THE TYPE THEORY OF PL/CV3*

Robert L. Constable
Daniel R. Zlatin

TR 81-464
August 1981
Revised March 1983

Department of Computer Science
Cornell University
Ithaca, New York 14853

*Research supported in part by NSF grant MCS-78-00953, in part by the IBM Corporation.
The Type Theory of PL/CV3*

Robert L. Constable
Daniel R. Zlatin

Cornell University

Abstract
The programming logic PL/CV3 is based on the notion of a mathematical type. We present the core of the type theory, from which the full theory for program verification and specification can be derived. Whereas the full theory was designed to be useable, the core theory was selected to be analyzable. This presentation strives to be succinct yet thorough. The last section consists of examples, but the approach here is not tutorial.

Key Words and Phrases: automated logic, program verification, program specification, semantics of programming languages, type theory, foundations of mathematics

CR Categories: 4.29, 5.21, 5.24

1. Introduction

1.1. Logic of programming versus logic of mathematics

We investigate the logic of programming because it will help us understand the programming process and enable us to be better at it. This "self-improvement" motive is not the typical reason that people study the logic of mathematics; and traditional logic is not always a good role model for programming logic. Why is there this difference, and does it matter?

The chief reason for the difference is the computer. Programming is a formal activity unlike the "doing" of most other mathematics. It requires communication with unintelligent machines which do not understand wonderful mathematical solutions to problems. A solution must be programmed, and this means formalized. Although the solution may be a function and hence similar in its form and precision to solutions in analysis and algebra, its description tends to be rather long by comparison with function descriptions in mathematics; say a page or two instead of a line or two.

*Research supported in part by NSF grant MCS-78-00953, in part by the IBM Corporation. This research was presented at the IBM Logic of Programs Conference, May, 1981. Authors' address: Department of Computer Science, 405 Upson Hall, Cornell University, Ithaca, N. Y., 14850.
More significantly, the exacting requirements of formality and the inherently arbitrary nature of programming formalisms make it very difficult for people to check every detail of these long solutions. In addition, features of programming languages introduced to allow efficient execution of programs complicate explanations of the solution by involving eccentricities of a specific formalism.

Confronted with these difficulties, computer scientists have been seeking logics which will explain programming solutions and permit mechanical help in finding them, checking them and changing them. As it becomes more important to know that a program really does what it is claimed to do, i.e. meet its specification, more details of the explanation are made explicit and formalized. The limit of this process is a completely formal proof that the program is correct. The program can be seen as the "executable part" of this proof (see [13]).

The more formal the explanation of a program, the greater the opportunity to use the computer to check or help generate it. This reflexive use of the computer to check itself is one of the most intriguing and promising areas of research into the problems of "software reliability" and into the nature of the programming process itself. These incentives to produce formal algorithmic proofs seem to be much higher than any incentive to do the same for mathematical arguments with no computational content. Nevertheless, one of the earliest and most significant efforts to actually use formal proofs was in the AUTOMATH project at Eindhoven in the Netherlands under the direction of N. G. deBruijn [14]. The goal of the project was to write formal mathematical proofs and check them by a computer program. DeBruijn imposed by desire the standards of extreme exactness that arise by necessity in programming.

The chief characteristic of AUTOMATH and the programming logics is that they are meant to be used; whereas the formal theories presented in logic textbooks are meant to be studied. This difference is crucial and is the driving force behind AUTOMATH as well as PL/CV [8,11] and PRL [2,3], for example. This characteristic means that the logic must be sufficiently expressive that it naturally captures all the informal arguments of the subject being considered. We claim that the V3 described here enjoys such
expressiveness.

1.2. Constructive mathematics and programming

The path of investigation that led to the V3 type theory is described in [9]. We concentrate here on explaining why the various AUTOMATH theories, expressive as they are, were not adopted without change, and in particular, why a very expressive theory such as Zermelo-Fraenkel set theory or some equivalent theory was not used. First let us notice that set theory has been formalized in AUTOMATH (in fact in AUT-68), and that the syntax and semantics of programming languages can be defined smoothly in such a theory [29], with the notion of a data type defined in terms of sets or algebras.

To define the notions of computable function and data type in this way would give meaning to a programming language, but it would also bring along much excess baggage as far as the needs of the programmer are concerned. There would be functions and computable functions, sets and data types, truth and provability. It may seem at first that this dualism is necessary if one is to reason about programs, but that is not so as we explain later.

Another more basic approach to defining a programming logic in AUTOMATH would be to axiomatize the programming concepts directly. All of the main AUTOMATH languages, AUT-68, AUT-QE and AUT-PI offer the typed lambda calculus whose types include the powerful notion of a dependent function space constructor (the \( \Pi \) of section 2.2). So the concepts of computable function and data type are built into AUTOMATH. In AUT-QE it is even possible to abstract over types so that a form of polymorphism is available. In such a "programming AUTOMATH" one could not only naturally write and execute this rich class of programs, but one could express a great deal of mathematics, namely computational mathematics. Indeed, the theory would be a basis for constructive mathematics. Dana Scott noticed exactly this about AUTOMATH, and in his 1970 paper "Constructive Validity" [26] he proposed precisely such a theory as foundation for Intuitionistic mathematics. To accomplish this he enriched the AUTOMATH type structure by adding the disjoint union over a family of types (the \( \Sigma \) operator of 2.2); this addition defines the existential quantifier, it also allows such
notions as the Algol 68 disjoint union. Starting in the same period, Per
Martin-Löf in a series of papers [22,23,24] perfected this approach to
constructive mathematics. He makes the claim in [23] that his theory,
which we call ITT75, can completely and faithfully formalize Bishop's style
of constructive analysis [5].

Bishop's style of constructive mathematics can be seen as the
computational core of classical mathematics. It is possible to recover
full classical theories by adding additional axioms such as "excluded
middle", $A \lor \neg A$. The computational core can be implemented as a
programming language as well as a logic of programming. Such an
implementation and its implications are the subject of the PRL ("pearl")
project at Cornell [2,3] which has produced a working system to investigate
these ideas in practice. There are reasons to believe that this
characteristic of constructive theories will be of fundamental importance
and may alone justify the departure from classical AUTOMATH theories. The
fact that the type theory is so rich means that a separate theory of sets
is unnecessary to express the mathematics of programming.

The initial contributions of the Cornell work on automated programming
logics to this line of development of AUTOMATH-like languages was to employ
concepts and techniques from computer science to render these theories more
useable. This is a goal shared with the Edinburgh LCF work [16]. Another
contribution was to explicitly treat procedural programming concepts [8].
All of this work eventually led us to consider concepts similar to those in
Scott and Martin-Löf's work. The result is the type theory V3 reported on
here which enriches Martin-Löf's theories to deal explicitly with notions
arising in programming. We will confine any precise comparison of our work
to that of Martin-Löf's and not to the AUTOMATH line to which it is closely
related.

The essential difference between our work, V3, and Martin-Löf's ITT
stems from the fact that ITT was designed to formalize mathematics, and V3
was designed to formalize the type theory concepts as they arise in

*Readers interested in further comparisons of AUTOMATH's AUT-QE and the
PL/CV style logics should examine the treatment of number theory in [11]
and [20].
programming. We saw that the ideas of deBruijn, Scott and Martin-Löf would help us solve problems that had arisen in our work on PL/CV2. We will comment on specific instances as the theory is unfolded, but in brief they led to these differences with ITT. V3 treats operations intensionally while ITT does not; V3 uses quotient types to hide information while ITT does not; V3 provides an explicit treatment of recursive data types (not presented here) while ITT does not, and V3 provides induction over the (open ended) universes and over the types while ITT does not.

1.3. Type theory as a programming logic

The PL/CV class of programming logics are all designed to be useable formalisms. This is accomplished with the techniques developed in building programming language translators, editors and environments to make our logical language readable and efficiently generable, see [19]. It is accomplished using decision procedures for equality theory and fragments of other theories such as arithmetic and lists to keep the level of formal detail manageable, see [11]. It is accomplished using a very expressive language such as V3 to provide the level of abstraction encountered when working informally, including the ability to express metamathematical reasoning [12].

We anticipate that further strides will be taken to render these languages useable by accumulating an organized library of knowledge. Such knowledge should include a store of proof tactics expressed perhaps in the style of Edinburgh LCF tactics [16] and incorporating perhaps some of the numerous techniques for proof building discovered and studied by research in automated theorem proving [6]. A great deal of effort will be required before any significant number of research groups can assess the effect of building automated logics on these principles. This paper illustrates some of the logic design issues implied by such a research program, in particular those issues which are very close to questions about programming language design.
2. Type Theory

2.1. Primitive Concepts

To understand the concept of a small type and the type of all small types, one must understand the concept of an inductive definition, the notion of an algorithm, and the notion that the range type of an algorithm can depend on its argument.

