven or disproven conclusively.

In practical terms, the program is partially evaluated during the verification process. Indeed, since types are computable objects, it may be necessary to evaluate x and A to discover that x : A.

5.5 Verification Rules

This section describes the rules used in the Brouwer verifier. The functional programming language VML (Verification Meta-Language) has been developed to describe these rules in succinct terms. VML programs define verification strategies for the ITT system.

5.51 VML Notation

A VML program is a sequence of requests to the verifier. Any request will either fail or succeed. In general, if a request fails, an error message is issued and the program is terminated. There are two exceptions to this rule, ('if' and 'or') as described below. For example, the statements

```
check_A  ==  A ::  type
check_B  ==  B ::  type
```

are equivalent to 'prove that A is a type, and then that B is a type'. If the first request fails, the second condition is never tested. However, if A is a valid type, then 'check_A' is bound to the deduced type and the second request is evaluated. The '==' operator binds names to values using lexical scoping rules. The only conditional form in VML is the **if** statement:

```
if condition then
   succeed_part
else
   fail_part
fi
```

If the condition succeeds, succeed_part is evaluated, otherwise fail_part is evaluated.

Logical connectives **and** and **or** (with lazy semantics) are available in VML. The **if** and **or** expressions are the only VML constructs to recover from failure. The only iteration construct in VML is the **for** loop, and it is of the form:

```
for i := low to high do
        request_0
          .
          .
        request_N
od
```

The bounds of the loop (low and high) must be natural numbers. If any request in the body of the loop fails, the loop itself fails.

All of the ITT constructs are available in VML as data objects. Thus, an ITT program is treated as data by VML, allowing us to analyze its syntax.

5.52 VML Operators

The VML operators are:

::	Verify
=	Check for Equality of terms
==	Assign Name to Value
~	Match Pattern
≡	Check for Type Conformity
◊	Return Value
{ x }	Add x to Assumption List
↑	Search Assumption List
*	Reduce
??	Signal Error

The VML expression a :: A is a request to the verifier to prove that a : A where A can be a meta-type: the value of a :: A is thus a type (if it succeeds). Since the VML program is a verifier, this is a recursive call. Checking for equal expressions is done with '=', and A ≐ B is shorthand for (itt_eq assump_list A B) (see section 5.32).

The request 'A ~ B' succeeds if A is an object whose structure matches the pattern B. If '~' succeeds, then all identifiers of B are bound to components of A and these are visible to all successive requests at the current scoping level. For example, the request

$$(((x{:}A) \rightarrow N) \times N) \sim (y \times z)$$

succeeds and y == (x:A) -> N and z == N, since the expression has the form of a tuple type.

The operator '◇' is an explicit return mechanism for a VML value. It also provides visual cues by labelling program exit points. Adding knowledge to the assumption list is done with the '{ .: }' operator. Any information added to the assumption list in this manner is only visible to the current request.

```
a :: hello      {x : B}
y :: D
```

In the second request, the information {x :'B} is not visible. The '{ }' operator automatically renames all variables which are previously known to avoid variable capture.

Searching the assumption list is done with the '↑' operator. For example, (name x) ↑ type searches through the list for an entry with name x and extracts the type of the entry.

The monadic operator '*' reduces its parameter to normal form: *(λ(x:N).x) (4) ---> 4. The operator '≡' checks for type conformity: the type A conforms to B if the following VML program succeeds (in the sense defined above). Note that B can be a type or a meta-type.

```
if B = unknown then
   ◇ A
else
    if B = type then
       redtype == * A

          (redtype ~ (x:D) -> E) or
          (redtype ~ (x:D) × E) or
          (redtype ~ D + E).    or
          (redtype ~ Eq (A,a,b)) or
          (redtype ~ N)         or
          (redtype ~ N_i)       or
          (redtype ~ U_i)

   else
       if A = B then
          ◇ A
       else
          ??
       fi
   fi
fi
```

The '??' operator always fails. It emits an appropriate error message (deduced from context) and returns. VML functions have the form:

```
function NAME (par1 ... parn)
    body
end
```

The parameterless function 'gensym' is used to generate unique variable names. While this feature may seem imperative, it is implemented by lazy evaluation.

5.53 The Brouwer Verifier in VML

The VML version of the Brouwer verifier is a recursive function with header:

function verify (spec exp)

All verification rules will be expressed as VML program segments and the assumption list mechanism is implicit. The program exp is totally correct with respect to spec if a return expression is successfully evaluated and the program terminates. Note that since exp and spec are in the function header, they are visible to all requests in the body of verify.

Variables

```
if exp ~ x then
   vartype == (name x) ↑ type
   ◊ vartype ≠ spec
```

If the expression is a variable, its type is extracted from the assumption list and compared with the supplied specification.

Lexical Scoping

The information the let construct contains in its explicit binding and value is type checked and inserted into the assumption list.

```
if exp ~ let (value, (var : vartype) expression) then
   boundtype  ==  if vartype = unknown then
                     ◊ unknown
                  else
                     ◊ vartype :: type
                  fi
   boundtype  ==  value :: boundtype
  ◊ expression :: spec                    {var : boundtype}
```

Function Type

The function type constructor is verified using the formation rule. There are three possibilities for valid specifications: **type**, **unknown** or some universe type. All three are handled by the following VML program segment. Note that the order of the conditions on an **if** expression is important: since the logical connective **or** is lazy, it is much more efficient to check for 'spec = unknown' before the universe match. The utility function check univs finds the smallest universe in which a given ITT type constructor, with domain A and range B, can exist.

```
if exp ~ (x:A) ->  B then
   if spec = type then
      boundtype  ==  A :: type
      restype    ==  B :: type      {x : A}
     ◊ (x:boundtype) -> restype
   else
      if (spec = unknown) or (*spec ~ U_n) then
          ◊ check_univs (spec A B)
      else
      ??
      fi
  fi
```

Function Object

The function object is verified according to its introduction rule.

```
if exp ~ λ(x:A).b then
   if spec = unknown then
      domtype ==  A :: type
      rantype ==  b  :: unknown {x : domtype}
      ◊ (x:domtype) -> rantype
   else
      if *spec ~ (y:C) -> D then
         checkdoms  ==  A = C
         restype    ==  b :: D     {x : A, y : C}
         ◊ (y:C) -> restype
      else
         ??
      fi
   fi
```

Function Application

```
if exp ~ c (a) then
   functype  ==  c :: unknown
   if functype ~ (x:A) -> B then
      checkarg  ==  a :: A
      ◊ B [a / x] ≡ spec
   else
      ??
   fi
```

