Report on the Type Theory (V3) of the Programming Logic PL/CV3

Robert L. Constable
Daniel R. Zlatin

TR 81-454

Cornell University
June, 1980
Revised April 3, 1981

Research supported in part by NSF grant MCS 78-00953.

Table of Contents

Preface

PL/CV3 is a logic for reasoning about constructive objects which include the data types of high level programming languages and programs themselves as well as the typical objects of a constructive mathematics. It provides a logical foundation for rigorous program development and has been tailored to suit such development for a class of procedural languages (including parts of Algol 68, PL/I, Pascal, Ada, or Russell).

PL/CV3 is a dialect of a more general class of logics characterized by seven properties.

(1) Each logic of the class is constructive and is intended to express all of constructive mthematics. Presently, the class is based around Martin-Löf's Intuitionistic Type Theory (ITT) [Martin-Löf 75], a theory which evolved frm [Scott 70] and is closely related to the typed lambda calculus of AUTOMATH [de Bruijn 70].

(2) The logics are designed to be automated and useable. Proofs can be built with mechanical assistance and are checked by a mechanical proof checker. One of the principal techniques for achieving this is the employment of automatic rules of inference and user defined proof strategies.

(3) The concept of a command and the attendant notion of a general recursive procedure are built-in (but the concept is not primitive and can be regarded as a linguistic convention). So each member of the class is a _programming logic_, and is intended to serve in

building practical programs.

(4) Functions (including proofs) and commands can be executed. Indeed programs are to be efficiently executed. So each logic defines a programming language as well.

(5) A rich notion of recursive type is available.

(6) Objects are treated in a strongly intensional way; they can be taken apart and reassembled by functions of the theory (somewhat in the style of LISP [see Allen 78]). So the metalanguage of level n is available at level n+1.

(7) The logics are integrated with a computer system to present a unified view of the universe to an interactive user who works inside the theory.

The family of logics with these characteristics was called AUTOLOG in [Constable 80].[1] This family is similar in spirit to the AUTOMATH family of [de Bruijn 70]. The full name for PL/CV3 would be AUTOLOG-PL/CV3. The name PL/CV3 arises because the system extends PL/CV2, an elementary constructive programming logic based on features (2), (3), and (4). PL/CV2 has been implemented and used since 1978.[2] The programs of PL/CV2 are a subset of PL/I (hence the name PL...) which can be executed by various PL/CS compilers [Conway, Gries 75], and the Cornell

---

[1]The name used there was FORMAL, which has since been superceded by the more descriptive name.

[2]It is described in the publications A Programming Logic [Constable, O'Donnell 78], A Computer System for Checking Proofs [Johnson 80], Introduction to the PL/CV2 Programming Logic and Verifier [Constable, Johnson, Eichenlaub 81], and AVID, An Interactive Proof Development System [Krafft 81].

Program Synthesizer [Teitelbaum, Reps 81]. The above features are more compatible with the Algol 68 programming language (without explicit use of ref objects), but the PL/CV2 system was built in an environment where PL/I was the principle language.

The principal aim of this system is feasible formal expressiveness. That is, we aim to express naturally a very wide class of algorithms, problems and solutions, including all those which commonly arise in mathematical computing. Moreover we aim to express them formally in a way that permits mechanical checking of arguments and execution of functions and commands. These goals are at variance with mathematical brevity. This logic is not small, it is not concise. While the concepts are powerful and elegant, and the core logic is simple, their elaboration into a useable system results in a complex mass of detail.

One system with comparable aims is Edinburgh LCF. In fact, LCF has been a model for us in phrasing properties (3) and (7) in detail. We subscribe to the methodology espoused in LCF, but we believe in properties (1) through (7) as well. The Edinburgh LCF system is also very large and complex, even though it too is the elaboration of simple and elegant mathematical concepts [Scott 70].

It is possible that building any feasible formal system will result in a large system. In any case, this system is large, and this report is not intended to be a leisurely introduction to it. We will write other documents which explain the system, which describe features of its possible full implementation, which reduce it to its essential mathematical core and which prove properties of this core.[3] Many of those reports will need to reference a defining document like this.

We are planning a proof checker for the full PL/CV3 logic. It will provide an experimental verification facility similar in spirit and philosophy to the Edinburgh LCF system [Gordon, Milner, Wadsworth 79]. But as with PL/CV2, the checker will use constructive reasoning, so classical arguments must exhibit non-constructive steps explicitly. Nevertheless, any classical argument should be expressible.

This report concentrates on describing the type theory and the applicative core language. We call the core V3. Details of the programming rules and the interactive environment are omitted. This core can be integrated into any programming logic similar in style to PL/CV (an Algol logic would be appealing). It can also be used as a completely applicative language in the style of LISP or the reduction languages of [Backus 78].

[3]The core logic will be described by the authors in an article to appear in the proceedings of the IBM Logic of Programs Conference, May 1981, to be published by Springer-Verlag Lecture Notes in Computer Science.

Acknowledgements

We wish to thank Alan Demers, Jim Donahue, Joseph Bates, Scott Johnson, Dean Krafft, Carl Eichenlaub, and Paul Dietz for helpful comments and stimulating discussions about types.

We appreciate the NSF support under grant MCS 78-00953 which made this work possible, and the support to the second author provided by the IBM Corporation.

1. Introduction

1.1. The Logic

The PL/CV3 logic is based on the concept of a constructive type. A type can be thought of as the intensional version of a set. i. e. it is more than a collection of objects, it is a method of constructing elements of the type. So a type is a method of construction. Among the objects of the theory are proofs and computations, as well as the integers, booleans, sets, and other standard types. Types are themselves objects of a higher type, forming an open hierarchy of types.

The logic of the underlying theory is constructive and all of the facilities for mechanical assistance in generating and checking proofs assume a constructive logic. Classical non-constructive results can be obtained in the usual ways if they are ever needed (see [Bishop 67; Constable 80]). As a consequence of the constructivity, all objects claimed to exist can be exhibited; in particular all functions are computable and every statement has computational meaning. It makes sense to execute the proofs of theorems of the form "for all x of type A there is a y of type B". By providing an object "a" of type A, the user can ask the proof to compute an object of type B.

An explicit program syntax is provided which allows for the more or less conventional procedural specification of algorithms as well as the applicative style of a purely functional logic. The particular syntax used here is PL/I-like for historical reasons, but the concepts apply equally well to similar parts of Algol 68, Ada, and related languages. Programs, unlike proofs, are intended to execute efficiently. It is possible that an interpreter will be available for the entire class of programs of PL/CV3.

The logic freely mixes programs with types, so they can be mixed with assertions to produce asserted programs and formulas of algorithmic logic (or programming logic or dynamic logic as it is sometimes called, see [Rasiowa 77, Pratt 76, Constable 77]).

The PL/CV2 language and its proof checker and proof synthesizer were designed to support elementary instruction in algorithmic problem solving and in certain mathematical aspects of programming and computer science. The full PL/CV3 logic is primarily a research tool for the study of program verification and mechanically assisted reasoning (especially in computational mathematics).

This report is a very brief account of the PL/CV3 logic. It is written for a mature reader familiar with the basic elements of mathematical logic and computer programming, and concentrates on the applicative core of the theory.

1.2. The universe

The PL/CV3 system maintains an environment in which users and other processes can interact with the universe of constructive mathematical objects. The PL/CV3 logic defines the universe and its laws, the system enforces these laws and enables processor actions.

Base types included in the theory are integers, strings, booleans and atomic propositions. New types can be built from them by the operations of the theory (unions, function spaces, quotients, etc.). Members of the base types, i.e. specific integers, strings and truth values, are not types themselves. However, every object has a type, including those objects which are themselves types. This leads to a hierarchy of types -- the type of all types that can be built from the base types and the operations is given a name and becomes a new base type from which a larger universe of types can be constructed. Thus, the universe is stratified into levels, 1, 2,..., n,... . At level n+1, the universe at level n can be described.

Objects in the theory that are not themselves types, that is, do not themselves have elements, are known as <u>individuals</u>. Integers, characters, the boolean constants "True" and "False", and objects formed from them by the action of injections are examples of individuals.

Following [Curry, Feys and Craig 58; Howard 69; Martin-Löf 75], who pointed out the analogy between propositions and types, we include propositions as types. Members of propositions viewed as types are proofs of the proposition. Combining this interpretation of a proposition as a type with the operations on types gives a natural constructive semantics to the predicate calculus, with natural extensions to higher order logic, corresponding to higher levels in the universe. In talking about the universe at level n, we use a meta-language that cannot be part of the theory at level n. Because of these natural extensions to higher levels, the meta-language can be expressed at level n+1 and hence is part of the theory as a whole.

PL/CV3 takes a strongly intensional view of the universe. That is, any object (in particular, a function) may be examined by a function (which is itself object of the theory) and different actions taken depending on the structure of the object. Because of the difference between our intensional functions and the usual definition of a function (an input/ouput relation), we shall from here on refer to intensional functions as operations.

The operations of the theory are indexed by level. The level of the operation used depends on the level of its arguments. Thus the operation itself (at a specific level) is an object of the theory.

1.2.1. Frames of reference

The PL/CV3 system provides the user a type in which to build objects. The level of this type is determined by the highest level of objects constructed within it.

Associated with the type in which the user is working is a type called the frame of reference, similar to a "state" as used in other semantic theories. It associates with each name in use a type and a value. The type associated with a name in the frame of reference is the type to which values bound to the name must belong; the corresponding value (of that type) gives the current value bound to the name. Every object the user creates can be accessed uniquely through the frame of reference.

Commands and declarations change the frame of reference. A command maps the frame to itself, changing the current values of variables. A declaration changes the frame of referrence by adding new components (new user names) to it. A command or declaration also has the effect of changing the universe so that user supplied names always refer to the newly transformed value. This feature of commands, called the _global naming convention_, is axiomatized in terms of the familiar assignment rule.

1.2.2. _Processes and operations_

The user can

(1) construct a member of a specified (non-empty) type,

(2) name a constructed object of the theory,

(3) apply (constructed and builtin) operations to other objects of the theory (both types and individuals),

(4) alter the frame of reference.

These operations can be given meaning within the system, and hence, as in the above discussion on operations within the theory, are operations with as many instances as there are levels. Furthermore, they can be used from objects constructed within the theory, much as LISP functions can themselves call EVAL on objects they have constructed.

Operation 1, constructing a member of a type known to be non-empty, is quite simple. For a logical theorem, finding a member of the type which is the theorem is equivalent to finding a proof of the theorem - but since the theorem (type) is known to be proven (non-empty, i.e. a proof has already been given) finding the proof is just a matter of looking it up (verifying it). This operation can be assisted by the system in varying amounts - some automatic inference rules can be built in, or a theorem prover could be incorporated (making this operation one of "attempt to find a member"), or a combination of human and automatic proving could be used.