The first step in an inductive definition of the collection of types is the specification of a finite number of primitive types which are familiar from informal computing experience. These include the type of Booleans for example, and may include the integers and character strings as well (depending on how parsimonious we wish to be). Part of the specification of these types must include a description of the elements of the type; we must say how to construct them, i.e. represent them in our stock of symbols or computer components. In the case of the integers for example, we might choose decimal numerals as the standard representation. Furthermore we must say when two elements are equal, e.g. we say that 01 and 1 are equal integer values. From these examples, one understands a more abstract concept of type and the notion of what it means to specify a new primitive type.

To specify a type, one tells how to construct the canonical objects of the type, and one gives a condition (in the form of an equivalence relation) telling when two such constructions are to be considered equal.\(^1\)

The canonical objects are specified linguistically in terms of collections of signs, but the signs are understood to name mental objects in such a way that the properties of the object can be determined from the arrangement of the signs. The possibility of adding new primitive types is left open, but a fixed finite number (three, to be exact) are specified in advance.

For the definition in 2.2 we will also need the concept of an operation which assigns to the elements of a type, say \(A\), objects of

\(^1\)This is precisely Martin-Löf's [24] way of putting the matter. It is a refinement of Bishop's conception of a "constructive set" [5].
another type, say B. In particular, it is possible to assign to elements of a previously specified type small types which have already been built. At the intuitive level, these algorithms or operations will be presented by a typed \( \lambda \)-notation, e.g. \( \lambda x \in A.x \) is the identity operation on type A.

Finally, one of the important notions used to describe operations in the theory is the intuitive function space constructor, \( + \). The notation \( x \in A \rightarrow B(x) \) denotes the type of all functions which, on input of an element of A, say \( x \), return an element of the type given by evaluating the function \( B \) on \( x \). This is the informal idea of what it means for an algorithm to have a range that depends on its argument.

2.2. The Type Hierarchy

With these preliminary concepts understood, we are ready to specify precisely the type of small types. This is done by defining first the primitive types, then individuals of these types, then the basic type constructors, finally a method of building individuals (canonical objects) of each constructed type. Thus the type of small types has an inductive character, and the definition of the type of small types (denoted \( V \)) is to be understood as an open-ended inductive definition. It is summarized in figure 1.

In this core theory, the primitive types are chosen to have respectively, zero, one and two elements. The type with zero elements, denoted \( \mathbb{E} \), is called the void type. It is a contradiction to construct any object whose type is \( \mathbb{E} \). The type with one element is denoted \( \mathbb{I} \), and has as its only element \( 0_{\mathbb{I}} \). The type with two elements is denoted \( \mathbb{B} \), and has as its elements \( 0_{\mathbb{B}} \) and \( 1_{\mathbb{B}} \).

Product Creation

One of the type constructors found commonly in programming languages is the cartesian product, in Algol 68 it is the \texttt{structure} constructor, in Pascal the \texttt{record}, etc. This concept is a special case of an infinitary cartesian product introduced into AUTOMATH. That constructor, which we call \texttt{product} and denote by \( \mathbb{P} \), subsumes the function space constructor (e.g. \texttt{proc} of Algol 68). We adopt the operator name and rules from ITT75, but because we
want to be able to analyze the structure of types inside the theory, we
depart fromITT75 and treat \( \Pi \) as an operator in the theory.

The product operation takes as arguments a type \( T \) from the class of
all (small) types \( V \) and an operation \( F \) which maps elements of \( T \) to elements
\( V \). Elements of product types are operations which map elements \( x \) of \( T \) to
elements \( F(x) \). We say more about these elements in 2.3, product
introduction.

Union Creation

Union types are created by the \( \Sigma \) operator, which takes arguments with the
same type specifications as the \( \Pi \) operator. Union types, however, are a
generalization of binary disjoint union in that the union can be indexed by
any arbitrary type, including one with an infinite number of elements.

Well-ordering Creation

The \( \mathbb{W} \) operator creates a well-ordering type upon application to a type \( A \)
in \( V \) and a function \( B \) from \( A \) to types in \( V \). An element of a well-ordering
can be thought of as a tree formed so that each branch has finite length,
but the number of descendants from any node in the tree may be infinite.
The nodes of one of these trees have values associated with them, taken
from the type \( A \). The fan-out from a node labelled with a value \( a \) in \( A \) can
be put in one to one correspondence with the elements of \( B(a) \). More
formally, an element of a well-ordering type is represented by an element \( a \)
of \( A \), and a function, which on application to an element \( b \) of \( B(a) \) gives
the element of the well-ordering type at the end of the edge labelled with
\( b \) that comes out of the node labelled with \( a \). This function is a
generalized predecessor function, as an element of a well-ordering has as
many predecessors as there are elements in the type specified by applying \( B \)
to the label on the root node.

Quotient Creation

A quotient type is created from a type \( A \) in \( V \) and an operation \( E \) of type
\( A \rightarrow A \rightarrow V \), and is denoted \( A / E \). The elements of a quotient type are
equivalence classes of elements of \( A \), under the equivalence relation formed
by taking the reflexive, symmetric, transitive closure of \( E \) treated as a
relation.
Equality Creation

With each type T we associate a companion type constructor denoted =T. For any two elements x, y of T, =T(x)(y) is also a type. Intuitively this is the type of the proofs that x and y are equal elements of type T. The axioms for =T will specify the equality relation on a type T. There are numerous other ways that such information might be supplied, see for example [12], but this approach is quite elegant and has been examined carefully by Martin-Löf who uses it in [21,24].

Universes

The inductively defined collection of types built from the primitive types by \( \Pi, \Sigma, \mathsf{W}, / \) and = is called the type of all \textit{small types} and is denoted \( V_1 \). We understand from this process how to apply it to more general collections. For instance, if we take \( V_1 \) itself to be a new type, then we can imagine extending \( \Pi, \Sigma, \mathsf{W}, / \) and = so that they operate on \( V_1 \). The resulting collection of types is denoted \( V_2 \) and the process is repeated to form \( V_i \) for any natural number i. In fact by imagining this hierarchy we see how to present the informal types of \( \Pi, \Sigma, \mathsf{W}, / \) and = as types in this new wider sense.

Martin-Löf has introduced the concept of universes of types built in this way in order to allow the expression of concepts which require arbitrary elements in a collection of types. Such hierarchical notions seem necessary in any strictly typed theory, e.g. Russell and Whitehead [32]. Martin-Löf chooses to introduce universes after he has defined the general category of type. We attempt to explain an entire hierarchy of universes simultaneously. This seems necessary if we are to be able to analyze the concept of type inside the theory itself as in section 2.6.

2.3. Introduction and Elimination

Having described the primitive types and their elements, and how to construct new types from pre-existing types, it remains to show how to build elements of composite types. This will be done separately for each type constructor. For each type formation method, there are operations for building elements of the type, and operations which, given an element of
Primitive small types:
- \( \mathbb{E} \): called the void or empty type \( \mathbb{E} \in V \)
- \( \mathbb{I} \): the unit type with exactly one element \( \mathbb{I} \in V \)
- \( \mathbb{B} \): the Boolean type with exactly two elements \( \mathbb{B} \in V \)

Individuals:
- \( 0_{\mathbb{I}} \in \mathbb{I} \)
- \( 0_{\mathbb{B}} \in \mathbb{B} \)
- \( 1_{\mathbb{B}} \in \mathbb{B} \)

Type constructors:
- \( \mathbb{X}: A \in V \rightarrow (A + V) + V \): the product constructor
- \( \mathbb{Z}: A \in V \rightarrow (A + V) + V \): the union constructor
- \( \mathbb{W}: A \in V \rightarrow (A + V) + V \): the well-ordering constructor
- \( /: A \in V \rightarrow (A + A + V) + V \): the quotient constructor

Figure 1

the type, produce different elements, possibly of other types. We refer to these two types of operations as "introduction" and "elimination" operations.

For every type \( T \), we will specify the equality operation \( =_T \), written \( =_T \), with the intuitive type \( =_T: x \in T + (y \in T + V) \). For each method of defining new types, we will give conditions under which elements of that type are equal. (Note that on quotient types, equality is not always decidable.)

Most of the functions introduced here are parameterized with respect to the types they will take as arguments when finally applied. Thus the equality operation described above actually has the type \( =: T \in V \rightarrow (T + (T + V)) \). Operations will be described without the extra parameters and will be used in examples without explicitly giving these parameters, but their full proper typing will be given in the table that follows.