Tuple Type

```
if exp ~ (x:A) X B then
   if spec = type then
      boundtype  ==  A :: type
```

```
      rangetype  ==  B :: type     {x : boundtype}
      ◇ (x:boundtype) X rangetype
   else
      if (spec = unknown) or (*spec ~ U_n) then
      ◇ check_univs (spec A B)
     else
        ??
     fi
  fi
```

Tuple Object

```
if exp ~ (a,b) then

   if spec = unknown then
      domtyp == a :: unknown
      rantyp == b :: unknown      {a : domtyp}
      ◇ (a : domtyp) X rantyp
   else
      if *spec ~ (x:A) X B then
         check1  ==  a :: A
         check2  ==  b :: B [a/x]    {a : A}
         ◇ *spec
      else
         ??
      fi
   fi
```

Tuple Split .

```
if exp ~ split (tuple, (x:A) (y:B) e) then

   xtype == if A = unknown then
               ◇ A
            else
               ◇ A :: type
            fi

   ytype == if B = unknown then
               ◇ B
            else
               if xtype = unknown then
                  ◇ xtype
               else
```

```
                ◇ B :: type     {x : A}
            fi
        fi

tuptype == if (xtype = unknown) or
               (ytype = unknown) then
              ◇ unknown
           else
              ◇ (x:A)X B
           fi

tuptype == tuple :: tuptype

if *tuptype ~ (z:C)X D then

   exptype == e :: unknown        {x : C, y : D[x/z]}
   exptype == exptype [tuple / (x,y)]
   exptype == exptype [split (tuple,(x:A)(y:B)x)/x]
   exptype == exptype [split (tuple,(x:A)(y:B)y)/y]
   ◇ exptype ≡ spec
else
   ??
fi
```

Free Union Type

```
if exp ~ A + B then

  if spec = type then
       checkA  == A :: type
       checkB  == B :: type
       ◇ checkA  + checkB
  else

     if (spec = unknown) or (*spec ~ U_n) then
        univ1  == A :: spec
        univ2  == B :: spec
        ◇ check_univs (spec univ1 univ2)
     else
        ??
     fi
  fi
```

Selection of First Free Union

```
if exp ~ or1 (alt) then

   if spec = unknown then
      ortype  ==  alt  ::  unknown
      ◇ ortype  +  unknown
   else

      redspec ==  *spec
      if redspec ~ A + B then
         check == alt  :: A
         ◇ redspec-
      else
         ??
      fi
   fi
```

The selection of the second alternative (or2 (alt)) has similar verification rules.

Case Expression

```
if exp ~ case (c, (x:A) d, (y:B) e) then

   checkA  ==  A :: type
   checkB  ==  B :: type
   critype ==  c :: A + B
   or1type ==  d :: unknown                    {x : CheckA}

   or2type ==  e :: or1type [or2(y)/or1(x)]
                                               {y : CheckB}
   ◇ or1type [c / or1(x)] ≡ spec
```

Finite Type

```
if exp ~ N_n then
   if spec = type then
      ◇ exp
   else
      if spec ~ U_n then
         ◇ U_0
      else
```

```
        ??
    fi
fi
```

Zero Object

```
if exp ~ 0 then
   if (spec = unknown) or (*spec ~ N) then
      ◊ N
   else
      if *spec ~ N_n then
         check == 0 < n
         ◊ N_n
      else
         ??
      fi
   fi
```

Successor Object

The function 'checkfornum' is a multiple-valued function which transforms ITT naturals into VML integers, for the purpose of verification. The ITT expression 'exp' is analysed to determine the number of times that the successor operation has been applied. This count is returned as 'num', and the remaining sub-expression becomes 'rest'. Clearly, if exp is a natural number, 'exp' is zero and 'num' is the number.

```
if exp ~ arg' then
   (rest num)  ==  checkfornum (exp 0)
   redspec     ==  *spec
   if redspec = N then
      if rest = 0 then
         ◊ N
```

```
      else
         ◊ rest :: N
      fi
   else
      if redspec ~ N_i then
         check  ==  num < i
         if rest = 0 then
            ◊ spec
         else
            check  ==  rest :: N_(i-num)
            ◊ spec
         fi
      else
         if spec = unknown then
            restyp  == rest :: N
            ◊ N
         else
            restyp  ==  rest :: unknown
            restyp  ==  restyp = N_i
            ◊ N_(i+num)
         fi
      fi
   fi
```

Ncase Expression

```
if exp ~ ncase (c, alt_0, ...,alt_(n-1)) then

   checkcrit  ==  c  ::  N_n
   redspec    ==  *spec
   for i := 0 to n-1 do
       if alt_i ~ alt ((x : A) e) then
          checkA  ==  A = N_n
          checke  ==  e :: redspec        {x : N_n}
       else
          ??
       fi
   od

   ◊ redspec
```

Natural Type

```
if exp ~ N then
   if spec = type then
      ◊ N
   else
      if (spec = unknown) or (*spec ~ U_i) then
         ◊ U_0
      else
         ??
      fi
   fi
```

Natural Loop

```
if exp ~ loop (c, d, (x:A) (y:B) e) then

   cisnat  ==  c  :: N
   checkx  ==  A  =  N
   lptyp   ==  B  :: type                    {x : N}
   checkd  ==  d  :: lptyp [0/x]
   checke  ==  e  :: lptyp [x'/x]            {x : N,
                                               y : lptyp}
   ◊ lptyp [c / x]   spec
```

Equality Type

```
if exp ~ Eq (A, a, b) then

   checkA  ==  A :: type
   checka  ==  a :: A
   checkb  ==  b :: A

   if spec = type then
      ◊ exp
   else
      if (spec = unknown) or
         (*spec ~ U_i) then
         univA  ==  A  :: spec
         ◊ univA
      else
         ??
```

```
    fi
```

Equality Element

```
if exp ~ eq_proof then

   if (spec = type) or (spec = unknown) then
      ??
   else
      redspec  ==  *spec

      if redspec ~ Eq (A, a, b) then
         checkA  ==  A  :: type
         checkeq ==  a = b
         ◊ redspec
      else
         ??
      fi
   fi
```

Universes

```
if exp ~ U_n  then
   bigU  ==  if spec ~ U_i  then
                ◊ U_i
             else
                if spec = unknown then
                   ◊ U_n+1
                else
                   if spec = type then
                      ◊ U_n
                   else
                      ??
                   fi
                fi
             fi

   checkU  == n < i
   ◊ bigU
```

Chapter 6

Compiling Constructive Logic

This chapter describes the source to source transformation scheme developed to translate Constructive logic into LISP code.