Note that the interpreter/theorem prover/verifier is an object of the system; it can be written down as a program for any specific level, and acts at a level which depends on the arguments to which it is applied.

Operation 2 provides a way of naming objects. It is also the source of one form of equality, which can be proven by reference to the definition. As discussed above in terms of declarations and commands, the frame of reference concept gives semantics to the binding of values to names and is still a part of the theory.

Operation 3 covers most of the actions users will ordinarily perform. Applications of builtin operations (such as "+") and operations which arise out of the rules for the intensionality of objects (such as "ISUNION") are placed in their most simplified form automatically. Applications of user defined operations are simplified whenever possible. Note that this category of action includes the conditional operation which tests if a (decidable) type is empty or not, choosing one of two actions as a result.

Operation 4 is distinct from declarations and most commands. By "altering the frame of reference", we mean changing one's viewpoint of which names are local and which global. A procedure call in ordinary programming languages comes close to emulating the effect. Before the transfer of control, one set of variable names is meaningful; some are easily accessible (local variables) and others (possible none), belonging to the called procedure, are less easily accessed. After the transfer, a different but not necessarily disjoint set of names is meaningful; some names may occur in both sets, but with different meanings.

Due to PL/CV3's strong intensionality, any name can be accessed from any position in the universe. However, some are considered local, while others need complete qualification, depending on the scope of the declarations involved. When the user (or a program) changes the frame of reference, a new set of names are considered local, while some of the old set may now need extensive qualification. This has similarities to constructs in several programming languages. In Pascal, the "WITH <structure reference> DO ... END" opens up the structure referred to, so that its fields need not be qualified in the begin block that follows. A similar feature is available in SIMULA 67, dealing with the variables available for reference in a class to which other classes have been prefixed.

As a consequence of operations 1, 2, 4 and the intensional interpretation of the universe, it is possible to enter and edit structured data by using the operators provided by the strong intensionality rules. Such structured editing can also support a (proof) synthesizer of the type described in [Teitlebaum 1979; Krafft 1981]. But it is also possible to use the same features to build a synthesizer which is aware of the logical structure of arguments and which enforces a particular development methodology such as top down refinement. (It appears that LCF tacticals could also be used to build such a specialized proof construction methodology.) In particular the program refinement logic of [Bates, 1979] can be viewed as a specialized collection of proof generating functions.

Since all operations and user actions can be specified from within the theory, PL/CV3 provides a uniform view of actions to both the user and to a process within the system. All actions taking place in the metalanguage at one level are parts of the theory at a higher level, and can be reasoned about at that level. For example, users are permitted to define operations which are given the maximum level number of their arguments as an extra argument, unspecifiable except by changing the actual parameters. Since operations always terminate, any invocation will use some maximum level of the universe, depending on the arguments supplied. The invocation can then be described as an object in the universe one level higher than that. This allows users to create objects similar to the built-in proof rules, where the level used is a function of the parameters.

2. Types

2.1. Introduction

This introduction to types is brief. Readers desiring a more explanatory and philosophical account of the type theory are referred to [Constable 80, Martin-Löf 75].

A *type* in this logic can be viewed as a method of constructing elements. For example, the product type $A \times B$ denotes the method of pairing an object from A with one from B. These types are similar to Algol 68 modes, but unlike a mode, types in PL/CV3 are themselves values, and can be manipulated with appropriate operators. In this respect, they are similar to those of Russell [Donahue and Demers, 79].

The type constants provided by the system are listed below.

INT	The type of integers.
BOOL	The type of the logical constants "true" and "false". The PL/I names "'1'B", "'0'B", "FIXED", and "BIT" are also allowed.
CHAR	The type of single characters in the character set of the language.
VOID	The canonical type with no members.
TYPE(i)	See below for further explanation.

Types in PL/CV3 can be treated as objects and operated on to produce more complex types. The operations available are:

Cartesian product: Given types S and T, $S \times T$ (or PROD(S, T)) is the type whose elements are built by pairing together an element of S with an element of T (the operation extends in the obvious way to an arbitrary number of types as operands). Labels can be applied to the operands of the product, as in PROD(first : INT, second : INT), and then used as selectors to obtain one of a pair. For example, given a value z in the above type, z.first is the value of its first component, while z.second is the value of its second. This construction has a parallel in many programming languages - a structure in PL/I, a mode of the form <u>structure</u>(field1 ..., field2 ...) in Algol 68, and a record in PASCAL can be considered types which are the cartesian products of the types of their components. Dependent products can also be defined in PL/CV3 - that is, products of the form PROD(x : T1, y : T2(x)), where the type of a later component depends on the value of a previous component. (PASCAL variant records are a special case, where the value of one component is used to choose among a finite number of alternative types for other components in a structure.)

<u>Disjoint union</u>: Given n types $S_1,\ldots,S_n$ (not necessarily distinct), $S_1 + \cdots + S_n$ (or UNION($S_1,\ldots,S_n$)) represents their disjoint union. The system provides a case discriminator operation, CASE(i,r), where r is a member of a union type, and $1 \le i \le n$, which represents the proposition that r comes from the i^{th} disjunct of a disjoint union. Injectors to move a value from a type into a union type are also provided by the system: INJ(S, UNION(S, T)) (s) is a member of S + T for s a member of S.

Unions over an infinite index type are also available. $\forall x \in T.P(x)$, where P is an operation from T to TYPE(i) for some i, represents the union of all the elements of the type P(x) as x varies over T. As will be seen later, the infinite union types effectively subsume the finite union types and the product types (above).

Recursive definition: New types can be constructed from old ones by recursive definition, that is, writing a definition for a type T that involves T as well as other types. For example, the type of binary trees of integer values is given by

DEFINE tree = INT + (INT × tree × tree).

As well as using a DEFINE statement to define a recursive type, a form similar to PROD can be used:

R(tree : INT + (INT × tree × tree))

is equivalent to using the above. Simultaneous recursion over several types is allowed; for example, the type of binary trees could also be defined as

R(tree : INT + tree_structure,
tree_structure : INT × tree × tree)

although this definition is not entirely equivalent to the one given above.

Recursive definition of types appears in many programming languages, often in the form of a structure containing pointers or references to new allocations of the structure itself. In particular, unlike PL/I pointers which can point to any location in memory, Algol 68 references are to specific modes, and PASCAL pointers are also bound to a specific type, which is the spirit of these recursive definitions.

Function types: Given a type T and an expression B(x) which evaluates to a type for every x in T, we can form the type of a function space by $\Lambda x \in T.B(x)$. This type has as members operations which given an argument y of type T return a value of <u>type</u> B(y). In the degenerate case where B(x) = S for some constant type S, $\Lambda x \in T.S$, where S does not depend on x, is abbreviated as $T \rightarrow S$. The Λ operator is (constructive) universal quantification over the type T.

<u>Quotient types</u>: Within the intensional type system, one can obtain extensional implementations (sets and extensional equivalence of functions, for example) by using the quotient operator. An operation f of type $A \rightarrow A \rightarrow$ TYPE is an equivalence relation if it is symmetric, reflexive, and transitive (interpreting the output type as a proposition, symmetry would be phrased "f(a)(b) <=> f(b)(a)"). We can then form the type A / f, the type of all equivalence classes of A under f. Functions from a quotient type to any other type are constrained to respect the equivalence relation of that quotient type; a function h of type $(A / f) \rightarrow B$ must have the property that for x and y in A / f (obtained from x' and y' in A), if f(x')(y') is true, then it must be the case that h(x) = h(y).

<u>Parameterized types</u>: An expression which evaluates to a type for all values of some variable in the expression can be abstracted with respect to that variable, giving a parameterized type. Parameterized types are operations which return types as their value. For example, an operation which when given a type returns the "square" of the type can be written as

FOR t TYPE DEFINE square(t) = t × t

This facility becomes more useful after the introduction of types which contain typea as members (see below).

Types as objects: Types take the status of objects when a 'large type' is constructed, that is, a type whose elements are themselves types. The simplest of such large types is the constant provided by the system, TYPE, or TYPE(1). This is the type whose elements are types constructible from the base types, and the operators listed above. Thus the type of the type INT is TYPE, as are the types of INT → INT, INT × CHAR, and 'tree' as defined above, for example. .

Similarly, since TYPE(1) itself must have a type, there is a constant TYPE(2) which denotes the type whose elements are all types constructible from the base types plus TYPE(1), and the operations listed above. Thus $\Lambda x \in$ TYPE.(x × x) (the type of all operation which take a type as argument and return as value a *member* of the "square" of that type) is of type TYPE(2), as is TYPE × INT, the type of all ordered pairs whose first element is a "small" type and whose second is an integer.

Continuing in this manner, the type of TYPE(2) is TYPE(3), and so on. This hierarchy of types is countably infinite, and, as argued elsewhere ([Martin-Lōf 75]), we do not see how to close off with a universal "type of all types" and remain a constructive theory consistent with ordinary mathematics.

According to the simple doctrine of types espoused here (see [Martin-Lōf 75]), every object must have a type. As a system for reasoning about programs, we must be able to deal with proofs and propositional reasoning. The type of a proof is the proposition it proves. For example, the type $\forall n \in$ INT.prime(n) (whose type is TYPE) is essentially the set of all ordered pairs whose first element is an integer n, and whose second element is a member of the type prime(n), that is, a proof of the proposition that n is prime (given a suitable definition of the proposition/type "prime").

2.2. The Syntax and Type Consistency of Expressions

In this section the syntax of an expression is presented in terms of the operators available in the language. The operators will be discussed in order of their precedence in an expression, and the type information connected with each operator is given.

In addition, the conditions under which an expression is type consistent are presented. Just as syntactic rules ensure that an expression can be parsed in only one way, type consistency rules ensure that every expression has exactly one meaning.

We include in our syntax the programming language portions of PL/CV3. Although the proof rules for these constructs are not described in this report, their construction is straight-forward, being similar to those of PL/CV2.

We first give a portion of syntax that is used frequently.

```
<list of typed variables> ::=   <list of typed variables> , <typed unit>
                              | <typed unit>

             <typed unit> ::=   <identifier> [ ε ] <expression>
                              | ( <list of identifiers> ) [ ε ] <expression>

    <list of identifiers> ::=   <list of identifiers> , <identifier>
                              | <identifier>

        <expression list> ::=   <expression list> , <expression>
                              | <expression>
```

2.2.1. The highest precedence operations in PL/CV3 are operation application, parenthesized expressions, and selection from ordered tuples.

```
<expression> ::=    <expression> . <expression>
                  | ( <expression> )
                  | <expression> ( <expression list> )
```

Note that the operations mentioned in the previous section, such as PROD, UNION, REC, etc., are examples of expression applications when supplied with appropriate arguments.