**Primitive Type Introduction**

There are no functions which introduce elements of our primitive types. The elements of the primitive types are assumed to exist.
**Primitive Type Elimination**

We need functions which operate on (eliminate) elements of our primitive types. For any choice of type $T$, there is a function $z$ which maps elements of $\mathbb{N}$ into an element of $T$. (Since there are no elements of $\mathbb{N}$, this corresponds to being able to construct an element of any type from the contradiction that an element of the empty type was supplied.) For any choice of a function $T: \mathbb{N} \to V$, there is a function $\text{if}$ which given elements $t_1$ and $t_2$ of type $T(0)$ and $T(1)$, respectively, maps an element $b$ of $\mathbb{N}$ to $t_1$ if $b = 0$ and to $t_2$ if $b = 1$. (Note that $1$ elimination falls under the $K$ combinator discussed below.) This gives us the intuitive types

\[
\begin{align*}
    z: \mathbb{N} &\to T \\
    \text{if}: x \in T(0) \to (T(1) \to (x \in \mathbb{N} \to T(x)))
\end{align*}
\]

**Product Introduction**

To construct objects of types built with the "product constructor" $\Pi$, we must have operations which construct functions and manipulate functions. The most primitive set of such functions, borrowed from the (untyped) combinatory calculus, are the $S$ and $K$ functions. Intuitively, we have the relations

\[
\begin{align*}
    K(x)(y) &= x & \text{or } K &= \lambda x.\lambda y.x \\
    S(f)(g)(x) &= f(x)(g(x)) & S &= \lambda f.\lambda g.\lambda x.f(x)(g(x))
\end{align*}
\]

A strict composition rule for typed combinators would require that the domain of the first operation be exactly that of the range of the second operation. In a system with operations whose range type can depend on the argument to the function, such a strict rule impedes the building of generalized operations. For specific terms of the argument type we would be able to compose the operations, but it would be impossible to build the general composition. Therefore, we allow a more liberal form of composition by modifying the $S$ combinator. It can compose two functions provided that a proof is supplied that demonstrates that the composition is type correct on all elements of the intended range.

The type of the $S$ and $K$ operations is, for some choice of types $A$, $B(x)$ and $C(x)$ and $D(x,y)$:
\( K: x \in A \to (B(x) \to A) \)
\( S: (x \in A \to (y \in B(x) \to D(x,y))) \to g \in (x \in A \to C(x)) \)
\( + (x \in A \to (B(x) \equiv C(x))) + x \in A \to D(x,g(x)) \)

Many of the properties of untyped combinators carry over to these typed combinators, although the types do introduce more complexity. In untyped combinators, the identity combinator can be defined in terms of \( S \) and \( K \) by \( I = SKK \). Using these typed combinators, we can construct an identity function for any type \( T \) by: replacing \( A \) by \( T \) and \( B(x) \) by \( \Pi(A)(\lambda x \in A.T) \) in the type of the first \( K \); replacing \( A \) by \( T \) and \( B(x) \) by \( A \) in the type of the second \( K \); and replacing \( A \) by \( T \), \( B(x) \) and \( C(x) \) by \( \Pi(A)(\lambda x \in A.T) \), and \( D(w,z) \) by \( T \) in the type of \( S \). For untyped combinators, given an expression involving a variable \( x \), we can abstract with respect to \( x \) to obtain a combinatory term containing no instances of \( x \), but which when applied to a value \( v \) evaluates to the same value as the original expression would if \( v \) were substituted for all occurrences of \( x \). Similar transformations are possible with typed combinators, given that we know the type of the variable being abstracted. Thus our intuitive notion of functions denoted by means of \( \lambda \)'s can be brought into the theory.

Product elimination

The elimination of elements of \( \Pi \) types is carried out by the operation of application. Given an element \( f \) of the type \( \Pi(A)(B) \), and an element \( a \) in \( A \), then \( f(a) \) is an element of the type \( B(a) \).

Two operations are equal (under \( =_{\Pi(A)(B)} \)) if and only if they have identical normal forms. This aspect of function equality will be discussed later.

Union Introduction

The introduction of elements of a type formed by the \( \Sigma \) operation is done by the \( p \) (for pair) function. It maps an element \( t \) of a type \( T \) and an element \( s \) of a type \( S(t) \) into the type \( \Sigma(T)(S) \).

Union Elimination

The elimination of such types corresponds to taking a \( p \)'ed element apart into the pieces it was formed from. Two projection functions, \( p1 \) (the "first" element) and \( p2 \) (the "second" element) exist to perform that action.
\[ p: t \in T \rightarrow (S(t) \rightarrow 2(T)(S)) \]
\[ p1: 2(T)(S) \rightarrow T \]
\[ p2: x \in 2(T)(S) \rightarrow S(p1(x)) \]

Two elements of a union type are equal if the first elements are equal according to their type, and the second elements are equal according to the type that they belong to. In terms of the operations introduced above, we have, for \( a \) and \( b \) of type \( 2(A)(B) \),
\[ a =_{2(A)(B)} b \equiv (p1(a) =_{A} p1(b) \& p2(a) =_{B} p2(b)) \]

**Well-ordering Introduction**

Creating elements of \( W \) types closely corresponds to definition by induction. Given an element \( x \) of a type \( A \), and a function from \( B(x) \) to the \( W \) type in question, the function \( \text{sup} \) creates an element of the \( W \) type.

**Well-ordering Elimination**

One form of elimination from a well-ordering type is similar to using the \( p1 \) and \( p2 \) operations on elements of a union type. That is, given an element \( \text{sup}(a, f) \), we can operate on it to obtain the \( a \) and \( f \) objects. \( 1b(x) \) gives as its value the label associated with a node in the tree; \( pd \) gives the predecessor function for that node. The other form of elimination of \( W \) types corresponds to the definition of recursive functions on the \( W \) type. The operation \( \text{rec} \) takes as its argument a function, which on an element \( x \) of the \( W \) type and a function to create elements of the type \( C(y) \) for all predecessors \( y \) of \( x \), produces an element of \( C(x) \). \( \text{rec} \) produces as its result a function, which given an element \( x \) of the \( W \) type produces an element of \( C(x) \).

\[
\begin{align*}
\text{sup}: & \quad x \in A \rightarrow (B(x) \rightarrow W(A)(B)) \rightarrow W(A)(B) \\
1b: & \quad x \in W(A)(B) \rightarrow A \\
pd: & \quad x \in W(A)(B) \rightarrow (B(1b(x)) \rightarrow W(A)(B)) \\
\text{rec}: & \quad (x \in W(A)(B) \rightarrow (v \in B(1b(x)) \rightarrow C(pd(x)(v))) \rightarrow C(x)) \rightarrow (W(A)(B))(C)
\end{align*}
\]

Two elements of a \( W \) type are equal if they have the same label at their root, and the same predecessor function. For the label, "the same" means equal by the equality defined for the type of the label.
\[ x =_{W(A)(B)} y \equiv (1b(x) =_{A} 1b(y) \& pd(x) =_{B(1b(x))}(W(A)(B))(pd(y))) \]

- 13 -
Quotient Introduction

Elements of $\mathcal{A}/\mathcal{E}$ types are formed by a "one way" operation; it is one of the few in the theory that is not in some sense "reversible". Consider a quotient type $\mathcal{A}/\mathcal{E}$. Let the equivalence relation induced by $\mathcal{E}$ be $\mathcal{F}$. $\mathcal{F}(x)(y)$ is a type which is non-empty if $x = y$ or if there is a chain $x_1, \ldots, x_n$ where $x_1 = x$ and $x_n = y$ and for each $x_i, x_{i+1}$ pair, either $\mathcal{E}(x_i)(x_{i+1})$ or $\mathcal{E}(x_{i+1})(x_i)$ is non-empty. For a particular function $\mathcal{E}$ and element $x$ of a type $\mathcal{A}$, the value $\mathcal{q}(x)$ is an element of the type $\mathcal{A}/\mathcal{E}$ such that if $\mathcal{F}(x)(y)$ is non-empty, then $\mathcal{q}(x) = \mathcal{A}/\mathcal{E}\mathcal{q}(y)$.

Quotient Elimination

Elimination of quotient types can only be carried out by functions formed in a special way. We say that a function $\mathcal{f}$ respects the equivalence relation when $\mathcal{F}(x)(y)$ non-empty implies $\mathcal{f}(x) = \mathcal{f}(y)$. A function $\mathcal{f}$ of type $x \in \mathcal{A} \rightarrow \mathcal{B}(x)$ can be converted into one that maps from elements of the quotient type into the same range as specified by $\mathcal{B}$, if both $\mathcal{B}$ and $\mathcal{f}$ respect the equivalence relation.

$$\mathcal{q} : \mathcal{A} \rightarrow (\mathcal{A}/\mathcal{E})$$
$$\mathcal{q} : \mathcal{f} \in \mathcal{A}(\mathcal{A})(\mathcal{B}) \rightarrow \mathcal{A}(\mathcal{A}/\mathcal{E})(\mathcal{B})$$
where $\mathcal{f}$ and $\mathcal{B}$ both respect $\mathcal{F}$ (the equivalence relation induced by $\mathcal{E}$), such that $(\mathcal{q}(\mathcal{f}))(\mathcal{q}(\mathcal{a})) = \mathcal{f}(\mathcal{a})$

Note that in this formulation it appears at first glance that the type requirements for $\mathcal{q}$ are circular, or at least recursive. But since any $\mathcal{B}$ to which $\mathcal{q}$ will be applied the second time will have a type such as $\mathcal{A}(\mathcal{A})(\mathcal{KV})$, the "recursion" stops there, since the constant function $\mathcal{KV}$ can easily be rewritten to be a function of type $\mathcal{A}(\mathcal{A}/\mathcal{E})(\mathcal{V})$, by changing the type of the $\mathcal{K}$ combinator used.