6.1 Source to Source Transformations

The process of translating programs written in a given langage (the source language) into programs written in another language (the target language) is called a source to source transformation. The term 'compilation' corresponds to the mapping of high-level to low-level programs. The translation of ITT constructs into LISP cannot be called a compilation because ITT types are not representable as LISP code. The transformation process make all type-related information inaccessible, lessening the expressive power of the resulting code. However, types are specifications in ITT, and a computable specification must be evaluated during verification: once the program is verified, there is no need for types since these are typically used for run-time checks.

The orthogonal structure of LISP objects (both data and

programs are represented by lists) and the 'quote' feature, which allows explicitly unevaluated programs to be manipulated, makes LISP an ideal candidate for the implementation of a program transformer [BM80].

6.2 Mechanism for Lazy Evaluation

As seen in Chapter 3, the ITT system uses normal order reduction to evaluate expressions. Since the LISP system relies on applicative order, the required reduction order must be enforced artificially to preserve meaning in the translation from ITT to LISP. To simulate normal order reduction, the evaluation of LISP expressions is postponed. This is similar to the call-by-name parameter passing mechanism of ALGOL, with the important difference that an expression's normal form is only computed once. This evaluation mechanism is aptly named call-by-need or lazy evaluation.

Delaying the evaluation of a LISP expression is done by embedding it in a parameterless function: (lambda () (expression)). To evaluate the expression, the function is called with no arguments, i.e.,

((lambda () (expression))) =(expression),

the LISP system evaluates the result in the usual fashion.

A special data structure called a SLOTH is defined as

(defrec /SLOTH DONE ACTION)

where DONE is a flag indicating whether ACTION is in canonical form. A LISP macro DELAY builds SLOTHs out of arbitrary expressions , e.g.

(delay '(add 1 2)) = [sloth nil PROC].

where PROC indicates a LISP function with the corresponding code. The SLOTHs are evaluated with the functions URGE and FORCE. URGE is a function which evaluates a SLOTHed expression one level at a time, setting the corresponding DONE flag to 't.

(urge '[SLOTH nil [SLOTH nil (lambda () exp)]])
= [SLOTH nil (lambda () exp)]

The function FORCE takes a SLOTH and applies URGE to it until the result is no longer delayed.

```
(define force /lambda (exp)
   /if (is sloth exp)
         (force /urge exp)
         exp
)
```

It may be necessary to apply URGE several times to find the normal form of the expression since it is common to delay

expressions several times before evaluating them (see Translation Rules).

6.3 Logic to LISP Translation Rules

When a declaration is verified, the assumption list has an entry of the form '[type name value]' where name is the newly defined identifier. Since no run-time checks are performed, only the value is relevant during translation. The definition is installed in the LISP environment by

```
(EVGLOBAL '/definem NAME
                  /delay /ITT_TO_LISP VALUE
)
```

where ITT_TO_LISP is the function performing the source-to-source transformations. The translation rules are of the general form:

```
[ITT construct]
--->
   (equivalent LISP code).
```

The LISP code is quoted (unevaluated) if the symbols used are lower-case letters. Upper-case letters are used to denote code evalutated during translation, e.g.

```
[let var expression
     stuff
]

--->

   (let var (delay /ITT_TO_LISP expression)
        (force /ITT_TO_LISP stuff)
   )
```

All lists are unevaluated in the generated code. In the above example, 'ITT_TO_LISP expression' is a call to the translation routine with the ITT construct 'expression'. The lower-case letters indicate the code actually produced by the translation process. If any auxilliary definitions or computations are needed during the translation of any construct, these are indicated by binding a lexically scoped identifier with the desired value, as in

```
hello_world == (ITT_TO_LISP '[succ [zero]]).
```

6.31 Variables, Constants and Types

ITT variables are simply translated directly into LISP identifiers since both ITT and LISP enforce lexical scoping. Since the only constants in ITT are natural numbers of the form

[succ [succ [succ ... [zero]]]],

it is a simple matter to transform these into LISP integers. ITT types do not survive the translation process: all

type-valued expressions are translated into 'nil. The lexical scoping mechanism introduced by the let expression is translated by the rule:

```
[let value [abstract [binding type variable]
                     expression]
]
--->
   (let variable (delay ITT_TO_LISP value)
        (force /ITT_TO_LISP expression))
```

6.32 Functions

The ITT function object has an obvious translation:

```
[lambda [abstract [binding type x]
                  body]
]
--->
   (lambda (x) (ITT_TO_LISP body)).
```

Functional application (function calling) also benefits from a rather trivial translation rule:

```
[apply function argument]
--->
   ((force /ITT_TO_LISP function)
        (delay /ITT_TO_LISP argument)
   )
```

The function is FORCEd during the applicative call, but it is not necessary to evaluate the argument.

6.33 Tuples

Rules for tuple objects and the tuple evaluator 'split' follow.

```
[tuple first second]
--->
   (cons (ITT_TO_LISP first) (ITT_TO_LISP second))
```

The list is the natural translation of the tuple construct since the tuple operators 'head' and 'tail' are direct equivalents to car and cdr.

```
[split tuple [abstract [binding xtype x]
              [abstract [binding ytype y]
                        expression]]
]
--->
   LISP_PAIR  ==  (force /ITT_TO_LISP tuple)

   (let x (delay /car LISP_PAIR)
        /let y (delay /cdr LISP_PAIR)
             /ITT_TO_LISP expression
   )
```

The line 'LISP_PAIR == (ITT_TO_LISP tuple)' indicates that this is performed at translation time. The combination (delay ("operation" (force object))) is used to postpone the

evaluation of object.

6.34 Free Unions

The disjunction objects of ITT use special alternative constructs which are abstractions (in the sense of chapter 3). The rules of translation are:

```
[first alternative]
--->
   (cons nil /ITT_TO_LISP alternative)
```

and

```
[second alternative]
--->
   (cons t /ITT_TO_LISP alternative).
```

The disjunction evaluator (case expression) becomes:

```
[case criterion alter1 alter2]
--->
   CRIT_VAR  ==  (GENSYM)
   (let CRIT_VAR (delay /ITT_TO_LISP criterion)
        /if (car /force CRIT_VAR)
              (TRANSLATE_ALT CRIT_VAR alter1)
              /TRANSLATE_ALT CRIT_VAR alter2
   )
```

where (GENSYM) returns a unique atom and the function TRANSLATE_ALT accepts an alternative and builds

corresponding LISP code.

```
(define translate_alt /lambda (crit alt)
  /match alt (make_alt (make_binding x var) exp)
      (let lisp_exp (itt_to_lisp exp)
           /if (not /freein var exp)
               (list 'delay lisp_exp)
               /list 'delay
                     /list 'let var
                           (list 'cdr /list 'force crit)
                           lisp_exp
      )
      /signal_error
)
```

6.35 Finite Types

The ncase expression of ITT uses the same alternative construct as the disjunction, and translate_alt plays an important role here.

```
[ncase criterion alternatives]
--->
   CRIT_VAR  == (GENSYM)
   (let CRIT_VAR (delay /ITT_TO_LISP criterion)
        /cxr (force CRIT_VAR)
             /hunk (TRANSLATE_ALT CRIT_VAR alt0)
                   (TRANSLATE_ALT CRIT_VAR alt1)

                   /TRANSLATE_ALT CRIT_VAR altn
   )
```

This is a similar process to the simple disjunction case: each alternative is translated into LISP code and stored

into the hunk. Executing an alternative of the ncase expression simply involves indexing the hunk with the value of the criterion. There are also the simple special cases where the number of alternatives is 0 or 1.