An expression of the form "f(a)" is type consistent if both f and a are type consistent, and f is of type $\bigwedge y \in A.B(y)$ for some types A and B, and a is of type A.

An expression of the form "a.b" is type consistent if a and b are type consistent, and a is of type

$$PROD(x_1{:}X_1, x_2{:}X_2(x_1), \ldots, x_n{:}X_n(x_1, \ldots, x_{n-1}))$$

for appropriate $X_1, \ldots, X_n$, and b is of type $\bigvee t \in INT.1 \le t \le n$.

2.2.2. $\bigwedge$, $\bigvee$, $\{x \in A \mid B(x)\}$, {<list of identifiers>}, $\neg$, -(unary): These are all unary or bracketing operators, and hence have equal precedence. However, their syntax and type requirements are different.

2.2.2.1. $\bigwedge$, $\bigvee$, $\{x \in A \mid B(x)\}$: These operators are constructive universal ($\bigwedge$) and existential ($\bigvee$, $\{x \in A \mid B(x)\}$) quantification. The words SOME and ALL are synonyms for $\bigvee$ and $\bigwedge$, respectively. The form $\{x \in A \mid B(x)\}$ is another way of saying $\bigvee x \in A.B(x)$.

```
<expression> ::= [Λ, V] <list of typed variables> . <expression>
<expression> ::= { <identifier> [ ϵ ] <expression 1> | <expression 2> }
```

An expression of the form $\bigwedge x \in T.S(x)$ is type consistent (similarly for $\bigvee$ or $\{x \in T \mid S(x)\}$) if T and S are type consistent, T is of type TYPE(i) for some positive i, and S is of type $T \rightarrow TYPE(j)$ for some positive j.

The variables appearing in the list are called <u>bound</u> variables, their scope being the expression over which they are quantified. Note that the <expression> is the shortest expression consistent with this precedence structure. Thus $\bigvee(x, y) \in INT.x = y$ is equivalent to $(\bigvee(x, y) \in INT.x) = y$, which is not legal, as the bound x does not evaluate to a type for every (or indeed any) element of the type INT.

2.2.2.2. {<list of identifiers>}: This syntax provides for the introduction of enumeration types, that is, for finite types whose elements are given by the identifier names in the list. The occurence of a name in the list constitutes a new definition for the name. The value associated with the name is an unspecified individual which is an element of the enumeration type. Equality between the members of the type is decidable.

2.2.2.3. ¬: This is the "not" operator, which acts either as the PL/1 boolean operator (when given a boolean operand), or as the constructive negation operator when applied to a type. As the negation operator, ¬A is equivalent to $A \Rightarrow VOID$.

¬A is type consistent if A is type consistent and of type TYPE(i) for positive i, or of type BOOL.

2.2.2.4. -: The unary "-" is the integer negation operator, with the usual semantics.

2.2.3. SECTION...END: Sections in a mathematical text are modeled by using the SECTION construction. The contents of a section is a list of definitions and theorems, and in fact looks much like the contents of a product type. The major use of a section is for the gathering of definitions into logical groups. Sections cannot be made recursive, so they provide a convenient grouping facility for mutually recursive operations.

```
<expression> ::= SECTION;
                        <list of expressions or definitions>;
                  END
<list of expressions or definitions> ::=
                 <list of expressions or definitions> ;
                 <expression or definition>
               | <expression or definition> .
<expression or definition> ::=
                 <expression>
               | DEFINE [INFIX] <identifier> = <expression>
```

2.2.4. PROCEDURE...END, PROOF...QED, (ARB...): A procedure definition is a form of operations definition, as is the notion of a proof block. Three different, but synonymous methods of operation definition are supplied, mainly for historical and readability reasons. PROCEDURE is the general notion; PROOF is included to allow more readable mathematical divisions; (ARB...) allows a simplified "statement function" capability. (Since proof blocks are only necessary for types formed by $\rightarrow$ and Λ, they are only used for operation construction. Types formed by the other type operators do not need such proof blocks in order to form elements of the constructed type.)

```
<expression> ::= [<identifier>:] {PROCEDURE | PROOF | ( }
                                  ARB <list of typed variables>;
                                  [ATTAIN <expression>;]
                                  <body>;
                              {END | QED | ) }
```

The list of variables are the parameters to the operation.

The value of these expressions is the operation thus produced, which can itself be assigned to a variable, passed as a parameter, or used as a subexpression in a larger expression. The syntax for the body of a procedure is given in the next section.

2.2.5. *, /: These are the PL/1 arithmetic operations of multiplication and division.

```
<expression> ::=   <expression> * <expression>
                 | <expression> / <expression>
```

x * y is type consistent for all type consistent x and y of type INT, and x / y is type consistent for all type consistent x of type INT and y of type $\forall z \in$ INT.(z¬=0).

2.2.6. +, -(binary): These are the PL/1 operations of integer addition and subtraction, and character string concatenation. x + y and x - y are defined for any x and y of type INT.

```
<expression> ::=   <expression> +  <expression>
                 | <expression> -  <expression>
```

2.2.7. =, ¬=, <, >, ≤, ≥, ¬<, ¬>: The relational operations include the normal PL/1 operations; however = and ¬= have additional meaning with respect to types. See section 2.5 for further discussion of the role of equality within the theory.

Relational operators can be "cascaded", that is, $A < B = C > D$ has the same meaning as $(A < B) \& (B = C) \& (C > D)$.

<expression> ::= <expression> {=, ≠, <, >, ≤, ≥, ≮, ≯} <expression>

$x = y$ and $x \neq y$ are type consistent for any type consistent x and y of the same type, say T. They then denote the type (proposition) that x is equal to y (not.equal to) in type T, denoted later in the rules as $x =_T y$ $(x \neq_T y)$. Expressions involving any other relational operators are type consistent if the arguments are type consistent and both of type INT or CHAR.

2.2.8. ⇒, →: These operations form the type of operations from a given type to another. The ⇒ can be considered as logical implication, and the → as operating on types, but the two are really identical. $A \Rightarrow B$ (or $A \rightarrow B$) is type consistent, but possibly empty, for all type consistent A and B, where A is of type TYPE(i) and B is of type TYPE(j) for $i, j > 0$.

```
<expression> ::=  <expression> ⇒ <expression>
                | <expression> → <expression>
```

2.2.9. ⇔: This is the logical equivalence operation. $A \Leftrightarrow B$ denotes a type whose elements are pairs of maps, and is true (non-empty) if and only if $A \Rightarrow B$ and $B \Rightarrow A$ are true (i. e., non-empty types). For propositions, the meaning is the usual (constructive) meaning of ⇔. For arbitrary types, $A \Leftrightarrow B$ intuitively means that A and B are in some sense isomorphic.

<expression> ::= <expression> ⇔ <expression>

A $\Leftrightarrow$ B is type consistent for all type consistent A and B where A is of type TYPE(i) and B is of type TYPE(j) for i.j > 0.

2.2.10. &, ×: These are the normal propositional "and". A & B is equivalent to A × B for arbitrary types A and B.

<expression> ::= <expression> [&, ×] <expression>

A × B is type consistent for all type consistent A and B where A is of type TYPE(i) and B is of type TYPE(j) for i,j > 0.

2.2.11. / : Quotient types are formed with this operator. T / f is to represent the types of the equivalence classes of the type T under the equivalence relation f.

<expression> ::= <expression> / <expression>

T / f is type consistent for T type consistent and of type TYPE(i) for i > 0, and f type consistent such that f describes an equivalence relation on T (i. e. f is symmetric, reflexive, and transitive). See section 3.7 for further rules on quotients.

2.2.12. ∨, + : These are the constructive versions of the propositional "or". A ∨ B is equivalent to A + B for arbitrary types A and B.

<expression> ::= <expression> [∨ , +] <expression>

A + B is type consistent for all type consistent A and B where A is of type TYPE(i) and B is of type TYPE(j) for i,j > 0.

2.2.11. The type restrictions given in the above paragraphs define the *pure* PL/CV3 type consistency requirements. Care must be taken when using them -- it is not always obvious whether an expression is type consistent or not. For example, as an attempt to create the type containing all elements of non-empty small types, one might write

$$\bigvee x \in (\bigvee y \in \mathrm{TYPE}.y).x$$

but this expression is not type consistent. The type $\bigvee y \in \mathrm{TYPE}.y$ is indeed "the type of all non-empty small types", but a more literal interpretation shows that its members are themselves *pairs* of objects of the theory, and hence individuals and not types. So the second occurrence of x should be "(x . 1)" in order to specify the type contained in the pair, rather than the individual which is the pair.

2.3. Building Functions from Commands

The commands used in PL/CV3 are taken directly from PL/CV2. Assignment statements, typings, IF, various forms of DO, SELECT, CALL, GOTO (forward only), and LEAVE, and statement composition (via the ";" operator) are all valid forms of creating operations within PL/CV3. In addition, because of the identification of operations and proofs, an operation can be built in a form that looks more like a proof than a procedure.

```
        <body> ::=   <body> ; <statement>
                   | <statement>
   <statement> ::=   <command>
                   | <expression>
                   | <typing>
```

2.3.1. := : The assignment operator in PL/CV3 is a command that modifies the frame of reference in which the system is operating, by changing the value assigned to a component of the frame of reference.

```
   <command> ::=   <identifier> := <expression>
```

2.3.2. IF: The usual form of an IF statement takes a boolean expression and two other expressions, evaluating one or the other depending on the value of the boolean.

```
<command> ::=    IF <expression>
                 THEN <command>
                 ELSE <command>
                 FI
```

In PL/CV3, we have extended the interpretation of IF statements to allow

<expression> to be a more general type, namely, a decidable type, that is, one for which <expression> ∨ ¬<expression> has been shown to be a non-empty type.

2.3.3. DO: From PL/1, there are four types of DO constructs. The first is a simple grouping operation.

```
<command> ::=    DO;
                    <body>;
                 END
```

The other three forms of the DO are repetition constructs. The indexed DO loop always halts if the body of the loop halts, and hence does not need a separate introduction proof.

```
<command> ::=   DO <identifier> = <expression 1> TO <expression 2>
                                             [BY <expression 3>];
                      <body>;
                END
```

<expression i> for 1 <= i <= 3 must be of type INT. The value of the identifier cannot be changed by any command within <body>.

The DO UNTIL and DO WHILE forms are the general loop constructs. They are special forms of recursive operation definitions.

```
<command> ::=   DO WHILE(<expression>);
                   ARBITRARY r R;
                   <body>;
                END
<command> ::=   DO UNTIL(<expression>);
                   ARBITRARY r R;
                   <body>;
                END
```

In both cases, <expression> must be a decidable type, as in the IF statement. The type R is a recursively defined type; the variable r is used in the proof of termination of the loop.