Two elements $\mathcal{q}(a)$ and $\mathcal{q}(b)$ of a quotient type are equal if and only if $\mathcal{F}(a)(b)$ is a non-empty type. Note that this definition of equality is in general undecidable, as it depends on the type $\mathcal{F}(a)(b)$ being empty or non-empty.

$$\mathcal{q}(a) = \mathcal{A}/\mathcal{E}\mathcal{q}(b) \equiv \mathcal{F}(a)(b),$$
where $\mathcal{F}$ is the induced equivalence relation

Equality Introduction

Elements are assumed to exist for all equality types of the form $=T(x),x$. The elements are written $ax_{x,x}$ and can be thought of as

-14-
primitive proofs or axioms stating that objects are equal to themselves.

**Equality Elimination**

In order to make an equivalence relation out of equality, we assume operations that modify elements of equality types to give us the properties of symmetry and transitivity.

\[
\text{sym: } = (T)(x)(y) \leftrightarrow = (T)(y)(x) \\
\text{tran: } = (T)(x)(y) \leftrightarrow = (T)(y)(z) \leftrightarrow = (T)(x)(z)
\]

So to add to our summary of the theory in Figure 1, we can list the functions mentioned above with their types, as in Figure 2.

\[
\begin{align*}
\text{=} & : T \in V \rightarrow (T \rightarrow (T \rightarrow T)) \\
\text{sym: } & : T \in V \times x \in T \times y \in T \rightarrow = (T)(x)(y) \rightarrow = (T)(y)(x) \\
\text{tran: } & : T \in V \times x \in T \times y \in T \times z \in T \rightarrow = (T)(x)(y) \rightarrow = (T)(y)(z) \rightarrow = (T)(x)(z) \\
\text{z: } & : T \in V \rightarrow (\mathbb{N} \rightarrow T) \\
\text{if: } & : T \in (B \rightarrow V) \times x \in T(0) \rightarrow (T(1) \rightarrow (x \in B \rightarrow T(x))) \\
\text{K: } & : A \in V \times B \in (A \rightarrow V) \times x \in A \rightarrow (y \in B(x) \rightarrow A) \\
\text{S: } & : A \in V \times B \in (A \rightarrow V) \times C \in (A \rightarrow V) \times D \in (x \in A \times B(x) \times V) \rightarrow \\
& \quad = (x \in A \rightarrow (y \in B(x) \times D(x, y))) \rightarrow g \in (x \in A \times y \in C(x)) \rightarrow (x \in A \rightarrow (B(x) = C(x))) \\
& \quad \rightarrow x \in A \rightarrow D(x, g(x)) \\
\text{p: } & : A \in V \times B \in (A \rightarrow V) \times t \in A \rightarrow (s \in B(t) \rightarrow \mathbb{Z}(A)(B)) \\
\text{pl: } & : A \in V \times B \in (A \rightarrow V) \times \mathbb{Z}(A)(B) \rightarrow A \\
\text{p2: } & : A \in V \times B \in (A \rightarrow V) \times x \in \mathbb{Z}(A)(B) \twoheadrightarrow B(p1(x)) \\
\text{sup: } & : A \in V \times B \in (A \rightarrow V) \times x \in A \rightarrow (B(x) \rightarrow W(A)(B)) \rightarrow W(A)(B) \\
\text{1b: } & : A \in V \times B \in (A \rightarrow V) \times x \in W(A)(B) \rightarrow A \\
\text{pd: } & : A \in V \times B \in (A \rightarrow V) \times x \in W(A)(B) \rightarrow (B(1b(x))) \rightarrow W(A)(B)) \\
\text{rec: } & : (A \in V \times B \in (A \rightarrow V) \times C \in (W(A)(B) \rightarrow V) \\
& \quad \times x \in W(A)(B) \rightarrow (v \in B(1b(x))) \rightarrow C(pd(x)(v))) \rightarrow C(x) \rightarrow \Pi(W(A)(B))(C) \\
\text{q: } & : A \in V \times E \in (A \rightarrow A \rightarrow V) \rightarrow A \rightarrow (A/E) \\
\text{q: } & : f \in A \in V \times B \in (A \rightarrow V) \rightarrow E \in (A \rightarrow A \rightarrow V) \rightarrow \Pi(A)(B) \rightarrow \Pi(A/E)(qB) \\
\text{where f and B both respect the equivalence relation induced by E}
\end{align*}
\]

**Figure 2**
2.4. Typings Within the Theory

The type constructor \( \Pi \) is intended to represent the intuitive arrow, \( \rightarrow \), so that for \( B \) of type \( A + V \), \( \Pi(A)(B) \) represents \( x \in A + B(x) \). Sometimes to show the correspondence to the intuitive concept we will write \( \Pi(A)(B) \) as \( \Pi x \in A.B(x) \), and we write \( \Sigma(A)(B) \) as \( \Sigma x \in A.B(x) \).

Using the correspondences noted above and in the previous section, we can convert the above type specifications to expressions called typings which are written in the notation of the \( V \) types themselves. For example, \( I \in A + A \) can be written \( I \in \Pi(A)(\lambda x \in A.A) \). This says that \( \Pi(A)(\lambda x \in A.A) \) represents the intuitive type \( A + A \). Since \( A \in V \) and \( \lambda x \in A.A \) is a constant operation \( A + V \), the application of \( \Pi \) to \( A \) and \( \lambda x \in A.A \) is type correct.

Once we realize the correspondence between \( \Pi \) and \( + \); \( K \), \( S \) and \( \lambda \)-terms, and hence combinators, it is tempting to express the intuitive concepts in the formal system itself. But when doing this, one must be careful about levels. We need the concept of \( + \) and \( \lambda \) to define \( \Pi \) and \( K \). This will be clear from an attempt to define the type \( \Pi(A)(\lambda x \in A.A) \) entirely within the system. We would expect this to be \( \Pi(A)(KA) \), but the \( K \) used here is \( \lambda y \in V.\lambda x \in y.y \), which has type \( V + (V + V) \), which if written as a typing, \( K \in \Pi(V)(\lambda x \in V.(x + V)) \), would be misleading because this \( \Pi \) is a more abstract operation mapping from the large type containing \( V \). We would have to distinguish it from the first \( \Pi \) by writing the first as \( \Pi_1 \) and the second as \( \Pi_2 \). Then we could say \( K \in \Pi_2(V)(\lambda x \in V.(x + V)) \).

2.5. Large Types

What we have said up to this point effectively describes the collection of small types. But also we can see that the collection itself is a type. We can imagine other objects like it, formed from different base types and closed under different operations. We can grasp the meaning of mappings \( V + V \) and unions \( \Sigma(V)(\lambda x \in V.V) \). If there were another large type \( U \), say the type of sets, we could imagine operations between them: \( V + U \), \( U + V \).
The particular concept of a large type that we have in mind consists of \( V \) as a new primitive, and permits all of the types of \( V \) to be "lifted" to large types (but it is not possible to create small types by mapping \( V_2 \) into \( V_1 \)). It is also closed under large versions of \( \Pi, \Sigma, \text{ and } / \). For example, \( \Pi_2 \) has the type \( x \in V_2 \rightarrow (x + V_2) \rightarrow V_2 \).

2.6. Intensionality

In the theory it is possible to analyse the structure of all objects. The first step in the analysis is to be able to recognize the building blocks out of which they are constructed. In order to build as strong a decidable equality as possible in the face of an open-ended universe, we must ensure that equality on the basic constants of the theory is decidable. We assume that basic constants (the ones mentioned in this paper) are recognizable by the use of the \( \text{atom} \) operation, and that equality of such atoms is decidable by the \( \text{eq} \) operation.

\[
\text{atom}: T \in V \rightarrow x \in T \rightarrow B \\
\text{eq}: T \in V \rightarrow \exists x \in T. \text{atom}(x) \rightarrow \exists y \in T. \text{atom}(y) \rightarrow B
\]

Using these operations, we can construct the operations \( \text{is} \Sigma, \text{is} \Pi \) and \( \text{is} B \) which will recognize the basic types from \( V \), and the decidable equality on boolean types comes from \( \text{eq} \).