6.36 Successor Function on Naturals

A call to the successor function 'succ' is translated using the same transformation as a natural constant. The code for this transformation is given below:

```
(definem check_for_num /lambda (exp_count)
    /let (exp count) exp_count

    /match exp (make_zero)
         (hunk nil count)

    /match exp (make_succ argument)
         (check_for_num /hunk argument
                        /add 1 count)

    /signal_error
)
```

The function check_for_num accepts a hunk of size 2, where the first field is an ITT expression, and the second is an integer (initialized to 0). The line 'let (exp count) exp_count' binds the names exp and count with the first and second fields of exp_count. If the zero construct is found during translation, then check_for_num returns [nil count], indicating that the expression has been translated completely. If this is the case, the function was supplied

with a constant, and count is the LISP integer form of the constant. If the expression is of the form [succ exp], then translation continues and the number of recursive calls is accumulated. The translation rule for the successor function then becomes:

```
[succ argument]
--->
    LET (REST INCR) == (CHECK_FOR_NUM /HUNK expression 0)
    IF (NULL REST)
        INCR
        (add INCR /force /ITT_TO_LISP rest).
```

6.37 Loop Construct

This ITT construct is the most interesting one to translate into LISP since it permits certain optimizations to be performed. Because there is no GOTO function in our dialect of LISP, a loop corresponds to a local tail-recursive function. The general translation of the loop is

```
[loop count first [abstract [binding nat index]
                  [abstract [binding restype result]
                            next_val]]
]

--->

  (delay (labels ((/LOOPNAME
                     /lambda (BOUNDVAR)
                        /TESTFORZERO
                           /let index FORCECOUNT
                           /let result (LOOPNAME index)
                                LISPNEXT
                  )

                  /LOOPNAME /delay /ITT_TO_LISP count
         ).
```

Since types disappear during the translation process, restype is not used. The row of the truth-table representing the status (free or bound) of index and result in the next_val expression indicates the applicable translation rule.

FREEIN index	next_val result		
0	0	--->	(if (= count 0) first next_val)
0	1	--->	full loop without index references
1	0	--->	(if (= count 0) first next_val [(count-1)/index])
1	1	--->	full loop

The loop translation algorithm is described incrementally. If count is zero, then no loop is needed and the LISP code is simply (ITT_TO_LISP first). If count is not obviously equal to 0, the status of index and result is established.

```
INDEXUSED    == (FREEIN index next_val)
RESULTUSED   == (FREEIN result next_val)
```

As can be seen from the table, if the result is not used then no recursive function should be generated. This rule controls the value of BOUNDVAR, the potential formal parameter of the loop.

```
BOUNDVAR  ==  (IF RESULTUSED (GENSYM) count)
```

The generated code segment which corresponds to the first Natural Equality rule (test for zero) is:

```
TESTFORZERO  (if (int,eq 0
                         /force /ITT_TO_LISP BOUNDVAR)
                 (delay /ITT_TO_LISP first)
```

When index is used, all occurrences of [succ index] in next_val are replaced by the formal parameter to the loop, thereby avoiding adding 1 only to subtract 1 right away. The following code segment also handles changing [succ index] to 'count - 1' when result is not used.

```
INDEXUSED == (AND INDEXUSED
                  /PROGN (SUBST BOUNDVAR
                                (MAKE_SUCC index)
                                next_val)

                         /FREEIN index next_val
             )
```

If result is not used, the translation will be a simple 'if'

expression.

```
(APPEND TESTFORZERO
        //delay
            /IF (NOT INDEXUSED)
                (ITT_TO_LISP next_val)
                ,/ITT_TO_LISP /SUBST
                                    (MAKE_PRED count)
                                    index
                                    next_val
)
```

Here MAKE_PRED is a predecessor function on naturals, which is well-defined since count is greater than 0. Since the loop is a function, a unique name is generated. The following declarations are needed.

```
LOOPNAME == (GENSYM)
DECCOUNT == (sub (force BOUNDVAR) 1)
LISPNEXT == (delay /ITT_TO_LISP next_val)
```

Finally, the full loop is built by the following code segment.

```
(labels (/LOOPNAME
            /lambda (BOUNDVAR)

            /APPEND TESTFORZERO

                //IF INDEXUSED
                        (let index FORCECOUNT
                        /let result (LOOPNAME index)
                            LISPNEXT
                        )

                        /let result (LOOPNAME DECCOUNT)
                            LISPNEXT
        )
        /LOOPNAME
            /delay /ITT_TO_LISP count
)
```

6.3 A Sample Translation

A short example is used to demonstrate the translation process. Consider the addition function, written in Brouwer.

```
function (A : natural)
    function (B : natural)
        loop to B giving (SUM : natural)
            zero +-> A,
            succ I +-> (SUM)'
        pool
    end;
```

This function is (rather trivially) translated into intermediate ITT equivalent code:

```
[lambda [abstract [binding Nat A]
        [lambda [abstract [binding Nat B]
                [loop B A [abstract [binding Nat I]
                          [abstract [binding Nat SUM]
                                    [succ SUM]]]]
        ]]]
].
```

Since translation into LISP proceeds inwards from the outermost brackets, the first rule to be used pertains to the function object. This rule is applied to each function.

```
(lambda (A)
    /ITT_TO_LISP [lambda . . . ]
)
--->
```

```
(lambda (A)
    (lambda (B)
        /ITT_TO_LISP [loop . . . ]
)
```

The next rule to be invoked performs loop translation. Checking for free occurrences of A and B in [succ SUM] is done. The bound variable of the loop and the test for zero are generated.

```
INDEXUSED    == nil
RESULTUSED   == t
BOUNDVAR     == q_4532

TESTFORZERO == (if (int_eq 0 /force q_4532)
                        (delay A)
                )
```