2.3.4. SELECT: This statement is a multi-way IF. The expressions a_i must be decidable, as in an IF statement. That is, the type $a_1 \vee \cdots \vee a_n \vee (\neg a_1 \mathbin{\&} \cdots \mathbin{\&} \neg a_n)$ must have an element.

```
<command> ::=   SELECT:
                WHEN(<expression a1>) <command b1>;
                WHEN(<expression a2>) <command b2>;
                . . .
                WHEN(<expression an>) <command bn>;
               [OTHERWISE <command bn+1>;]
                END
```

2.3.5. CALL: The CALL statement is a means of specifying parameters to a procedure. A procedure is an operation that returns no explicit result, but rather maps a local frame of reference onto itself, modifying the values of variables passed as parameters.

```
<command> ::= CALL <identifier> ( <expression list> )
```

2.3.6. GOTO, LEAVE: A discussion of these commands, which (in the usual interpretation of PL/1) are used for abrupt changes of execution order, must include a mention of the use of labels in a procedure. Since procedure definitions are values in the language and can be passed as parameters and operated on by built-in operations, we restrict the labels accessible by a GOTO or LEAVE to those defined within the operation containing the GOTO or LEAVE itself. Thus the procedures

```
PROCEDURE                          PROCEDURE
      LEAVE;                             GOTO X;
END                                END
```

are null procedures, no matter what label is substituted for X.

The set of labels and GOTO statements in a procedure can be viewed as providing a set of mutually recursive definitions. The easiest way to understand the treatment of GOTO's and labels is to view them as in [van Wijngaarden, ????]. GOTO's and labels can be removed from a procedure, turning the procedure into a set of smaller, recursively defined procedures. Thus it suffices to be able to handle a set of mutually recursive procedure definitions.

LEAVE statements are very similar to GOTO statments, but the label involved is often implicit, rather than explicitly mentioned. To simplify matters further, we agree that our language will not support backward GOTO's. Thus the set of procedures obtained by following the van Wijngaarden process will not be recursive, except for those explicitly involving DO loops.

```
<command> ::=   GOTO <identifier>
              | LEAVE [<identifier>]
```

2.3.7.

;: The ';' operator, or statement separator of PL/1, is an operation composition operator. Since statements (commands) in PL/CV3 are operations from the frame of reference (a product type) to the frame of reference, treating statement separation as operation composition preserves the usual semantics of the programming language. That is, S1; S2 means Meaning(S2)(Meaning(S1)(state)) for statements S1 and S2. The syntax for ";" is shown above as

```
<body> ::= <statement> ; <body>
```

2.1.8. ϵ : Typings. although not part of the theory itself (that is. the expression "$x \epsilon y$" is not itself an object), are a useful piece of syntax carried over from PL/CV2. Statements of the form "y BY ALLEL, F, x;" expresses exactly that the type y is proven by the value f(x), where f is an element of the type F. From a semantic point of view, one can think of the statement "$x \epsilon y$" as shorthand for TYPEOF(x) = y.

<typing> ::= <expression 1> ϵ <expression 2>

Interested readers are referred to the PL/CV2 reference manual for the syntax of the older forms of typings.

2.4. Block Structure and Variable Bindings

Block structure in PL/CV3 is similar to that of PL/1, with the addition of several new block constructs. A block is a syntactic structure in which (possibly new) identifiers can be introduced and bound to a type for the duration of some syntactically defined extent, possibly hiding earlier binding occurrences of the same identifier. The familiar versions of PROCEDURE's and PROOF's are all considered blocks in PL/CV3. In addition, quantified expressions (i. e. those which begin with the quantifiers $\wedge$ and $\vee$) are blocks. In the following discussion of blocks, the symbols '{' and '}' will be used as arbitrary block beginning and ending symbols, and ';' as the separation between the declarations and the body of the block.

In a block { <declaration list>; <expression> }, the group of declarations at the front is called the block heading. Identifiers declared in the block heading are constrained to reference only values of the type with which they are declared, within the scope of the identifier declaration. The scope of an identifier declaration is defined as all of <expression> with the exception of those blocks contained in <expression> which themselves contain a declaration of the same identifier.

A parameterized definition with parameter 'x' is said to be 'free for z' (for some expression z) if on substituting 'z' for all occurrences of the parameter, no occurrence of any identifier appearing in the parameter is 'captured' by a block heading; that is, if no identifier appears both in the parameter (bound to some external block heading) and in a block heading within whose scope the substitution is being made. If a parameterized block b(x) is free for z, then writing b(z) is exactly equivalent to writing the expansion of b with the syntactic substitution performed. If b(x) is not free for z, then in writing the expansion of b(z), some identifier in z will be bound by an internal block heading in b, rather than the external heading by which it was originally bound. This phenomenon, called capture, may result in proofs which the user may think are correct, but are not, due to a capture of a variable. This can be avoided by systematically renaming internal bound identifiers so they do not conflict with any identifier in the parameter. For example, suppose we have the definition (for an appropriate definition of inj that takes an element of $\bigvee z \in \mathrm{INT}.(1<z<x)$ into the corresponding integer)

```
prime: PROCEDURE
     ARB x ∈ INT;
     ATTAIN ⋀y ∈ (⋁z ∈ INT.(1 < z < x)) . ¬div(inj(y), x);
END
```

in the block

```
PROCEDURE
   ARB y INT;
   y := 5;
   prime(y) BY <proof text>;
   ⋀y ∈ (⋁z ∈ INT.(1 < z < y)) . ¬div(inj(y), y);
END
```

Even though prime(y) has been proven, the next line, which looks as though it is the systematic replacement of y for the parameter, does not have the same meaning because the parameter has been captured by the (internal) quantifier. This can be remedied by changing the line to

$$\bigwedge q \in (\bigvee z \in INT.(1 < z < y)) \, . \, \neg div(q, y);$$

This example is rather blatant. As a user builds up a complicated theory with many definitions and types, capture becomes harder to detect.

2.5. Type Equality

The intensional view of types requires a decidable equality (see [Constable, 1980]). Type equality in PL/CV3 is built up from four simpler notions, denoted $=_r$, $=_\alpha$, $=_\beta$, and $=_\delta$, into a decidable relation.

The most primitive form of equality is reflexive equality, $=_r$. $T =_r T$ holds for all types T, that is, every type is equal to itself. Since we are working within a strongly intensional theory, a type is not necessarily equal to a syntactically different expression which "evaluates" to the same type. (This sort of equality can be obtained in the system -- see section 3.7 on quotient types.)

Alpha conversion, the renaming of bound variables, defines the next form of equality, $=_\alpha$. Any expression denoting a type is alpha-equivalent to the same expression where bound variables have been systematically renamed (without capture). For example

$$(\Lambda x \in \mathrm{INT}.P(x)) =_\alpha (\Lambda y \in \mathrm{INT}.P(y))$$

This form of equality applies to all bound variables described in the previous section.

Beta conversion is the third form of equality, $=_\beta$. An element of a functional type applied to an element of the parameter type is equal under beta conversion to the body of the operation where the formal parameter is replaced by the actual parameter. Say f(x) is defined to be body(x) for x in some type T, then $f(t) =_\beta \mathrm{body}(t)$ is an application of beta conversion, where $t \in T$. Note that beta equality assumes that block headings internal to body have been renamed (by alpha equality) to avoid capture of the parameter 't'.

The last form of equality on types is that of definitional equality, $=_\delta$. If an identifier A is defined to designate some type T, then A $=_\delta$ T holds.

The equality actually implemented in the system is best described in terms of the transitive closure of the union of these four forms of equality. Let $=_{basis}$ denote $=_r \cup =_\alpha \cup =_\beta \cup =_\delta$. Then = in PL/CV3 is generated from the transitive closure of $=_{basis}$, by the operation of the substitution property, which states that if A=B holds, then exp(A)=exp(B) holds, where exp(x) is any expression. Thus we obtain statements such as C=D from the hypotheses C = T1 $\times$ T2, T2=T3, and D = T1 $\times$ T3 by repeated application of definitional equality and substitution to one instance of reflexive equality, that being T1 $\times$ T2 = T1 $\times$ T2.

2.6. Type Coercions

As was demonstrated at the end of section 2.2, one can write expressions which intuitively mean what one wants them to mean, but which, because of the pure type consistency requirements, are nevertheless incorrect. One approach would have been to restrict the syntax to a much greater degree; we have chosen rather to allow many "obviously incorrect" expressions to be used, easing the burden on the user by providing (and allowing the user to provide) type coercions enabling the system to deduce the (or a) correct interpretation for an expression which would be incorrect under the pure type consistency requirements.

Automatic type coercions have often been used in programming languages, sometimes to the dismay of programmers who have had to struggle to understand exactly what a compiler has done to their code. PL/1 is the most commonly used language with large numbers of built-in type coercions -- integers can be coerced to reals, character strings, and bit strings without any programmer intervention. The controversy over "strong" and "weak" typing in programming langusges is mainly one of coercion application. Only recently have languages been designed to allow users to specify what coercions they would like to use when converting from one type to another; only recently, with the advent of languages such as Russell, Ada, and others which implement "abstract data types" has the number of data types a programmer must deal with multiplied to the point where automatically applied but user specified "transfer functions" have become necessities. Our approach allows the flexibility of weak typing with the correctness and error checking of strong typing; the interactive implementation hides nothing from the user -- at any time, the user may ask to see the (set of) coercions being applied to any expression, as the system sees it.

A "coercion" from type A to type B in PL/CV3 is an injection $inj_{A\to B}$ such that $inj_{A\to B}(a)\in B$ for $a\in A$, together with an inverse $inj^{-1}_{A\to B}$ such that $inj^{-1}_{A\to B}(inj_{A\to B}(a))$ = a for all $a\in A$, but $inj^{-1}_{A\to B}(b)$ is not necessarily defined for all $b\in B$. Some coercions are built into the system; users can specify any other coercions they want between any types they are using. It should be emphasized that the pure type consistency requirements are still in effect; coercions are only a means to reduce what needs to be entered by the user to obtain a correct program/proof.

Two paths can be taken when the system is applying coercions. One approach is to insist, as in the syntax and consistency section, that there be only one correct interpretation of any expression. In this mode, the system will reject an entry if either it knows of no way to interpret it as a type consistent expression, or if there is more than one way to interpret it, through the use of one or more coercions. In either case it would require the user to clarify the expression until there was exactly one valid interpretation. The other, more flexible approach is to that there be <u>at least</u> one valid interpretation, rather than exactly one. The system will keep track of a set of meanings for any ambiguous expressions, removing possible meanings as context shows them to be incorrect.

3. Proof Rules

The rules of the system determine what objects can be built, specify their types (including those operators and tests that can be applied to objects), and provide operations whereby objects can be taken apart. The rules for building objects are called *introduction* rules; those for taking objects apart are called *elimination* rules. There is an introduction and an eliminition rule for each way of building objects (including types), thus rules for $\rightarrow$, Λ, and members of these function types built by *projection*, *composition*, *conditional*, and *recursion*; for $+$, $\mathbf{V}$ and members of these types built by *injection*; for the base types and their members, etc. Rules for equality between individuals are also presented.