We also have a discriminator

\[
\text{isap}: T \in V \rightarrow x \in T \rightarrow B
\]

which will decide if an object is of the form \( f(a) \). Objects of this form arise whenever a primitive function (such as \( S \) or \( K \), for instance) is applied to fewer arguments than is necessary to be able to reduce the application to a simpler form. They also arise in cases where the form of an object is given by the application of one function to some arguments, for example with functions formed by \( \text{rec} \) or \( \text{q} \).

\( \text{isap} \) and \( \text{atom} \) are related by the fact that \( \text{isap}(T)(t) \Rightarrow \neg \text{atom}(T)(t) \), and \( \text{atom}(T)(t) \Rightarrow \neg \text{isap}(T)(t) \). But the open-ended universe allows the possibility that an object is neither an atom nor formed by application, but is rather imported into the system by some new method of constructing objects.
Given that an object is an application, we want to be able to analyze the function being applied and the object to which it is being applied. The operation **split** performs this function, returning an element of a union type which contains all the relevant information.

\[
\text{split: } T \to V \times x \in T \to \text{isas}(x) \to \\
\exists S \in V. S \in V \times f \in \Pi(S)(B). S \in S \times p \in S(s) = v.T.f(s) = x
\]

Using these operations, we can build operations which analyze the intensional structure of \( V_1 \) objects. Each such operation maps \( V_1 \) into \( B \).

\[
is\Sigma: V \to B \quad \text{isas: } V \to B \\
is/ : V \to B \quad \text{is } \Pi: V \to B
\]

We can construct decomposition operations that allow us to analyze an element of a function type that is not a single primitive combinator, and in particular to obtain information about its domain and range.

\[
\text{optype: } \Sigma y \in \Pi(T)(F).\text{isas}(y) \to V \\
\text{argtype: } \Sigma y \in \Pi(T)(F).\text{isas}(y) \to V \\
\text{op: } x \in \Sigma y \in \Pi(T)(F).\text{isas}(y) \to \text{optype}(x) \\
\text{arg: } x \in \Sigma y \in \Pi(T)(F).\text{isas}(y) \to \text{argtype}(x)
\]

We can also construct combinators to decompose the types. For \( \Pi, \Sigma, \) and \( W \) combinators, one can obtain the type over which the quantification is being performed, and the function mapping the type to \( V \). For quotient types, one can obtain the base type and the equivalence relation being used.

\[
\text{index: } \Sigma y \in V.(\text{isas}(y) \vee \text{isas}(y) \vee \text{isas}(y)) \to V \\
\text{family: } x \in \Sigma y \in V. \text{isas}(y) \vee \text{isas}(y) \vee \text{isas}(y) \to (\text{index}(x) \to V) \\
\text{base: } \Sigma y \in V. \text{isas}(y) \to V \\
\text{rel: } x \in \Sigma y \in V. \text{isas}(y) \to (\text{base}(x) \to \text{base}(x) \to V)
\]

where

\[
\text{index}(\Pi(A)(F)) = A \\
\text{family}(\Pi(A)(F)) = F \\
\text{base}(A/E) = A \\
\text{rel}(A/E) = E
\]

The strength of allowing intensionality lies in the ability to completely break down an object and build a new one from its components. This is accomplished in informal reasoning by a form of structural induction on the expression representing the object. To mirror that in the theory, we must have as a primitive a combinator allowing recursion on the
form of an object, with appropriate typing to ensure the recursion terminates.

\[ \text{RV: } C(\mathbb{I}) + C(1) + C(\mathbb{E}) + \]
\[ (x \in \mathbb{Z} \land y \in \mathbb{V}, \text{is}\mathbb{I}(y) + C(\text{index}(x)) + \]
\[ f \in \mathbb{I}(\text{index}(x))(C(\text{family}(x))) + C(x) + \]
\[ (x \in \mathbb{Z} \land y \in \mathbb{V}, \text{is}\mathbb{E}(y) + C(\text{index}(x)) + \]
\[ f \in \mathbb{E}(\text{index}(x))(C(\text{family}(x))) + C(x) + \]
\[ (x \in \mathbb{Z} \land y \in \mathbb{V}, \text{is}\mathbb{V}(y) + C(\text{index}(x)) + \]
\[ f \in \mathbb{V}(\text{index}(x))(C(\text{family}(x))) + C(x) + \]
\[ (x \in \mathbb{Z} \land y \in \mathbb{V}, \text{is}\mathbb{T}(y) + C(\text{base}(x)) + \]
\[ f \in \mathbb{T}(\text{base}(x))(\mathbb{T}(\text{base}(x))(C(\text{rel}(x))(y))) + C(x) + \]
\[ (x \in \mathbb{V} + C(x)) + (x \in \mathbb{V} + C(x)) \]

The eight operands are functions which return a result under the assumption that the argument is the type \( \mathbb{I} \), the type \( 1 \), the type \( \mathbb{E} \), formed from a \( \mathbb{I} \) operation, etc. It reduces in the "obvious" manner:

\[ \text{RV}(f_1, \ldots, f_8)(x) = \]
\[ \text{if } x = \mathbb{I} \text{ then } f_1 \]
\[ \text{else if } x = 1 \text{ then } f_2 \]
\[ \text{else if } x = \mathbb{E} \text{ then } f_3 \]
\[ \text{else if } x = \mathbb{T}(\mathbb{F}) \text{ then } f_4(x, \text{RV}(f_1, \ldots, f_8)(\text{index}(x))) \]
\[ \text{else if } x = \mathbb{E}(\mathbb{F}) \text{ then } f_5(x, \text{RV}(f_1, \ldots, f_8)(\text{index}(x))) \]
\[ \text{else if } x = \mathbb{V}(\mathbb{F}) \text{ then } f_6(x, \text{RV}(f_1, \ldots, f_8)(\text{index}(x))) \]
\[ \text{else if } x = \mathbb{T}/\mathbb{E} \text{ then } f_7(x, \text{RV}(f_1, \ldots, f_8)(\text{base}(x))) \]
\[ \text{else } f_8(x) \]

In a similar manner, we want to break apart operations into their components. There is a structural recursion combinator on the \( \mathbb{T}(\mathbb{F}) \) types similar to \( \text{RV} \) that makes this possible.

\[ \text{RF: } T \in \mathbb{V} + x \in T + (T + C(\text{op}(x)) + C(\text{arg}(x)) + C(x)) \]
\[ + (T + C(x)) + (T + C(x)) + C(x) \]

The combinator acts as

\[ \text{RF}(T)(f_1, f_2, f_3)(x) = \]
\[ \text{if } T = \mathbb{T}(\mathbb{F}) \text{ then if isop}(x) \]
\[ \text{then } f_1(x, \text{RF}(\text{optype}(x))(f_1, f_2, f_3)(\text{op}(x))) \]
\[ \text{RF}(\text{argtype}(x))(f_1, f_2, f_3)(\text{arg}(x))) \]
\[ \text{else } f_2(x) \]
\[ \text{else } f_3(x) \]

We can summarize these intensionality functions as is done in figure 3.
atom: T ∈ V + x ∈ T → B

eq: T ∈ V → x ∈ T . atom(x) → x ∈ T . BATOM(y) → B

isap: T ∈ V → x ∈ T → B

split: T ∈ V → x ∈ T → isap(x) →
∈ T . 2S ∈ V . 2B ∈ S + V . 2f ∈ Π(S)(B) . 2s ∈ S . 2p ∈ S(s) = y . T . f(s) = x

isE: V → B

isI: V → B

isB: V → B

isΠ: V → B

index: ∈ V . (isΠ(y) ∨ isE(y) ∨ isI(y)) → V

family: x ∈ V . (isΠ(y) ∨ isE(y) ∨ isI(y)) → (index(x) → V)

base: ∈ V . is / (y) → V

rel: x ∈ V . is / (y) → (base(x) → base(x) → V)

RV: C(Π) + C(1) + C(B) →
(x ∈ V . isΠ(y) + C(index(x)) →
∈ Π(index(x))(C(family(x))) → C(x)) →

(x ∈ V . isE(y) + C(index(x)) →
∈ Π(index(x))(C(family(x))) → C(x)) →

(x ∈ V . isI(y) + C(index(x)) →
∈ Π(index(x))(C(family(x))) → C(x)) →

(x ∈ V . is / (y) + C(base(x)) →
∈ Π(base(x))(Π(base(x))(C(rel(x)(y)))) → C(x)) →

(x ∈ V + C(x)) → (x ∈ V + C(x))

optype: ∈ Π(T)(F) . isap(y) → V

argtype: ∈ Π(T)(F) . isap(y) → V

op: x ∈ Π(T)(F) . isap(y) → (Bopdom(x) + Boprng(x))

arg: x ∈ Π(T)(F) . isap(y) → T

Ref: T ∈ V + x ∈ T → (T + C(op(x)) + C(arg(x)) + C(x)) → (T + C(x)) + (T + C(x)) → C(x)

Figure 3

Using V2 concepts and functions, we can formalize the informal notions used to define V1. For example, Π1 has the type

Π1 ∈ Π2(V1)(∀x ∈ V1.(x + V1) + V1).