The next step involves generating the loop name, the count decrement and the translation of [succ SUM].

```
LOOPNAME   == q_4533
DECCOUNT   == (sub (force q_4533) 1)
LISPNEXT   == (delay /add 1 /force sum)
```

The LISP version of the loop in theform:

```
(labels (/q_4533
            /lambda (q_4533)
                /if (int_eq 0 /force q_4532)
                        (delay A)
                        /let result (q_4533
                                        /sub (force q_4532) 1)

                                /delay /add 1 /force SUM
        )
        /q_4533 /delay B
)
```

Chapter 7

Conclusion

It has been demonstrated that the Brouwer Program Verifier described in this thesis properly checks for total correctness of applicative programs. Clearly, significant implications include the attainment of improved program reliability and a close mapping of computer programs and their underlying mathematical theory. Formal verification of programs has been the focus of intense research efforts in recent years, and several programming languages have been developed with the aim of proving programs (ADA [Ada81], ALPHARD [ALP81]).

The Brouwer Program Verifier has Intuitionistic Type Theory as its mathematical foundation. The task of verifying total correctness is therefore much simplified because the underlying formal system provides great computational versatility without the use of non-terminating constructs.

Brouwer is a prototypical system and certainly cannot compete with established verifiers like the Stanford Pascal Verifier, but it proves a far stronger property, namely that any correct program terminates. Systems based on ITT or

variants thereof include the PRL system at Cornell, which is an interactive specification system, and the proof construction system of the Programming Methodology Group at Goteborg.

While PRL is supported by fast symbolic processors (such as the LISP machine), the current version does not include all the types of ITT. In particular, PRL does not support high-order functions.

The system implemented by the PM group at Goteborg [Pet83] supports all of ITT's types, except well-orderings, but the type theoretical notation is used to formulate theorems and proofs. Their proof of the property of associativity of addition is more than three times as long as the corresponding proof in Brouwer. The reason is that no system-wide verification strategies are built into their system. The proof writer must spell out each and every step and the system simply informs him of illegal derivations.

In Brouwer, the verifier attempts to prove the program or theorem and accepts hints from the programmer at strategic points. Once the proof is written, the verification process is completely automatic.

This is not to say that Brouwer is the definitive verification system. The proof tactics employed are

appropriate to an experimental system but need to be refined before the system will be practical for large scale software development.

In contrast to most other verification systems, Brouwer has a single notation for specifications, programs and proofs, thereby reducing the effort necessary to master all aspects of the system. Furthermore, the language's effective proof strategies minimize the amount of time spent coding specifications and proofs. Since the type theory is computational in nature, the potential for efficient execution of Brouwer programs is excellent. However, this would require a compiler specifically tuned to the unusual mix of high- and low-level constructs in the language.

The VML (Verification Meta-Language) has proven to be a useful tool for the description of verification rules. It is probable that the main concepts of VML would be found useful in describing other formal systems. This would provide a fruitful area for further research.

A complete verification system should include an extensible library of pre-verified modules and an interactive proof checker. A sophisticated version control system is necessary to guarantee integrity of programs throughout the software life cycle. One solution is to maintain a dependency graph relating library modules. This

graph would be used to determine when and where re-verification is necessary.

It has been suggested that a careful description of the hardware functions in Brouwer would enable the verification of certain assembly programs.

Another possibility is a hardware ITT machine. Lazy evaluation would be implemented by graph reduction algorithms written in microcode. Of course, imperative features would support the input and output operations. A similar machine has been developed for Turner's [Tur79] combinators: the SKIM machine [SKI80].

Bibliography

Abbreviations

- LNCS - Lecture Notes in Computer Science, Springer-Verlag.
- ACM - Association for Computing Machinery.
- CACM - Communications of the ACM.
- JACM - Journal of the ACM.

[Ada81] Ada Reference, 'The Programming Language Ada, Reference Manual', Proposed Standard Document, United States Department of Defense, LNCS 106, 1981.

[ALP81] ALPHARD, 'ALPHARD: Form and Content', Edited by Mary Shaw, Springer-Verlag, New York, Heidelberg, Berlin, 1981.

[AU79] Aho, A.V., Ullman, J.D., 'Principles of Compiler Design', Addison-Wesley Series in Computer Science and Information Processing, 1979.

[Bac78] Backus, J., 'Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs', CACM Vol. 21, No. 8, pp. 613-641, 1978.

[Bis67] Bishop, E., 'Foundations of Constructive Analysis', McGraw Hill, New York, 1967.

[BG80] Burstall, R.M. and Goguen, J.A. 'The Semantics of CLEAR, A Specification Language', LNCS 86, New York, 1980.

[BM80] Boyle, J.M. and Muralidharan, M.N. 'Program Reusability through Program Transformation', Proc. of the Workshop on Reusability in Programming, T. Biggerstaff, Editor, 1980.

[BoM79] Boyer, R.S. and Moore, J.S., 'A Computational Logic', Academic Press, New York, 1979.

[Boo82] Boom, H.J., 'Programming in the Logic of Types', CIPS Session '82 Proceedings, Saskatoon, pp. 109-117, 1982.

[Boo84] Boom, H.J., 'Can a verifier verify itself? or Learn to live with partial correctness', Technical Report PLSG-1, Dept. of Computer Science, Concordia University, 1984.

[Bro07] Brouwer, L.E.J., 'On the Foundations of Mathematics', PhD Thesis, University of Amsterdam, 1907.

[Bur75] Burge, W.H., 'Recursive Programming Techniques', Addison-Wesley, Reading, Massachusetts, 1975.

[Can55] Cantor, G., 'Contributions to the Founding of the Theory of Transfinite Numbers', Dover Publications, Inc. N.Y., 1955.

[CB83] Constable, R.L. and Bates, J., 'Proofs as Programs', Technical Report No: 82-530, Department of Computer Science, Cornell University, Ithaca, New York, 1983.

[CB84] Constable, R.L. and Bates J., 'The Nearly Ultimate PEARL', Technical Report No. 83-551, Department of Computer Science, Cornell University, Ithaca, New York, 1984.

[CF58] Curry, H.B. and Feys, R., 'Combinatory Logic, Vol. I', North-Holland, Amsterdam, 1958.

[Chu51] Church, A. 'The calculi of Lambda-Conversion', Annals of Mathematical Studies 6, Princeton University Press, Princeton, 1951.

[CJE82] Constable, R.L., Johnson, S.D. and Eichenlaub, C.D., 'An Introduction to the PL/CV2 Programming Logic', LNCS 135, Berlin, 1982.