Note that there are no free variables in this theory -- all variables must have a binding occurrence which supplies a type for the variable, although such binding occurrences may be lexically very far away. The function of the frame of reference is to record such binding occurrences for use in obtaining the meaning of an expression within the scope of the declarations that are the binding occurrences.

3.1. Rule format

Rules are presented here in three kinds, instead of two as is usual in natural deduction systems. The two ordinary kinds, introduction and elimination, are of course used. The third, definition, is used mainly for clarity in exposition when dealing with the rules operator by operator. It is really a form of introduction rule for a larger type, but does not logically belong there, as we are attempting to keep all the rules for one operator grouped together. A definition rule gives the type of a construct that uses the operator, given that its operands are of specified types.

We present rules for the major type operations: $\bigvee$, $\bigwedge$, and $\mathbb{R}$, as well as the type constants. Rules for the variants ($+$, $\times$ and $\rightarrow$) can be derived quite simply. Rules are given in the form

$$\frac{\text{list of hypotheses}}{\text{list of conclusions}}$$

Hypotheses and conclusions are usually of the form "element ϵ type". The ϵ relation is not expressible in the syntax, nor can one reason about it in its full generality within the system. (Part of the relation is available in the form of the TYPEOF operation (see below), and the "typings" of the command language.) The concept of an element belonging to a type is a linguistic notion, and is referred to as the element's *typing*. It is used here in the presentation of the proof rules to indicate what typings must be known before a rule is applied, and what typings can be deduced after the application of a rule.

The second form of conclusion is used where a primitive operation of the logic is being defined. The operation is defined as

function(arguments) = expression

This equality is used as a reduction rule in the normalization of expressions, rather than interpreting it as an equality type.

3.2. Constants

$$\frac{}{\text{INT} \in \text{TYPE}(1)} \qquad \text{TYPE(1) intro}$$

$$\frac{}{\text{CHAR} \in \text{TYPE}(1)} \qquad \text{TYPE(1) intro}$$

$$\frac{}{\text{BOOL} \in \text{TYPE}(1)} \qquad \text{TYPE(1) intro}$$

$$\frac{}{\text{VOID} \in \text{TYPE}(1)} \qquad \text{TYPE(1) intro}$$

$$\frac{}{\text{TYPE}(i-1) \in \text{TYPE}(i)} \text{ for } i \geq 2 \qquad \text{TYPE(i) intro}$$

$$\frac{}{n \in \text{INT}} \text{ for any integer } n \qquad \text{INT intro}$$

$$\frac{}{\text{'c'} \in \text{CHAR}} \text{ for any character } c \qquad \text{CHAR intro}$$

$$\frac{}{\text{True} \in \text{BOOL}} \qquad \text{BOOL intro true}$$

$$\frac{}{\text{False} \in \text{BOOL}} \qquad \text{BOOL intro false}$$

These are the introduction rules for the constant types; there are no elimination rules. Note that VOID has no members; it is used as a specific empty type in the proof rules. Empty types of various levels can be easily built:

$$\text{TYPE}(i) \times \text{VOID}$$

which we will denote by VOID_{i+1}, is an empty type which is a member of TYPE(i+1) for $i \geq 1$. Thus VOID_i is an element of TYPE(i); VOID is understood to represent the first element of this list, that is, VOID_1.

In the proof rules, the term **axiom** refers to an element of a type (usually a proposition) in the conclusion of a rule which can be supplied by the system as an individual depending on the syntactic form of the terms involved in the application of the rule. For example, in the rule

$$\frac{\text{el} \in \text{expl}}{\textbf{axiom} \in \text{f(el)}} \qquad \text{rule name}$$

axiom is more properly represented by $\textbf{axiom}_{\text{el.expl.f(el).rule name}}$.

The conditions of an expression being type consistent are the hypotheses for a rule stating that the type itself has a type.

$$\frac{\text{exp is type consistent}}{\text{exp} \in \text{TYPE(i) for appropriate i}} \qquad \text{TYPE(i) intro}$$

It is assumed from here on that any expressions used in proof rules are at least type consistent; that is, any proof rule mentioning an arbitrary expression "exp" has as an implicit hypothesis "exp is type consistent".

One can reason about the type of expressions by using the TYPEOF operation; its meaning is given by

$$\frac{\text{expl} \in \text{exp2}}{\text{TYPEOF(expl) = exp2}} \qquad \text{TYPEOF intro}$$

In addition to the type formation rules, there is a rule that allows one to deduce the level of any type expression. In particular, it introduces the operation LEVEL:

$$\frac{\text{exp} \in \text{TYPE(i)}}{\text{LEVEL(exp) = i}}\ i \geq 1 \qquad \text{LEVEL intro}$$

Note that this operation, and most others presented in this chapter, are actually parameterized by the level of their arguments. There are a denumerably infinite number of operations with name LEVEL; the proper one is chosen by the system, depending on the level of the argument. In this case, for a type of type TYPE(i), the proper LEVEL will be an element of TYPE(i) $\rightarrow$ INT.

The following rule states that an element of TYPE(i) for some i can be injected into an element of TYPE(j) for any j>i.

$$\frac{T \in \mathrm{TYPE}(i)}{inj_{\mathrm{TYPE}(i) \rightarrow \mathrm{TYPE}(j)}(T) \in \mathrm{TYPE}(j)} \qquad \text{TYPE(i) intro}$$

3.3. Equality types

Given any two elements of a type, the proposition that those two elements are identical is itself a type.

$$\frac{\begin{array}{l} T \in \mathrm{TYPE}(i) \\ x \in T \\ y \in T \end{array}}{\begin{array}{l} (x =_T y) \in \mathrm{TYPE}(i) \\ \mathbf{axiom} \in \mathrm{IsEquality}(x =_T y) \\ (\mathrm{LeftSide}(x =_T y) = x) \\ (\mathrm{RightSide}(x =_T y) = y) \end{array}} \qquad = \text{def'n}$$

That is, the operation $=_T$ has type $(T \times T) \rightarrow \mathrm{TYPE}(i)$.

The introduction rules for equality types are spread throughout the rest of the rules, as they are often elimination rules for other type constructors. The elimination rule for equality types states that when two objects are proven equal, they may be substituted for one another in any expression to give equal results, as was spelled out in section 2.5.

3.4. Union

Special cases of type unions give finite disjoint unions, cartesian products, and disjoint unions of an infinite collection of types. A binary union operation is presented, but the extension to n-ary unions is simple. The binary union type is represented as $\bigvee x \in T.S(x)$

The members of the dependent product are created from the members of the types in the collection being unioned. Since this theory of types maintains that every object has a type, we cannot use the objects directly, but rather must "hide" their values via an injection, which tags an object with the type into which it is being placed. The union introduction rule specifies the conditions under which a union type can be shown non-empty, and defines the selector operation for deciding which part of a union a member was originally in.

$$\frac{\begin{array}{c} e \in T \\ f \in S(e) \end{array}}{inj_{e, \bigvee x \in T.S(x)}(f) \in \bigvee x \in T.S(x)} \qquad \text{Union intro}$$

$$inj^{-1}_{e, \bigvee x \in T.S(x)}(inj_{e, \bigvee x \in T.S(x)}(f)) = f$$

$$(inj_{e, \bigvee x \in T.S(x)}(f)).1 = e$$

$$(inj_{e, \bigvee x \in T.S(x)}(f)).2 = f$$

Note that the inverse injection does not really have a range of all of the union type, but rather only those which were themselves formed by application of the injection.

The union elimination rule allows us to take a constructed member of a union apart into its components.

$$\frac{e \in \bigvee x \in T.S(x)}{\begin{array}{c} e.1 \in T \\ e.2 \in S(e.1) \end{array}} \qquad \text{Union elim}$$

The union definition rules give the meaning of the operations IsUnion, First and Second, which are part of the type system because of the intensional treatment of types.

$$\frac{\begin{array}{l}T \in \mathrm{TYPE}(i),\\ S \in T \rightarrow \mathrm{TYPE}(j)\end{array}}{\begin{array}{l}\mathsf{V}x \in T.S(x) \in \mathrm{TYPE}(\max(i,j))\\ \mathbf{axiom} \in \mathrm{IsUnion}(\mathsf{V}x \in T.S(x))\\ \mathrm{First}(\mathsf{V}x \in T.S(x)) = T\\ \mathrm{Second}(\mathsf{V}x \in T.S(X)) = S\end{array}} \qquad \text{Union def'n}$$

Finally, we must have axioms for equality: both equality between members of union types, and equality between types constructed with the V operation. Two types formed by V are equal (in TYPE(max(i,j))) if their index types are equal (in TYPE(i)) and the operations used to choose types to be unioned are equal (in "index type $\rightarrow$ TYPE(j)").

$$\frac{\begin{array}{l}x \in (T_1 =_{\mathrm{TYPE}(i)} T_2)\\ y \in (F_1 =_{T_1 \rightarrow \mathrm{TYPE}(j)} F_2 \quad)\end{array}}{\mathbf{axiom} \in (\mathrm{SOME}x \in T_1.F_1(x) =_{\mathrm{TYPE}(\max(i,j))} \quad \mathsf{V}x \in T_2.F_2(x))} \qquad \mathrm{TYPE} =$$

Elements of union types are equal if they are formed by applying injections to equal elements of the base types involved.

$$\frac{\begin{array}{l}x \in (a_1 =_T a_2)\\ y \in (b_1 =_{F(a)} b_2 \quad)\end{array}}{\mathbf{axiom} \in (\mathrm{inj}_{a_1}.\mathsf{V}z \in T.F(z)(b_1) =_{\mathsf{V}z \in T.F(z)} \quad \mathrm{inj}_{a_2}.\mathsf{V}z \in T.F(z)(b_2))} \qquad \mathsf{V}=$$

To construct ordinary finite disjoint unions, it is sufficient to take a finite type T (say, for example the type containing the integers 1 and 2, or BOOL) and an operation B of type $T \rightarrow$ TYPE(k) (for appropriate k) such that B(t) takes on as values all the types wanted in the union as t varies over the elements of T. Then the type $\mathsf{V}t \in T.B(t)$ is exactly the union of the types enumerated by B.

The proof rules for finite disjoint unions parallel those for disjunction introduction and elimination in constructive logic, and can be derived from the above rules. The proposition "$A \vee B$" can be represented by the type $\bigvee i \in \{1,2\}.C(i)$, where $C(1) = A$ and $C(2) = B$. Given a member x of this representation of "$A \vee B$" (a proof of $A \vee B$) we can decide which of A or B was actually proven by looking at the value of "x.1". If it is 1, then A was proven, and if it is 2, then B was proven. Given this information, it is easy to write a version of the constructive "or" elimination rule.