But this concept requires the informal concept of a dependent operation. By leaving the level structure open-ended, we create the illusion that the entire system can be formalized within itself.
The theory, although complex and powerful, has been reduced to a simple core of combinators and primitive types. The summarized form in figures 1 through 3 presents all the information necessary to describe level 1 of the theory.

2.7. Definitional Equality

In practice, one wants to introduce various definitions. For example, one might want to define the binary disjoint union, say, as

\[ S \cup T = \Sigma(\mathcal{B})(\text{if}(S)(T)) \]

This form of definition and the notion of equality used in it is a linguistic matter. That is, \( S \cup T \) is not a new canonical form of the theory, it is merely an abbreviation of existing forms. The equality \( S \cup T = \Sigma(\mathcal{B})(\text{if}(S)(T)) \) is not a new mathematical identity over \( V \), it simply relates expressions.

We adopt the approach to definitional equality taken in [22,23]. The form of definitions is

\[
\text{for } x_1 \in A_1, \ldots , x_n \in A_n(x_1, \ldots , x_{n-1})
\text{ define } f(x_1) \cdots (x_n) = \text{exp}
\]

where \( \text{exp} \) cannot refer to \( f \).

The rules of definitional equality are standard and the relation is decidable in time \( n \cdot \log(n) \) [11].

2.8. Equality and Normal Forms

The normal form of an expression is obtained by performing all substitutions for definitional equalities, and then performing all the reductions given by the equalities in Figure 4 below. The resulting expression will have no occurrences of applications which could be simplified by substituting arguments for parameters, or by applying the "obvious" simplification rules.

We can now discuss function and type equality in more detail. Two types or functions are equal if their normal forms are the same. With the intensionality functions described in an earlier section, we can almost
write a function of level $V_2$ which would decide equality for types (or functions) of level $V_1$. Such a decision procedure does not handle types and functions allowed into the universe by the open-ended nature of the constructive theory. We can agree that a type which is from outside the theory and one constructed with $\Xi$, for example, are different types; but there is no obvious answer in the case of two un-analyzable objects.

2.9. Working at Higher Levels

One simplifying restriction made on functions is that one is not able to build a function which maps from a $V_2$ type to create new $V_1$ types. In order to be able to work at higher levels using concepts from lower levels, we include the up combinator. This combinator raises elements of $V_1$ to be elements of $V_2$, and similarly transforms elements of those types to be elements of the newly created type.

\[
\begin{align*}
if(x)(y)(0_{\text{up}}) &= x \\
if(x)(y)(1_{\text{up}}) &= y \\
p(p1(x))(p2(x)) &= x \\
p1(p(x)(y)) &= x \\
p2(p(x)(y)) &= y \\
\overline{q}(f)(q(x)) &= f(x)
\end{align*}
\]
\[
\begin{align*}
\text{K}(x)(y) &= x \\
\text{S}(f)(g)(x) &= f(x)(g(x)) \\
\text{sup}(1b(x))(pd(x)) &= x \\
1b(\text{sup}(x)(f)) &= x \\
pd(\text{sup}(x)(f)) &= f
\end{align*}
\]
\[
\begin{align*}
is\Pi(\Pi(A)(B)) &= 1_{\text{up}} \\
is\Pi(T) &= 0_{\text{up}} \text{ otherwise} \\
is\Xi(\Xi(A)(B)) &= 1_{\text{up}} \\
is\Xi(T) &= 0_{\text{up}} \text{ otherwise} \\
\Pi(index(T))(family(T)) &= T \\
index(\Pi(A)(B)) &= A \\
family(\Pi(A)(B)) &= B \\
\Xi(index(T))(family(T)) &= T \\
index(\Xi(A)(B)) &= A \\
family(\Xi(A)(B)) &= B \\
\text{base}(T) / \text{rel}(T) &= T \\
\text{base}(A / E) &= A \\
\text{rel}(A / E) &= E
\end{align*}
\]

Figure 4
\[ \eta p : V_1 \rightarrow V_2 \]
\[ \eta p_T : T \rightarrow \eta p(T) \text{ for every type } T \]

Figure 5

This final combinator, given in figure 5, allows us to "renumber" the levels at which we have constructed objects. All that is necessary to describe any level of the hierarchy are the descriptions in figures 1 to 5.

2.10. Theories

In PL/CV3, a specific theory is an element of a dependent product type, and theories can be parameterized. In this core version, we simply take a theory to be a sequence of typings, definitions and definitional equalities. This is a linguistic notion of theory which we do not attempt to identify with a mathematical object. Typings of the form \( x \in T \) for \( x \) a variable are assumptions, other typings must follow from previous typings and equations by one of the rules listed in the previous sections.

2. Applications

In this section we relate the core theory to more familiar concepts, such as the predicate calculus, natural numbers, and a representation of lists. We are not concerned with the pragmatic issues that arise in trying to really use the full type theory to represent these concepts. Such matters will be discussed in the Ph. D. thesis of the second author, in [10], and in the work of the PRL project at Cornell.

3.1. Embedding Constructive Logic in the Theory

We will illustrate how the type theory can interpret (higher order) constructive logic. Propositions will correspond to types; this is the "proposition-as-types" principle. To begin we assume that we are given a translation of atomic propositions, say \( A(x) \), into atomic types, also
denoted A(x). Thus an atomic predicate will be interpreted as a type valued function. The objects of the atomic type A(x) are proofs of the proposition A(x). We have seen an example of this correspondence in section 2.2 where \( axT, x, x \) is a cannonical proof object belonging to the type \( x = _T x \).

Given this correspondence for atomic propositions, our goal is to extend it to compound propositions. We must translate a proposition P into a type, Trans(P), such that if P is provable then Trans(P) is nonempty. For a given notion of proof in higher order logic, we show that if p is a proof of P, then we can translate p to an object p' of type Trans(P).

This is not the right paper for a complete proof of such a theorem because so much detail concerning the logical system is required. Therefore we omit many details, most important is the abstraction algorithm to convert lambda terms to combinators. Also we refer the reader to the literature for some standard natural deduction presentation of constructive higher order logic, say Prawitz [25]. More details of this result can be found in [12].

The higher order logic we will embed is modelled after that presented in [25]. We assume the existence of a type T, the type of an individual variable. The types of the logic are built from T using +. An n-ary predicate whose arguments are of types \( S_1, S_2, \ldots, S_n \) is itself of type \( [S_1, S_2, \ldots, S_n] \). Then the following clauses give the definition of a formula of the system:

1. If P is a predicate of type \( [S_1, S_2, \ldots, S_n] \) and \( x_1, x_2, \ldots, x_n \) are of types \( S_1, S_2, \ldots, S_n \), respectively, then P(\( x_1, x_2, \ldots, x_n \)) is a formula.
2. If P and Q are formulas, then P & Q, P v Q, \( P \Rightarrow Q \), and \( \neg P \) are formulas.
3. If P(x) is any formula in which the variable x occurs as a free variable of type S, then \( \forall x \in S. P(x) \) is a formula, and \( \exists x \in S. P(x) \) is a formula.

We assume the usual constructive interpretations for the logical connectives \&, \vee, \Rightarrow, \neg, \forall, \text{ and } \exists. \ In \ particular, \ we \ assume \ that \ proofs \ are \ given \ in \ an \ introduction/elimination \ (or \ natural \ deduction) \ style \ proof
tree. We will use the notation $\vdash P$ means that the formula $P$ is provable within the logic, and that $\mathcal{H} \vdash P$ means that $P$ is provable within the logic under the assumptions in the set $\mathcal{H}$.

We first create a function which translates propositions from the syntax of the logic to the syntax of types. We assume that atomic predicates $B$ of type $[S_1, \ldots, S_n]$ are translated to atomic type valued functions also denoted $B$. We have seen in 2.2 how the atomic equalities, $x =_T y$, can be regarded as types so that $=_T$ is a type valued function.