[CLU81] Liskov, B., Atkinson, R., Bloom T., Moss, E., Schaffert, J.C., Scheifler, R. and Snyder, A., 'CLU Reference Manual', LNCS 114, 1981.

[CZ84] Constable, R.L. and Zlatin, D.R., 'The Type Theory of PL/CV3', ACM Transactions on Programming Languages and Systems, pp. 94-117, Vol. 6, No. 1, January 1984.

[DD80] Demers, A. and Donahue, J., '"Type-Completeness" as a Language Principle', Principles of Programming Languages, 1980.

[deB70] de Bruijn, N.G., 'The mathematical language AUTOMATH, its usage, and some extensions', Symposium on Automatic Demonstration, Lecture Notes in Mathematics, Vol. 125, Springer-Verlag, Berlin, pp. 29-61, 1970.

[Dij76] Dijkstra, E.W., 'A Discipline of Programming', Prentice-Hall, Englewood Cliffs, 1976.

[Dyb83] Dybjer, P., 'Semantics and Specification - A Short Introduction', Proc. Declarative Programming Workshop, University College, London, pp. 140-145, 1983.

[Flo67] Floyd, R.W., 'Assigning Meanings to Programs', Proc. Symposium on Applied Mathematics 19, J.T. Schwartz, ed., Mathematical Aspects of Computer Science, pp. 19-32, American Mathematical Society, 1967.

[Gar81] Garman, J.R., 'The "bug" heard "round the world', ACM SIGSOFT Software Engineering Notes, Vol. 6, No. 5, 1981.

[Gor79] Gordon, M.J.C., 'The Denotational Description of Programming Languages, an Introduction', Springer-Verlag, Berlin, 1979.

[Gut77] Guttag, J.,'Abstract Data Types and the Development of Data Structures',CACM Vol. 20, No. 6, pp. 396-404, 1977.

[Hen80] Henderson, P., 'Functional Programming, Application and Implementation', International Series in Computer Science, Prentice-Hall, C.A.R. Hoare, Series Editor, 1980.

[Hey56] Heyting, A., 'Intuitionism, an Introduction', North-Holland, 1956.

[Hil25] Hilbert, D., 'On the Infinite', Mathematische Annalen, Vol. 95, pp. 161-190, 1925.

[Hoa69] Hoare, C.A.R., 'An Axiomatic Basis for Computer Programming', CACM, Vol. 12, No. 10, 1969.

[How76] Howden, W.E., 'Methodology for Generation of Program Test Data', IEEE Trans. on Software Engineering, Vol. 2, No. 3, 1976.

[How80] Howard, W.A., 'The Formulae-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.), Academic Press, London, pp. 479-490, 1980.

[Jer77] Jervell, H.R., 'Constructive Universes I', Proc. Conference on Higher Set Theory, G.H. Muller and D.S. Scott, editors, Lecture Notes in Mathematics 669, pp. 73-98, 1977.

[Joh83] Johnsson, T., 'The G-Machine: An Abstract Machine for Graph Reduction', Proc. Declarative Programming Workshop, University College, London, pp. 1-18, 1983.

[JW78] Jensen, K. and Wirth, N., 'Pascal User Manual and Report', 2nd ed., Springer-Verlag, New York, 1978.

[Lan55] Landau, E., 'Foundations of Analysis', 3rd ed., Chelsea Pub. Co., New York, NY, 1955.

[Lan66] Landin, P.J., 'The Next 700 Programming Languages', CACM Vol. 9, No. 3 , 1966.

[LCF79] Gordon M., Milner R., Wadsworth C., 'Edingburg LCF', LNCS 78, Berlin, 1979.

[Kli80] Kline, M., 'Mathematics: the Loss of Certainty', Oxford University Press, Oxford, 1980.

[McC60] McCarthy, J., 'Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I', CACM Vol. 3, 1960.

[McC80] McCarthy, J., Abrahams, P.W., Edwards, D.J., Hart, T.P. and Levin, M.I., 'LISP 1.5 Programmer's Manual', 2nd ed., M.I.T. Press, Cambridge, Massachusetts, 1980.

[McP67] McCarthy, J. and Painter, J.A., 'Correctness of a Compiler for Arithmetic Expressions', Mathematical Aspects of Computer Science, J.T. Schwartz, ed., American Mathematical Society, 1967.

[ML73] Martin-Lof, P., 'An Intuitionistic Theory of Types: Predicative Part', Logic Colloquium, H.E. Rose and J.C. Sheperdson (eds.), pp. 73-118, 1973.

[ML79] Martin-Lof, P., 'Constructive Mathematics and Computer Programming', Sixth International Congress for Logic, Methodology and Philosophy of Science, Hannover, August 1979.

[MMN75] Milner, R., Morris, L. and Newey, M., 'A Logic for Computable Functions with Reflexive and Polymorphic Types', Proc. Conference on Proving and Improving Programs, 1975.

[MP74] Manna, Z. and Pnueli, A., 'Axiomatic Approach to Total Correctness of Programs', ACTA Informatica, No. 3, 1974.

[Nor81] Nordstrom, B., 'Programming in Constructive Set Theory: Some Examples', Proc. Conference on Functional Languages and Computer Architecture, 1981.

[NP83] Nordstrom, B. and Petersson, K., 'Types and Specifications', Working Group 2.1, Munich, 1983.

[NS83] Nordstrom, B. and Smith, J., 'Why Type Theory for Programming? A Short Introduction', Proc. Declarative Programming Workshop, University College, London, pp. 21-43, 1983.

[Pet83] Petersson, K., 'An Introduction to the Programming System for Type Theory', Proc. Declarative Programming Workshop, University College, London, pp. 44-65, 1983.

[Pol81] Polak, W., 'Compiler Specification and Verification', LNCS 124, 1981.

[RA83] Reps, T. and Alpern, B., 'Interactive Proof Checking', Proc. Symposium on Principles of Programming Languages, January 1984.

[RW10] Russell, B., Whitehead, A.N., 'Principia Mathematica', 3 vols., Cambridge University Press, N.Y., 1st ed., 1910-1913, 2nd ed., 1925-1927.

[Sc70a] Scott, D., 'Outline of a Mathematical Theory of Computation', Proc. Fourth Annual Princeton Conference on Information Sciences and Systems, 1970.

[Sc70b] Scott, D., 'The Lattice of Flow Diagrams', Semantics of Algorithmic Languages, Springer Lecture Notes Series, Springer-Verlag, Berlin, 1970.