Simple cartesian product types are obtained by writing a union type such as

$$\bigvee x \in T_1.(\bigvee x \in T_2.\cdots(\bigvee x \in T_n.T_{n+1})\ldots)$$

which is abbreviated as

$$T_1 \times T_2 \times \cdots \times T_n \times T_{n+1}$$

where the types T_i in the construction are such that T_i does not depend on any of the preceeding T_j. The selector mechanism can then be used for picking out elements of a member of such a product type. For x a member of such a type, x.1 is an element of T_1, x.2 an element of $T_2 \times \cdots \times T_{n+1}$, and so on.

3.5. Recursion

A recursive type definition is based on an operation f from types to types (of a specific level, say i). The recursive type itself is referred to as $\mathbf{R}(f)$, and is of level i. However, associated with a recursive definition is a type of one higher level, referred to as the language of the definition (denoted Lang(f)). Intuitively, Lang(f) has as members types formed from the recursion equation given in the definition of the type; $\mathbf{R}(f)$ has as members all elements of all types in Lang(f). Note that recursive types are defined for all operations f, whether they are (in some sense) monotonic, or continuous, or neither. The only restriction we make on f is a syntactic one, for technical reasons, that f may not refer to the intensional operations on recursive types.

We first give the conditions under which a recursive type can be defined. Recursive types can be made mutually recursive. In the most general case, each recursive type being defined can depend on itself in several ways, and on the other types being defined in other numbers of ways; the i^{th} type of the recursive definition depends on $p_{i,j}$ copies of the j^{th} type.

$$\frac{f_i \in \mathrm{TYPE}(k)^{p_{i,1}} \rightarrow \ldots \rightarrow \mathrm{TYPE}(k)^{p_{i,n}} \rightarrow \mathrm{TYPE}(k),\ 1 \le i \le n}{\begin{array}{l}\mathbf{R}_j(f_1,\ldots,f_n) \in \mathrm{TYPE}(k),\ 1 \le j \le n \\ \mathbf{axiom} \in \mathrm{IsRecursive}(\mathbf{R}_j(f_1,\ldots,f_n)).\end{array}} \quad \text{rec. def'n}$$

We assume for the remainder of the rules for recursion that the hypotheses

$$\begin{array}{c} f_i \in \mathrm{TYPE}(k)^{p_{i,1}} \rightarrow \ldots \mathrm{TYPE}(k)^{p_{i,n}} \rightarrow \mathrm{TYPE}(k),\ 1 \le i \le n \\ \mathbf{R}_j(f_1,\ldots,f_n) \in \mathrm{TYPE}(k) \end{array}$$

are added to any other hypotheses explicitly listed for the rule.

The recursive language can be introduced immediately (under the above assumptions). Note that Lang(f) is also a function of the syntactic form of the operations supplied. That is, if two recursive types define trees, one in the presence of other recursive definitions and one not, the two language types thus defined are not equal.

$$\frac{}{\mathrm{Lang}_i(f_1,\ldots,f_n) \in \mathrm{TYPE}(k+1),\ 1 \le i \le n} \qquad \text{Lang def'n}$$

$$\textbf{axiom} \in \mathrm{IsRecLang}(\mathrm{Lang}_i(f_1,\ldots,f_n))$$

$$\mathrm{RecFn}(i,\ \mathrm{Lang}_j(f_1,\ldots,f_n)) = f_i,\ 1 \le i \le n,$$

$$\mathrm{NumRecEq}(\mathrm{Lang}_j(f_1,\ldots,f_n)) = n$$

$$\mathrm{MajorRec}(\mathrm{Lang}_j(f_1,\ldots,f_n)) = j$$

Members of the language are formed by an iterative process. A basis element (an image of VOID_k under an injection) is assumed to be in the type.

$$\frac{\mathrm{Lang}_i(f_1,\ldots,f_n) \in \mathrm{TYPE}(k+1)}{\mathrm{inj}^{-1}_{\mathrm{Lang}_i(f_1,\ldots,f_n) \to \mathrm{TYPE}(k)}(\mathrm{VOID}_k) \in \mathrm{Lang}_i(f_1,\ldots,f_n)} \qquad \text{Lang intro}$$

$$\mathrm{inj}_{\mathrm{Lang}_i(f_1,\ldots,f_n) \to \mathrm{TYPE}(k)}$$

$$(\mathrm{inj}^{-1}_{\mathrm{Lang}_i(f_1,\ldots,f_n) \to \mathrm{TYPE}(k)}(\mathrm{VOID}_k)) = \mathrm{VOID}_k$$

From this element, new elements can be constructed according to the recursion described by the operations. Function f_i expects $p_{i,j}$ independent types from outputs of f_j as arguments.

$$\frac{e_{i,j} \in \mathrm{Lang}_i(f_1,\ldots,f_n),\ 1 \le i \le n,\ 1 \le j \le p_{i,j}}{\mathrm{inj}^{-1}_{\mathrm{Lang}_i(f_1,\ldots,f_n) \to \mathrm{TYPE}(k)}(f_i(e_{1,1},\ldots, e_{1,p_{i,1}},\ldots, e_{n,p_{i,n}})) \in \mathrm{Lang}_i(f_1,\ldots,f_n)} \qquad \text{Lang intro}$$

An elimination rule tells us what can be deduced by assuming we are given an object of a type. For the language of a recursive definition, the rule provides information on how the member was formed.

$$\frac{\begin{array}{l} e \in Lang_i(f_1,\ldots,f_n) \\ \text{Let } inj_i \text{ denote } inj_{Lang_i(f_1,\ldots,f_n)\rightarrow TYPE(k)} \end{array}}{\begin{array}{l} \mathbf{axiom} \in (inj_i(e) = inj_i^{-1}(VOID_k)) \vee \\ \quad (\forall e_{1,1} \in Lang_1,\ldots, e_{n,p_{n,i}} \in Lang_n . \\ \quad\quad inj_i^{-1}(f_i(inj_1(e_{1,1}),\ldots, inj_n(e_{n,p_{n,i}}))) = e) \end{array}} \quad \text{Lang elim}$$

Having shown how to reason about the language of a recursive definition, we can now turn to the recursive type itself. The type is nonempty if any of the types in the language of the type are nonempty. The BASIS operation applied to an element of the type gives the element of the language type from which the element was obtained.

$$\frac{\begin{array}{l} el \in Lang_i, \\ \text{Let Basetype} = inj_{Lang_i(f_1,\ldots,f_n)\rightarrow TYPE(k)}(el) \\ e2 \in \text{Basetype} \end{array}}{\begin{array}{l} inj_{Basetype\rightarrow \mathbf{R}_i(f_1,\ldots,f_n)}(e2) \in \mathbf{R}_i(f_1,\ldots,f_n), \\ inj^{-1}_{Basetype\rightarrow \mathbf{R}_i}(inj_{Basetype\rightarrow \mathbf{R}_i}(e2)) = e2 \\ BASIS(inj_{Basetype\rightarrow \mathbf{R}_i}(e2)) = el \end{array}} \quad \mathbf{R}_i \text{ intro}$$

Similar to the $Lang_i(f_1,\ldots,f_n)$ elimination, the $\mathbf{R}_i$ elimination tells us that we can find out how an element of the type was constructed.

$$\frac{\begin{array}{l} e \in \mathbf{R}_i \\ \text{Let Basetype denote } inj_{Lang_i\rightarrow TYPE(k)}(BASIS(e)) \end{array}}{\begin{array}{l} \mathbf{axiom} \in \forall el \in Lang_i.(el = BASIS(e)) \\ \mathbf{axiom} \in \forall e2 \in \text{Basetype}.(inj_{Basetype\rightarrow \mathbf{R}_i}(e2) = e) \end{array}} \quad \mathbf{R}_i \text{ elim}$$

It only remains to give rules for deducing when two recursive types are equal, and when two elements of a recursive type are equal. In both cases strong intensionality holds; they are equal only when formed in the same way.

$$f_i \in TYPE(k)^{p_{i,1}} \to \cdots \to TYPE(k)^{p_{i,n}} \to TYPE(k),\ 1 \le i \le n$$

$$f_i' \in TYPE(k)^{p_{i,1}} \to \cdots \to TYPE(k)^{p_{i,n}} \to TYPE(k),\ 1 \le i \le n$$

$$\frac{x_i \in (f_i = f_i')}{\begin{array}{l}\textbf{axiom} \in Lang_j(f_1,\ldots,f_n) = Lang_j(f_1',\ldots,f_n') \\ \textbf{axiom} \in R_j(f_1,\ldots,f_n) = R_j(f_1',\ldots,f_n')\end{array}} \quad \text{Equality}$$

Elements of the language type have very little equality information available -- if the elements were built from equal elements, then they are equal.

$$e_{i,j} \in Lang_i,\ 1 \le i \le n,\ 1 \le j \le p_{i,j}$$

$$e_{i,j}' \in Lang_i,\ 1 \le i \le n,\ 1 \le j \le p_{i,j}$$

$$\frac{x_{i,j} \in (e_{i,j} = e_{i,j}')}{\begin{array}{l}\textbf{axiom} \in (inj_i^{-1}(f_i(e_{1,1},\ldots,e_{n,p_{i,n}})) \\ \qquad = inj_i^{-1}(f_i(e_{1,1}',\ldots,e_{n,p_{i,n}}')))\end{array}} \quad \text{Lang} =$$

Elements of the recursive type itself are slightly more interesting, as the equality on which the element equality is based is itself an equality on types which could possibly be complex.

$$\frac{e_1 \in \mathrm{Lang}_i \quad e_2 \in \mathrm{Basetype} \quad e_2' \in \mathrm{Basetype} \quad x \in (e_2 = e_2')}{\mathbf{axiom} \in (\mathrm{inj}_{\mathrm{Basetype} \to \mathbf{R}_i}(e_2) = \mathrm{inj}_{\mathrm{Basetype} \to \mathbf{R}_i}(e_2'))} \quad \mathbf{R}=$$

3.6. Operations

The basis for the creation of operations is a collection of combinators divided into several families. The C_i^j (Composition) family is a generalization of distributivity and composition. and the P_i^j (Projection) family allows selection from a list of arguments. A conditional combinator allows for more explicit construction of conditional expressions, and a recursion combinator permits recursion on various types.