$\text{Trans}(\text{false}) = 0$
$\text{Trans}(\text{true}) = 1$
$\text{Trans}(x) = x$ for all variables $x$
$\text{Trans}(T) = T$
$\text{Trans}(S_1 \rightarrow S_2) = \text{Trans}(S_1) \rightarrow \text{Trans}(S_2)$
$\text{Trans}([S_1, S_2, \ldots, S_n]) = \Pi x \in \text{Trans}(S_1). \text{Trans}([S_2, \ldots, S_n])$
$\text{Trans}(f(x)) = \text{Trans}(f)(\text{Trans}(x))$
$\text{Trans}(A \land B) = \exists x \in \text{Trans}(A). \text{Trans}(B)$ where $x$ is a new variable
$\text{Trans}(\exists x \in A.B(x)) = \exists (\text{Trans}(A))(\text{Trans}(B))$
$\text{Trans}(A \lor B) = \exists n \in \mathbb{N}. \text{if} (\text{Trans}(A), \text{Trans}(B), n)$ where $n$ is a new variable
$\text{Trans}(A \Rightarrow B) = \Pi x \in \text{Trans}(A). \text{Trans}(B)$ where $x$ is a new variable
$\text{Trans}(\forall x \in A.B(x)) = \Pi (\text{Trans}(A))(\text{Trans}(B))$

Note that the $\exists$ and $/$ type formation operations are not used in the result of any translation. One can imagine a logic including a form of "recursive proposition" or "infinite proposition" which would be translated into a type involving the $\exists$ operation. However, the inclusion of $\exists$ and $/$ in the type theory means that we cannot in general translate from types back into propositions.

To complete the proof that logic is embedded in the type theory, we should show that the proof rules for the logic have corresponding operations in the type theory that preserve the notion of truth. This is equivalent to showing that for all propositions $P$,

$\vdash P$ implies there is an element of the type $\text{Trans}(P)$.

We will sketch the proof of a slightly stronger statement.

**Theorem:**

Given a proof of $\mathcal{H} \vdash P$ we can find a $\forall x$ expression of type $\text{Trans}(P)$ whose only free variables are of type $\text{Trans}(\text{h})$ for $\text{h}$ in $\mathcal{H}$.

- 25 -
Proof:

Proceed by induction on the height of the proof tree for \( H \vdash P \).

Base case:
The tree height is 0. Then \( P \) is an \( h \) in \( H \). The proof expression is \( x \) of type \( \text{Trans}(h) \).

Inductive step:
The height of the tree is \( k \), and we can assume the theorem for all propositions and proof trees of height \( < k \). We must show the theorem holds for height \( k \) as well, and will do so by cases on the rule being applied at the root of the proof tree.

& Intro:
Let the propositions proven in the left and right subtrees be \( A \) and \( B \), respectively. The left and right subtrees are of height less than \( k \), so by the induction hypothesis (working with \( H \vdash A \) and \( H \vdash B \)), the types \( \text{Trans}(A) \) and \( \text{Trans}(B) \) can be shown non-empty. Let the elements of those types be "a" and "b". Then the element \( p(a)(b) \) is of type \( \exists x \in A \cdot B \) where \( B \) does not depend on \( x \). But this is exactly an element of \( \text{Trans}(A \& B) \).

& Elim:
By the induction hypothesis, we have an element of the type corresponding to the proposition proven at the root of the subtree, which is of the form \( A \& B \). Let the element be "a", an element of the type \( \text{Trans}(A \& B) \), that is, an element of \( \exists x \in \text{Trans}(A) \cdot \text{Trans}(B) \). Then \( p2(a) \) is an element of the type \( \text{Trans}(B) \), or \( p1(a) \) is an element of the type \( \text{Trans}(A) \), whichever was required.

\& Intro:
We are attempting to prove a proposition of the form \( A \lor B \), and have proven either \( A \) or \( B \) in the subtree below the root. By the induction hypothesis, we can construct an element "a" of the type \( \text{Trans}(A) \) (or \( \text{Trans}(B) \), as the case may be), and then \( p(0)(a) \) (or, respectively, \( p(1)(a) \)) will be a member of the type \( \exists n \in B.\text{if}(\text{Trans}(A),\text{Trans}(B),n) \), which is exactly \( \text{Trans}(A \lor B) \).
\( \forall \text{ Elim:} \)

The first subtree below the root proves a proposition of the form \( A \lor B \), the second proves a proposition of the form \( A \Rightarrow C \), and the third proves one of the form \( B \Rightarrow C \). By the induction hypothesis applied to the first subtree, we can find an element "a" of the type \( \operatorname{Trans}(A \lor B) \). Applying the induction hypothesis to the second subtree gives a function \( f_a \) of the type \( \operatorname{Trans}(A \Rightarrow C) \). Similarly, from the third subtree, we obtain a function \( f_b \) of the type \( \operatorname{Trans}(B \Rightarrow C) \). Then the element \( \text{if } f_a f_b (p1(a)) p2(a) \) is an element of \( C \) (where \( D(0_b) = A \Rightarrow C \) and \( D(1_b) = (B \Rightarrow C) \) and \( \text{if}(f_a f_b (p1(a))) \in D(p1(a)) \)).

\( \Rightarrow \text{ Intro:} \)

Given a proof tree of a proposition of the form \( A \Rightarrow B \) from hypotheses \( H \), we can obtain, by the induction hypothesis, an element "b" (expressed using combinators) of the type \( \operatorname{Trans}(B) \) from the proof tree (minus the root node) that corresponds to the proof \( H \cup \{ A \Rightarrow B \} \). Abstracting "b" with respect to the element of \( A \) assumed in its creation gives a function "\( f_b \)" which maps elements of \( A \) to elements of \( B \). This abstraction can be performed using the combinators provided, leaving us with a function expressed as a sequence of combinators.

\( \Rightarrow \text{ Elim:} \)

The subtrees provide proofs of a proposition of the form \( A \Rightarrow B \), and of the proposition \( A \). By the induction hypothesis, we can obtain from the proof of \( A \Rightarrow B \) a function \( f \) mapping elements of \( \operatorname{Trans}(A) \) to elements of \( \operatorname{Trans}(B) \), and from the proof of \( A \) an element \( a \) of \( \operatorname{Trans}(A) \). Applying the function to the element of \( \operatorname{Trans}(A) \) gives an element \( f(a) \) of \( \operatorname{Trans}(B) \), as is required.

\( \forall \text{ Intro:} \)

The subtree provides a proof of a proposition of the form \( B(x) \) for an arbitrary \( x \) drawn from some type, say \( A \), from a set of hypotheses \( H \). Applying the induction hypothesis to the proof of \( H \cup \{ x \in A \Rightarrow B \} \), we obtain an element of \( B \) expressed in combinators, depending on the element of \( \operatorname{Trans}(A) \) chosen. Abstracting with respect to this element gives the function we want to interpret as a proof of the proposition \( \forall x \in A. B(x) \), which is a member, as required, of \( \Pi(\operatorname{Trans}(A))(\operatorname{Trans}(B)) \).
∀ Elim:
Similar to the ⇒ elimination case. Note that since all objects are represented as combinators, we do not need to worry about capture of bound variables -- there are none to be captured.

∃ Intro:
The subtree of the proof is a proof of a proposition of the form B(x) where x is a particular element of some type, say A. By the induction hypothesis, we can build an element "b" of Trans(B(x)). So the element p(x)(b) is an element of Σ(Trans(A))(Trans(B)), which is Trans(∃x ∈ A.B), as was to be proven.

∃ Elim:
Similar to the ∀ elimination case.
ORD.

So any proposition in this extended logic can be modeled by a type, and if the proposition is true, possibly under some hypotheses, then the translated type is non-empty. Furthermore, all the proof rules of this extended constructive logic correspond to some action on types.

3.2. Building the Integers

We have not assumed the existence of the type of non-negative integers in the core theory because they can be built as a well-ordered type. Thinking again of elements of a W type as trees, the integer 0 will be represented as a tree of exactly one node labelled with 0_B: the successor of an integer n will be represented as a tree with a root labelled by 1_B and a single outward edge to the tree which represents n. So labels come from the type B, and there are either no edges out of a node if the label is 0_B, or one edge if the label is 1_B. So the number of outward edges corresponds to elements of 0 and 1, respectively. In order to build such a W type, we need a function f of type B→V such that f(0_B) = 0, and f(1_B) = 1. Such a function is λx ∈ B.if(0)(1)(x), or, using combinators, if(0)(1). So the type

Nat = W(B)(if(0)(1))
describes the type of non-negative integers.

2.3. An Iteration Function

Preliminary to defining lists in the theory, we define an iteration operation. Using the non-negative integers just built, we construct an operation which takes as input a type T, a function f of type $T \to T$, and an integer n, and returns as output the function of type $T \to T$ produced by composing f with itself n times ($f^n$). The function we are looking for is thus of type $T \in V \to f \in \Pi(T)(\Xi(T)) \to \text{Nat} \to \Pi(T)(\Xi(T))$. Since it is built as a recursive function, it must be constructed as a consequence of W elimination; specifically, elimination on the non-negative integers as defined above. In order to build a recursive function, we need a function which, given

(1) natural number, or, more specifically, the label and predecessor function which determine the number, and

(2) a function, which given an element labeling one of the out-edges from the root node of the given number, yields an element of the desired type; in this case, $T \to T$,

returns an element of the desired type, again, namely $T \to T$.