[SKI80] Clarke, T.J.W., Gladstone, P.J.S., Maclean, C.D., Norman, A.C., 'SKIM - The S, K, I Reduction Machine', Proc. LISP Conference, 1980.

[Sok77] Sokolowski, S., 'Axioms for Total Correctness', Acta Informatica, Vol. 9, No. 1, 1977.

[SS71] Scott, D. and Strachey, C., 'Toward a Mathematical Semantics for Computer Languages', Proc. of the Symposium on Computers and Automata, Microwave Research Institute Symposia Series, Vol. 21, Polytechnic Institute of Brooklyn, New York, 1971.

[Sto77] Stoy, J., 'Denotational Semantics - The Scott-Strachey Approach to Programming Language Theory', MIT Press, Cambridge, 1977.

[Tur79] Turner, D.A., 'A New Implementation Technique for Applicative Languages', Software Practice and Experience, pp. 31-49, 1979.

[vaD73] van Daalen, D.T., 'A description of AUTOMATH and some aspects of its language theory', Proc. Symposium on APL, P. Braffort, ed., Paris, 1973.

[VAX81] Digital Equipment Corp., 'VAX Architecture Handbook', DEC Publishing, 1981.

[WLS76] Wulf, W.A., London, R.L. and Shaw, M., 'An Introduction to the Construction and Verification of Alphard Programs', IEEE Trans. on Software Engineering, Vol. 2, No. 4, 1976.

Appendix A

LITHP: the Implementation Language

The LITHP language differs from standard LISP [McC80] in several ways. One important notational difference is the use of '/' to reduce the number of parentheses in a LITHP expression. The '/' character matches the closest unbalanced right parenthesis, and generates a new right parenthesis, e.g.

```
'(hello /out /there (world))
```

is equivalent to

```
'(hello (out (there (world))))
```

There are new data types and facilities: strings, array-like constructs, saved environments and macros. LITHP strings are denoted by

```
%string_expr
```

where string_expr is a sequence of ASCII characters. Certain characters are special (' ', '.', '&', etc.) and must be preceded by '&'. Various string-handling functions

are provided: predicates like str_eq, str_lt and str_gt, conversion functions like char_to_int and int_to_char, and several miscellaneous utilities.

An important feature is the concept of a hunk: a data structure whose size is fixed at definition time. The application (new_hunk n) returns a fresh hunk with n fields (LITHP pointers), indexed by values in the range 0 .. n-1. Selecting a particular component of a hunk is performed with the function cxr, e.g. (cxr 0 /hunk 'a 'b) = a. Hunks are ideal for hash tables, returning multiple values from functions and as jump tables for semantic actions.

The Brouwer semantic action routines are defined as a hunk containing LITHP code, with as many fields as there are productions in the Brouwer grammar. If the hunk SEM_ACTIONS is defined as

```
(define SEM_ACTIONS
    /hunk (lambda () action0)
          (lambda () action1)

          (lambda () actionN)
)
```

then given a production number in the range 0 to N-1, the expression ((cxr PROD_NUMBER SEM_ACTIONS)) performs the desired semantic action.

There exists an if function in LITHP which has the

following form:

```
(if condition
      true_part
      false_part
)
```

where true_part is evaluated if condition is not nil; otherwise false_part is evaluated.

The LITHP dialect uses lexical scoping rules to control the visibility of variables. The let function binds a local name to a value and makes this name visible in the let body:

```
(let (var type)  (check_type exp)
      (..... var .... type)
)
```

The function check_type must return a hunk of size 2. In the let's body, var is bound to (cxr 0 /check_type exp) and type is bound to (cxr 1 /check_type exp).

There is no GOTO function in LITHP, but it is possible to define local recursive functions with LABEL and LABELS.

```
(labels ((factorial /lambda (n)
                 ( /if (int_eq n 1) 1
                       /mul n /factorial /sub n 1
                 ))
)
```

The LABELS function is typically used to define local

mutually recursive functions.

Macros can be defined with the function DEFMAC and particularly useful ones are DEFREC and MATCH. The DEFREC macro builds a hunk whose first component is a tag, along with several functions:

```
(defrec (name slot1 slot2 ...) action).
```

This results in the construction of a function MAKE_name which can produce data structures with the required components, as well as a predicate function IS_name and accessing functions name_slot1, name_slot2, etc. Component values can be set by using functions like set_name_slot1. The action expression is used to describe any special storing mechanism used when MAKE_name is invoked.

The MATCH macro is a pattern matcher which can be used with all LITHP objects.

```
(match greetings (cons hello goodbye)
     (true_exp)
     (false_exp)
)
```

An object matches the pattern iff it has the same structure. This allows us to bind local names with sub-expressions, and use these names in the true_exp. Obviously, these names are not defined if the supplied object does not match the

pattern. In the example given, if greetings was defined as '(1 2 3 4), then hello would have value 1 and goodbye would have value (2 3 4) in true_exp.

An interesting feature of the LITHP system is that of saved worlds. The state of the system can be saved on a file, and the session continued at a later date. All information (including stack contents) is saved and can be restored, using the functions SAVE_WORLD and RESTORE_WORLD. The format of the save file depends on the size of the object code in the LITHP system: any change in the size of the executable image invalidates all previously saved worlds. Since installing the Brouwer system takes 40 minutes of CPU time on the average, it is highly desirable that the LITHP system be stable.

Another interesting feature (installed by the author) allows the LITHP system to perform any operating system command. A call to the operating system is disguised as a LITHP function, and a subprocess is spawned to run the VMS command interpreter.

The LITHP system is currently supported by an interpreter, and a compiler is being developed. We expect a large increase in the verifier's speed when compilation becomes available.

Appendix B

ITT Term Equality Rules

The rules used to decide whether two ITT expressions are equal may be categorized as: general rules, substitution rules and object equality rules.

General Rules

Only the general rules for elements are given: the rule for the types are similar.

I. Symmetry

$$\frac{a = b : A}{b = a : A}$$

II. Transitivity

$$\frac{a = b : A \qquad b = c : A}{a = c : A}$$

III. Equality of Types

$$\frac{a : A \qquad A = B}{a : B}$$

Substitution Rules

$$\frac{a = c : A \qquad [x : A] \quad B = D}{B\,[a / x] \;=\; D\,[c / x]}$$

$$\frac{a = c : A \qquad [x : A] \quad b = d : B}{b\,[a/x] \;=\; d\,[c/x] \;:\; B\,[a/x]}$$

Object Equality Rules

The general and substitution rules are sufficient to derive all of the object equality rules, if the requirement that variables be renamed to avoid variable capture is added. For example, the function type equality rule is

$$\frac{A = C \qquad [x : A] \quad B = D}{(x{:}A) \to B \;=\; (x{:}C) \to D}$$

and the function introduction equality is:

$$\frac{[x : A] \quad b = d : B}{\lambda(x{:}A).b \;==\; \lambda(x{:}A).d \;:\; (x{:}A) \to B}$$

Appendix C

Description of Brouwer Syntax

This notation for syntax description is directly accepted as input by Dr. Boom's parser general ANAL. The following conventions are observed:

- Symbols appearing on the left-hand side of the colon (which must be non-terminals) are defined by the comma-separated sequences appearing on the right, the production being terminated with a period. The null production therefore appears as an isolated dot.