Operation types are created by the Λ operation. and introduce the meaning of the Domain, Range and IsFunction operations.

$$\frac{T \in \mathrm{TYPE}(i), \quad S \in T \to \mathrm{TYPE}(j)}{\Lambda x \in T.S(x) \in \mathrm{TYPE}(\max(i,j))} \quad \Lambda \text{ def'n}$$

$\mathbf{axiom} \in \mathrm{IsFunction}(\Lambda x \in T.S(x))8$

$\mathrm{Domain}(\Lambda x \in T.S(x)) = T$

$\mathrm{Range}(\Lambda x \in T.S(x)) = S$

3.6.1. Non-recursive Operations

$$\frac{T_j \in \mathrm{TYPE}(i_j),\ 1 \le j \le n}{P_i^{T_1,\ldots,T_n} \in T_1 \rightarrow (\cdots \rightarrow (T_n \rightarrow T_i)),\ 1 \le i \le n} \quad \wedge \text{ intro}$$

$$\textbf{axiom} \in \mathrm{IsProj}(P_i^{T_1,\ldots,T_n})$$

$$(\mathrm{ProjectionNumber}(P_i^{T_1,\ldots,T_n}) = i)$$

On application, a projection operation can produce either another projection operation, or an object known as an application. The projection operation is produced if the argument the original projection operation was to produce was not the first. Otherwise, the application, which acts like a constant-valued function, is produced.

$$\frac{\begin{array}{l} P_i^{T_1,\ldots,T_n} \in T_1 \rightarrow (\cdots(T_n \rightarrow T_i)) \\ t \in T_1 \end{array}}{\begin{array}{l} P_i^{T_1,\ldots,T_n}(t) = P_{i-1}^{T_2,\ldots,T_n} \quad \text{if } i > 1 \\ P_i^{T_1,\ldots,T_n}(t) = t \quad \text{if } i = 1 \text{ and } n = 1 \\ \text{otherwise } (i = 1 \text{ and } n > 1): \\ \textbf{axiom} \in \mathrm{IsApp}(P_i^{T_1,\ldots,T_n}(t)) \\ \mathrm{AppProj}(P_i^{T_1,\ldots,T_n}(t)) = P_i^{T_1,\ldots,T_n} \\ \mathrm{AppObject}(P_i^{T_1,\ldots,T_n}(t)) = t \end{array}}$$

Given an application, it can be simplified in a manner similar to that of a projection operation.

$$x \in \mathrm{IsApp}(P_1^{T_1,\ldots,T_n}(t1))$$

$$t2 \in T_2$$

$$\frac{}{P_1^{T_1,\ldots,T_n}(t1)(t2) = P_1^{T_1,T_3,\ldots,T_n}(t1) \text{ if } n > 2}$$

$$\mathbf{axiom} \in \mathrm{IsApp}(P_1^{T_1,T_3,\ldots,T_n}(t1))$$

$$\mathrm{AppProj}(P_1^{T_1,T_3,\ldots,T_n}(t1) = P_1^{T_1,T_3,\ldots,T_n})$$

$$\mathrm{AppObject}(P_1^{T_1,T_3,\ldots,T_n}(t1)) = t1$$

$$P_1^{T_1,\ldots,T_n}(t1)(t2) = t1 \quad \text{if } n = 2$$

Similarly, we can reason about compositions:

$$f \in T_1 \to (\cdots \to (T_m \to R))$$

$$h_j \in S_1 \to (\cdots \to (S_n \to T_j)),\ 1 \le j \le m$$

$$\frac{}{C_{R,T_1,\ldots,T_m}^{S_1,\ldots,S_n}(f,h_1,\ldots,h_m) \in S_1 \to (\cdots \to (S_n \to R))}$$

$$\mathbf{axiom} \in \mathrm{IsComp}(C_{R,T_1,\ldots,T_m}^{S_1,\ldots,S_n}(f,h_1,\ldots,h_m))$$

$$\mathrm{MainComp}(C_{R,T_1,\ldots,T_m}^{S_1,\ldots,S_n}(f,h_1,\ldots,h_m)) = f$$

$$\mathrm{SubComp}(i,\ C_{R,T_1,\ldots,T_m}^{S_1,\ldots,S_n}(f,h_1,\ldots,h_m)) = h_i$$

And the elimination of such compositions:

$$C^{S_1,\ldots,S_n}_{R,T_1,\ldots,T_m}(f,h_1,\ldots,h_m) \in S_1 \to \cdots \to S_n \to R$$

$$x \in S_1$$

$$C^{S_1,\ldots,S_n}_{R,T_1,\ldots,T_m}(f,h_1,\ldots,h_m)(x_1) =$$

$$C^{S_2,\ldots,S_n}_{R,T_1,\ldots,T_m}(f,h_1(x_1),\ldots,h_m(x_1)) \quad \text{if } n > 1$$

$$C^{S_1,\ldots,S_n}_{R,T_1,\ldots,T_m}(f,h_1,\ldots,h_m)(x_1) = f(h_1(x_1))\cdots(h_m(x_1))$$

Conditional operations are created with the IF combinator. Taking three operations as arguments, it constructs a new operation which chooses between two of the operations based on the third.

$$f \in \bigwedge x \in A.(B(x) \vee \neg B(x))$$

$$g \in \bigwedge x \in A.C(x)$$

$$h \in \bigwedge x \in A.D(x)$$

$$IF(f,g,h) \in \bigwedge x \in A.(C(x) + D(x))$$

axiom $\in$ IsCond(IF(f,g,h))

Condition(IF(f,g,h)) = f

ThenPart(IF(f,g,h)) = g

ElsePart(IF(f,g,h)) = h

The execution of a conditional has the obvious property.

$$\frac{\begin{array}{l} IF(f,g,h) \in \Lambda x \in A.(C(x) + D(x)) \\ a \in A \end{array}}{\begin{array}{l} IF(f,g,h)(a) = inj_{C(a) \rightarrow (C(a) + D(a))}(g(a)) \text{ if } f(a).1 \\ IF(f,g,h)(a) = inj_{D(a) \rightarrow (C(a) + D(a))}(h(a)) \text{ if } \neg f(a).1 \end{array}}$$

3.6.2.

Recursive operations

Recursive operations must be known to terminate. They are only defined on types with a recursive structure. In PL/CV3, in addition to the built-in type INT, recursive types and their language types have enough recursive structure to permit recursive definitions of operations. Also, given the intensional nature of types and operations, recursive operations can be constructed over TYPE(i) and elements of Λ types. Finally, since the theory is constructive (and thus unions are decidable), unions over any of these structures also permit recursive definitions.

Recursive operations are built by decomposing an object of one of the above recursive types into "smaller" objects of the same type, which are either in a "base" type for which a result can be produced, or can be further decomposed. Thus the important parts of a recursive operation are: the operation which works on pieces in the base case; the decomposing operation(s) to break the value into smaller pieces; and the operation which takes the value returned by the recursive call(s) and builds a new object. We put these components together to form a recusive function by means of a recursion combinator.

The recursion combinator will be presented for single recursive operations. Mutually recursive operations can be constructed, but the basic organization is more complex, and is not described here.

We first define the class of decomposition operations for the three major types over which recursion can be performed: TYPE(i) for $i > 0$, recursive Lang types and operation types. (Decomposition operations for arbitrary unions of these types can be described, giving recursion over unions of such recursive types, but this is not done here.) The recursive operation combinator can then be given in terms of the decomposition operations.

The recursive structure of types is found in the intensional operations and their properties. Thus, in order to do recursion on the formation of a type, one must be able to break the type apart into the components out of which it was built. The collection of decomposition operations for TYPE(i) (denoted $\mathrm{DeComp}_{\mathrm{TYPE}(i)}$) is thus {Domain, Range, First, Second, RecFn, QBase, QRelation, LeftSide, RightSide and $\mathrm{inj}_{\mathrm{TYPE}(j) \rightarrow \mathrm{TYPE}(i)}$ for $1 \leq j<i$}.

Recursion on the structure of operations is similar to that on types. The decomposition operations (denoted $\mathrm{DeComp}_{\Lambda x \in A.B(x)}$) are {AppProj, AppObject, MainComp, SubComp, Condition, ThenPart, ElsePart, RecBasis, RecRepeat, RecDeComp}.

Recursion on recursive types is slightly different, and is accomplished through the use of the language type, rather than the recursive type itself. The appropriate decomposition operations (denoted $\mathrm{DeComp}_{\mathrm{Lang}}$) are {$\mathrm{inj}_{\mathrm{Lang} \rightarrow \mathrm{TYPE}(i)}$, any f in $\mathrm{DeComp}_{\mathrm{TYPE}(i)}$}.

The REC combinator can now be outlined. Let R be any of the above types, with an associated operation InBasis, which decides if a member of R is in the simplest form of the recursive type. Let $DeComp_R$ be the closure under composition (and, of course, type consistency) of the class of operations outlined above as appropriate decomposition operations for R.

$$g_1 \in \Lambda x \in \{y \in R : \mathrm{InBasis}(y)\}.B(x)$$
$$g_2 \in \Lambda x \in R.(R^n \rightarrow B(x))$$
$$h_i \in \mathrm{DeComp}_R, 1 \le i \le n$$

$$\overline{\hspace{20em}}$$

$$\mathrm{REC}(g_1,g_2,h_1,\ldots,h_n) \in \Lambda x \in R.B(x)$$
$$\mathrm{RecBasis}(\mathrm{REC}(g_1,g_2,\ldots,h_n)) = g_1$$
$$\mathrm{RecRepeat}(\mathrm{REC}(g_1,g_2,\ldots,h_n)) = g_2$$
$$\mathrm{RecDeComp}(i,\mathrm{REC}(g_1,g_2,\ldots,h_n)) = h_i,\ 1 \le i \le n$$
$$\mathbf{axiom} \in \mathrm{IsRecFn}(\mathrm{REC}(g_1,g_2,\ldots,h_n))$$

The method of formation of recursive operations guarantees that they terminate on any arguments of the proper types.

$$\frac{\begin{array}{l}\mathrm{REC}(g_1,g_2,\ldots,h_n) \in \Lambda x \in R.B(x)\\ F = \mathrm{REC}(g_1,g_2,\ldots,h_n)\\ y \in R\end{array}}{\begin{array}{l}F(y) = g_2(y,F(h_1(y)),\ldots,F(h_n(y)))\ \text{if}\ \neg\mathrm{InBasis}(y)\\ F(y) = g_1(y)\quad \text{if}\ \mathrm{InBasis}(y)\end{array}}\ \Lambda\ \mathbf{elim}$$

For recursion over TYPE(i), the InBasis proposition is defined to be

InBasis(x) $\Leftrightarrow$ x is a primitive type

For recursion over function types ($\Lambda x \in A.B(x)$), we have

InBasis(x) $\Leftrightarrow$ x is a primitive combinator of appropriate type

For recursion on $\mathbf{R}(f)$, we must have that InBasis is a decidable proposition, i.e., the type

ALL $x \in \mathbf{R}(f)$.(InBasis(x) $\vee$ $\neg$InBasis(x))

It must correspond to defining those elements in F(VOID), the "bottom" element of a language type. For those operations f built explicitly from + and ×, we can specify InBasis in the usual way. For example,

if f(x) is A(x) + B(x) then
InBasis $\Leftrightarrow$((case(x) = 1 & $\text{InBasis}_{A(x)}(\text{inj}_1^{-1}(x))$) $\vee$
(case(x) = 2 & $\text{InBasis}_{B(x)}(\text{inj}_2^{-1}(x))$))

if f(x) is A(x) × B(x) then
InBasis $\Leftrightarrow$($\text{InBasis}_{A(x)}$(x.1) & $\text{InBasis}_{B(x)}$(x.2))

if f(x) is a primitive type then
InBasis(x) $\Leftrightarrow$True
if f(x) is x then
InBasis(x) $\Leftrightarrow$False

We can also specify the decomposition operations explicitly in this case. For example, decomposition operations can take the forms given below for operations of the kind specified:

if f(x) is A(x) + B(x) then
$d_{f(x)}(y)$ = if case(y) = 1 then $d_{A(x)}(\text{inj}_1^{-1}(y))$
else $d_{B(x)}(\text{inj}_2^{-1}(y))$

if f(x) is A(x) × B(x) then
$d_{f(x)}(y)$ = y.1 or $d_{f(x)}(y)$ = y.2

where $d_{A(x)}$ and $d_{B(x)}$ are decomposition functions for the types A(x) and B(x).