A function with these objects as arguments has type

$$n \in \text{Nat} \to (\Pi_m \in \text{if}(\Xi, 1, n).(T \to T)) \to (T \to T)$$

Intuitively, it is putting together an answer for the number from an answer for the previous number. If the number was 0, we want the identity function on T to be returned. Otherwise, we want to compose the function whose power we are taking with the answer obtained from the predecessor. By the above definition of Nat, the number is 0 if its label is 0. So part of the answer is $\text{if}(\text{Id}_T, \ldots, 1b(n))$. What goes into the $\ldots$ must be the composition of the given function, f, together with the predecessor's answer. Let g be the function to produce the answer for the predecessor; then the function to which the recursion combinator is applied is

$$\lambda n \in \text{Nat}. \lambda g \in (\text{if}(\Xi, 1, 1b(n)) \to (T \to T)).
(\text{if}(\text{Id}_T, S(\Xi(f))(g(0)_1)). 1b(n)).$$
Abstracting with respect to $f$ and $T$, we obtain the real definition of the exponentiation function:

$$
expon = \lambda T \in V. \lambda f \in T \to T. \rec(\lambda n \in \text{Nat}. \lambda g \in (\text{if}(\text{if}(\lambda f. \lambda n. \text{if}(0, 1, \text{if}(\text{if}(\lambda f. \lambda x. f(x), s(f)), g(0)), 1), n)))).
$$

This can be rewritten completely in combinators, but would be more unpalatable than the above notation. In a user-pleasant system, it would be specifiable as

$$
expon = \lambda T \in V. \lambda f \in T \to T. \lambda n \in \text{Int}. \text{if } n = 0 \text{ then } \text{Id}_T \text{ else } \text{expon}(T)(f)(n-1)
$$

with automatic translation to the internal form. Research is currently being done into the efficiency of such automatic transformations.

3.4. Implementing Lists

A list of elements of a specified type is either a marker indicating the end of the list, or it is an element of the type followed by a list.

Consider a fixed type $T$. Using the type $\mathbf{1}$ as the type containing our marker, we would like to say that the type of lists of elements of $T$ is given by the disjoint union of $\mathbf{1}$ and the product of $T$ and the list type. This method of definition can be used to define recursive types [10], but introduces complications that are better left out of a core theory. In this presentation, we will build lists by using the exponentiation operation defined in the previous section to build the type of lists of length up to $n$ for all integers $n$; these types can then be united by the $\Sigma$ operation to form the type of all finite length lists.

We will use the shorthand notation $A+B$ for $\Sigma n \in \mathbf{B}. \text{if}(A, B, n)$, representing the disjoint union of two types; similarly $A \times B$ will be shorthand for $\Pi x \in A. B$, the cartesian product of two types.

Define the function $\text{LL}(S) = 1 + T \times S$. Note that $\text{LL}(\mathbf{1})$ is the type of all empty lists; $\text{LL}(\text{LL}(\mathbf{1}))$ is the type of all lists with one or zero elements, and so on. In particular, $\text{expon}(V)(\text{LL})(n)(\mathbf{1})$ (for expon redefined to take large types as a first argument) is the type of all lists.
of length less than or equal to n. So the union of these types over all integers n is the type we are looking for.

\[ \text{List} = \forall T : \text{V} \cdot \exists n : \text{Nat} \cdot \text{expon}(\text{V})(\text{LL})(n)(\text{I}) \]

So List is a function which on an arbitrary type returns the type of lists over that type.

3.5. A Programming Problem

We show that any list of objects of any small type can be reversed. This illustrates a simple case of polymorphic programming. Reversal is defined recursively as:

\[
\text{Rev}(n, x, y) = \begin{cases} 
\text{if } n = 0 & \text{then } ((\text{len}(x) = 0) \Rightarrow y = \text{nil}) \\
\text{else } & ((\text{len}(x) = \text{len}(y) \& \text{len}(x) = n) \Rightarrow \\
& \text{hd}(x) = \text{last}(y) \& \text{Rev}(n-1, \text{tl}(x), \text{top}(y)))
\end{cases}
\]

where \( \text{last}(y) = \begin{cases} 
\text{if } \text{tl}(y) = \text{nil} & \text{then } y \text{ else } \text{last}(\text{tl}(y)) \text{ fi and}
\end{cases} \)

\[
\text{top}(y) = \begin{cases} 
\text{if } \text{tl}(y) = \text{nil} & \text{then } \text{nil} \text{ else } \text{cons}(\text{top}(\text{tl}(y)), \text{hd}(y)) \text{ fi}
\end{cases}
\]

We know as a lemma that \( \text{last}(\text{append}(z, x)) = z \).

The task of reversing a list of elements of type T is specified by asking for a member of the type

\[ \Pi x : \text{List}(T). \exists y : \text{List}(T). \text{Rev}(\text{len}(x), x, y). \]

An element of this type is a procedure which on input x produces a list y and a proof that y is the reversal of x. The procedure can be regarded as an inductive proof of the proposition or as a recursive procedure. Let us present it intuitively as a proof. A common tactic to produce such results is to replace \( \text{len}(x) \) by n and use natural number induction. So we prove:

\[ \forall n : \text{Nat}. \forall x : \text{List}(T). \exists y : \text{List}(T). \text{Rev}(n, x, y) \]

Proof (by induction on n):

Base case: \( n = 0 \)
Take \( y = \text{nil} \) and take as proof of \( \text{Rev}(0, x, \text{nil}) \) a proof of \( 0 = 0 \Rightarrow \text{nil} = \text{nil} \).
(Notice, it is important here that \(\text{Rev}(0, x, y)\) does not mention \(\text{Rev}(n-1, \text{tl}(x), \text{top}(y))\) which is not well-defined when \(n=0\).)

**Induction case:** assume \(\forall x \in \text{List}(T). \exists y \in \text{List}(T). \text{Rev}(n, x, y)\).

\(\forall x \in \text{List}(T). \exists y \in \text{List}(T). \text{Rev}(n+1, x, y)\)

**Proof:** arbitrary \(x \in \text{List}(T)\)

\(\exists y \in \text{List}(T). \text{Rev}(n+1, x, y)\) by cases

**Proof**

- **case** \(\text{len}(x) \neq n+1\); take \(y=\text{nil}\) and use \(z\) to prove \(\text{len}(x) = n \Rightarrow \text{hd}(x) = \text{last}(y) \& \text{Rev}(n, \text{tl}(x), \text{top}(y))\)

- **case** \(\text{len}(x) = n+1\);

  \(\text{len}(\text{tl}(x)) = n\) hence \(\exists y \in \text{List}(T). \text{Rev}(n, \text{tl}(x), y)\) by induction

  Choose \(y_0\) where \(\text{Rev}(n, \text{tl}(x), y_0)\),

  Take \(y = \text{append}(\text{hd}(x), y_0)\),

  As proof of \(\text{hd}(x) = \text{last}(y)\) return the appropriate equality axiom since as a lemma about \(\text{last}\) we know \(\text{last}(\text{append}(\text{hd}(x), y_0)) = \text{hd}(x)\). As proof of \(\text{Rev}(n, \text{tl}(x), y_0)\) return the induction assumption applied to \(\text{tl}(x)\).

QED

QED

Although this proof is written in the PL/CV2 style, it can be routinely converted to the appropriate recursion combinator. The form of the combinator is \(\text{rec}\ f\) where \(f\) is the body of the proof whose structure is essentially

\[
\lambda n:\text{Nat}. \lambda p:\Pi x \in \text{List}(T). \exists y \in \text{List}(T). \text{Rev}(n, x, y).
\]

\[
\lambda x \in \text{List}(T). \text{if } n=0 \text{ then } b(x) \text{ else if } \text{case}(x) \text{ then } h_1(x) \text{ else } h_2(p, x) \text{ fi}
\]

3.6. **Further Use of the Theory**

Data structures common to computer programs can be written as types in the type theory; most, except for those that themselves have types as components, will lie in \(V_1\). Because of the intensionality of the theory, we can reason about implementations of these data structures with the theory, as well as proving theorems concerning the complexity of functions which operate on these types.
4. Acknowledgements

We would like to thank all our associates whose conversation and criticism have helped us shape the theory. In particular, Per Martin-Löf, Mike O'Donnell, Stuart Allen, Joe Bates, Alan Demers, Paul Dietz, Carl Eichenlaub, Bob Harper, and Daniel Leivant have given their time and thought to the project.

5. References


15. Feferman, S., "Constructive Theories of Functions and Classes", Logic Colloquium '78, North-Holland, Amsterdam, 1979, pp. 159-224.


- 34 -