- Alternatives always appear as separate productions.

- Symbols enclosed in quotation marks are terminals as supplied by the lexical scanner.

```
-- Module Declaration Facility

  START       : MODULE_LIST, ".".
  MODULE_LIST : MODULE_LIST, MODULE.
  MODULE_LIST : MODULE.

  MODULE      : MODULE_HEAD, IMPORTS, EXPORTS, DECLARATIONS
                , PROPERTIES, "eludom".

-- Module Header (possibly with type).

  MODULE_HEAD : "MODULE", TYPE_OR_ID.
  TYPE_OR_ID  : TYPED_ID.
```

```
TYPE_OR_ID   : IDENT.

-- Declarations, with Imports and Exports.

IMPORTS  : "import", IDENT_LIST.
IMPORTS  : .

EXPORTS  : "export", IDENT_LIST.
EXPORTS  : .

-- Possible Module Declarations.

DECLARATIONS : DECLARATIONS, DECLARATION.
DECLARATIONS : DECLARATION.

DECLARATION  : MODULE.
DECLARATION  : "define", TYPED_ID,
               DECL_EXPR.

DECL_EXPR    : " ==", EXPRESSION.
DECL_EXPR    : .

-- Module Property Lists

PROPERTIES   : "property", PROP_LIST.
PROPERTIES   : .

PROP_LIST    : PROPERTY, ";", PROP_LIST.
PROP_LIST    : PROPERTY.

PROPERTY     : "prove", TYPED_ID, DECL_EXPR.
```

```
-- Basic Primitive Types.

P_TYPE : "void".
P_TYPE : "unity".
P_TYPE : "boolean".
P_TYPE : "scalar", "(", IDENT_LIST, ")".
P_TYPE : "scalar", CONSTANT.
P_TYPE : "natural".
P_TYPE : "universe", CONSTANT.

-- Type Constructors (with priority And > Or > Func.)

TYPE     : TYPE, "=", FUN_TYPE.
TYPE     : FUN_TYPE.

FUN_TYPE : FUN_TYPE, "->", OR_TYPE. -- Function Type
FUN_TYPE : OR_TYPE.
OR_TYPE  : OR_TYPE, "|", AND_TYPE.  -- Free Union
```

```
OR_TYPE  : AND_TYPE.
AND_TYPE : AND_TYPE, "&", DEF_TYPE. -- Records
AND_TYPE : DEF_TYPE.

DEF_TYPE : "(", IDENT, " ", OTH_TYPE, ")".
DEF_TYPE : OTH_TYPE.

OTH_TYPE : P_TYPE.       -- Primitive Types.
OTH_TYPE : SIM_EXPR.     -- Other Type Expressions
```

```
-- Brouwer Objects.

   -- Local Definitions with Lexical Scoping.

   LET_STAT  : "let", TYPED_ID, " ==",
               EXPRESSION, "in", EXPRESSION,
               "end".

   -- Function Object and Reference.

   FUNCTION  : "function", "(", FUN_ARGS, ")",
               EXPRESSION, "end".

   FUN_ARGS  : FUN_ARGS, ";", ARGUMENT.
   FUN_ARGS  : ARGUMENT.
   ARGUMENT  : IDENT_LIST, " ", TYPE.
   FUNC_REF  : FACTOR, "(", EXPRESSION, ")".

   -- Tuple Object and Reference.

   TUPLE     : "[", TUP_LIST, "]".
   TUP_LIST  : TUP_LIST, ",", EXPRESSION.
   TUP_LIST  : EXPRESSION.
   TUP_LIST  : .

   TUP_WITH  : "with", EXPRESSION, "do",
               EXPRESSION, "od".

   TUP_SPLIT: "split", EXPRESSION, "into",
               TUPLE, "in", EXPRESSION, "end".
   TUP_REF   : CLOSURE, ".", IDENT.

   -- Disjoint Union Object and Reference.

   CASE      : "case", EXPRESSION, "of",
               CASE_BODY, "esac".

   CASE_BODY : CASE_BODY, ",", CASE_CLAUSE.
   CASE_BODY : CASE_CLAUSE.

   CASE_CLAUSE : TYPE_OR_ID, CASE_EXPR.
```

```
CASE_EXPR   : "+->", EXPRESSION.

IF_EXPR     : "if", EXPRESSION, "then",
              IF_PART, "else", IF_PART, "fi".

IF_PART     : EXPRESSION.

-- Loop with Natural Number as Counter.

LOOP        : LOOP_HEAD, RESULT_DEC, LOOP_BODY.

LOOP_HEAD   : "loop", "to", EXPRESSION.
RESULT_DEC  : "giving", TYPED_ID.
RESULT_DEC  : .

LOOP_BODY   : ZERO_PART, SUCC_PART.
ZERO_PART   : "zero", CASE_EXPR, ",".
SUCC_PART   : "succ", IDENT, CASE_EXPR, "end".
```

```
-- Brouwer Expressions.

EXPRESSION : TYPE.              -- Basic Constructions

SIM_EXPR    : SIM_EXPR, "'".
SIM_EXPR    : TERM.

TERM        : TERM, "^", FACTOR.
TERM        : FACTOR.

FACTOR      : CLOSURE.
FACTOR      : FUNC_REF.
FACTOR      : TUP_REF.

CLOSURE     : "(", EXPRESSION, ")".
CLOSURE     : CONSTANT.
CLOSURE     : IDENT.
CLOSURE     : LOOP.
CLOSURE     : TUPLE.
CLOSURE     : TUP_WITH.
CLOSURE     : TUP_SPLIT.
CLOSURE     : LET_STAT.
CLOSURE     : FUNCTION.
CLOSURE     : IF_EXPR.
CLOSURE     : CASE.

CONSTANT    : "NUMBER".
IDENT       : "IDENTIFIER".
IDENT_LIST  : IDENT_LIST, ",", IDENT.
IDENT_LIST  : IDENT.

TYPED_ID    : IDENT, ":", TYPE.
```