The fundamental mechanism for dealing with general defined recursive types is recursion on Lang(f). The combinator can also take the simple form R_f:

$$\frac{\begin{array}{l} g \in A(\mathrm{VOID}) \\ h \in \Lambda x \in \mathrm{Lang}(f).(A(x) \to A(f(x))) \end{array}}{\begin{array}{l} R_f gh \in (\mathrm{ALL}x \in \mathrm{Lang}(f).A(x)) \\ (R_f gh)(\mathrm{VOID}) = g \\ (R_f gh)(f(x)) = h(x)(R_f ghx) \end{array}}$$

This combinator is used, in conjunction with the injections from Lang(f) to into $\mathbb{R}(f)$ to define the operationns and propositions on $\mathbb{R}(f)$. These operations can in turn be used to define decompositions and InBasis predicates over $\mathbb{R}(f)$. Such operations and predicates can then be used with a recursion combinator on $\mathbb{R}(f)$.

In a minimal logical theory, we would use only this R_f combinator on Lang(f) and define the other combinators from it. (We would explicitly define the inductive part of TYPE(i) and $\Lambda x \in A.B$ and define the appropriate combinators.)

3.6.3. Other operation properties

The intensional properties of operation types allows one to obtain from an arbitrary operation its domain and range. Note that the range of an operation is itself given as an operation from the domain to types.

$$\frac{\begin{array}{l} A \in \mathrm{TYPE}(i) \\ B \in A \to \mathrm{TYPE}(j) \end{array}}{\begin{array}{l} \mathbf{axiom} \in \mathrm{IsFunction}(\Lambda x \in A.B(x)) \\ \mathrm{Domain}(\Lambda x \in A.B(x)) = A \\ \mathrm{Range}(\Lambda x \in A.B(x)) = B \end{array}} \qquad \Lambda \text{ def'n}$$

It is assumed as an axiom that operations from empty types to other types can always be constructed.

$$\frac{}{\textbf{axiom} \in \Lambda x \in \mathrm{VOID}_k.\mathrm{exp}} \quad k \geq 1 \qquad\qquad \Lambda \text{ intro}$$

Since "¬T" is syntactic shorthand for "T → VOID", and "VOID → S" is always nonempty, proving "¬T" implies "T → S" for any type S, by composition of operations.

Function elimination has the obvious property:

$$\frac{\begin{array}{l} f \in \Lambda x \in A.B(x) \\ z \in A \end{array}}{f(z) \in B(z)} \qquad\qquad \Lambda \text{ elim}$$

Equality of Λ types follows the general rule for types. Two types formed using Λ are equal if and only if the domain types are equal as types and the range operations are equal as operations.

Two operations are themselves equal if and only if both are formed in the same way (they are both compositions, conditionals, projections, or recursive operations) and, recursively, the operations that these combinators act on are themselves equal from one operation to the other.

3.7. Quotient Types

To hide the details of a construction and to allow for reasoning more closely related to ordinary set theory, the quotient operator builds types from equivalence classes of objects. Since the relation defining the equivalence classes can be an arbitrary equivalence relation, one can define types that look like sets (with no intensionality properties), collections of functions with an extensional equality rather than an intensional one, etc.

$$\frac{\begin{array}{l} A \in TYPE(i) \\ R \in (\forall f \in A \rightarrow A \rightarrow TYPE(j). \\ \quad (\bigwedge(x,y,z) \in A.(f(x)(x) \\ \quad\quad \&(\ f(x)(y) \Rightarrow f(y)(x)\ ) \\ \quad\quad \&(\ (f(x)(y)\ \&\ f(y)(z)) \Rightarrow f(x)(z)\ )\ ))) \end{array}}{\begin{array}{l} (A\ /\ R) \in TYPE(max(i,j)) \\ axiom \in IsQuotient(A\ /\ R) \\ QBase(A\ /\ R) = A \\ QRelation(A\ /\ R) = R \\ q_{A\ /\ R} \in A \rightarrow (A\ /\ R) \end{array}} \quad /\ \text{def'n}$$

In order to build elements of a quotient type, one must have elements of the base type. The operation "q" assumed as a result of the above rule acts to take an element of the base type to the corrresponding equivalence class in A / R.

$$\frac{\begin{array}{l} (A\ /\ R) \in TYPE(i) \\ q \in A \rightarrow (A\ /\ R) \\ a \in A \end{array}}{q(a) \in A\ /\ R} \quad /\ \text{intro}$$

In order to reflect the fact that R is the equivalence relation whereby A / R was formed, we need a special rule for / equality. It should be noted that equality between members of quotient types is not, in general, decidable, as it depends on being able to find an element of the type to which the equivalence relation maps the elements of the base type.

$$\frac{\begin{array}{l}(A \,/\, R) \in \mathrm{TYPE}(i)\\ x \in A\\ y \in A\\ z \in R(x)(y)\end{array}}{\mathbf{axiom} \in (q(x) =_{A\,/\,R} q(y)).} \qquad /\ \text{equality}$$

Since elements of quotient types can only be built by applications of the q operation, one cannot use any information about the value to which q was applied. Thus in order to build an operation (other than a trivial operation) having as its domain a quotient type, there must be a special mechanism for inducing operations on the quotient type from operations on the base of the quotient type which respect the equivalence relation of the quotient type (R). An operation f on the base type is said to respect the equivalence relation if for all x and y in the base type where R(x)(y) holds then f(x)=f(y).

$$\frac{\begin{array}{l}(A \,/\, R) \in \mathrm{TYPE}(i)\\ f \in (\bigvee g \in (\bigwedge x \in A.B(x)).\\ \qquad\qquad (\bigwedge (x,y) \in A.\\ \qquad\qquad\quad (R(x)(y) \Rightarrow (B(x) =_{\mathrm{TYPE}(j)} B(y)\\ \qquad\qquad\qquad\qquad\qquad \& \ g(x) =_{B(x)} g(y)))))\end{array}}{\begin{array}{l}\mathrm{Qind}(f) \in (\bigwedge x \in (A \,/\, R).B(x))\\ \mathbf{axiom} \in (\bigwedge x \in A.(\mathrm{Qind}(f)(q(x)) =_{B(x)} q(f.1(x))))\end{array}} \qquad \bigwedge\ \text{intro}$$

3.8. More on Intensionality

In addition to the rules already given for the intensionality operations IsFunction, IsRecursive, IsQuotient, IsEquality and IsUnion we need an additional rule to ensure that an object cannot be both a union and a recursive type, for example. Let $IS_1, \ldots, IS_5$ be any permutation of the operations listed above. Then the following holds:

$$\frac{exp \in \mathrm{TYPE}(i),\ i \geq 1}{\mathbf{axiom} \in \mathrm{IS}_1(exp) \Rightarrow (\neg \mathrm{IS}_2(exp)\ \&\ \cdots\ \&\ \neg \mathrm{IS}_5(exp))}$$

Note that this does not mean that every type can be classified as one of these type forms. The universe is explicitely left open-ended as a means of ensuring that all constructive types can be included within the system.

Furthermore, having classified an object as an element of a function type, we need an axiom that states that it is formed in only one way. Thus, letting $\mathrm{IS}_1,\ldots,\mathrm{IS}_4$ stand for any permutation of the operations IsPerm, IsComp, IsRec, IsCond, we have

$$\frac{f \in \Lambda x \in A.B(x)}{\mathbf{axiom} \in (\mathrm{IS}_1(f) \Rightarrow (\neg \mathrm{IS}_2(f)\ \&\ \neg \mathrm{IS}_3(f)\ \&\ \neg \mathrm{IS}_4(f)))}$$

Again, note the open-ended definition of what constitutes an operation. Constructively speaking, we may not have included all methods of operation definition in the system, and so leave it open for future expansion.

4. Why PL/CV3?

4.1. Advantages of the theory.

The unified view of types, propositions and user operations presented in PL/CV3 offers several advantages. The approach to editing, verification, meta-reasoning and execution of commands is simple, and all can be explained within the theory. From earlier experience with version 2 of PL/CV, we have found that meta-reasoning is very useful. The PL/CV3 approach does not limit us to meta-reasoning - we can reason about the theory at any level of abstraction.

Experience has [illegible] that the facility of saving large amounts of already-checked logic for [illegible]se as one proves more theorems is essential, yet it must be flexible enough to allow changes to occur without the need to re-check theorems that do not depend on what was altered. The proposed implementation uses techniques derived from incremental verification [see Krafft, 1981] and program synthesis [see Teitlebaum, Reps 1979]. In order to maintain the intensional view of the universe, incremental verification becomes a necessity - once a proof of a theorem is found, it must be kept so its structure can be inspected. For incremental verification it must be kept in case any theorems or values it depends on are changed. Taking a step beyond program synthesis, we insist not only that all objects derived in the system be syntactically correct, but that they be semantically correct. A program must be proven to terminate before it can be executed, a theorem must have a proof constructed before it can be used in the derivation of other facts.

Commands are treated as internal objects, and their semantics can also be explained within the theory. The concept of a frame of reference, and a global naming convention enforced by the system provide the equivalent to a state-space semantics.

Since commands can be treated internally, users and internal processes can be treated identically. Anything a user can do can be written as a procedure; thus high-level editing procedures can be written in the same language as the language of the theory being edited.

4.2. Comparison to other languages.

LISP is also an strongly intensional language. Every (S-)expression is itself a list of objects and can be operated on as a list, or evaluated by EVAL, which itself is a list and can be expressed (very succinctly) as part of the language. Pure LISP does not permit the introduction of editing functions, although most LISP implementations use the concept of variables and pointers to implement editing functions which can then be used from within other LISP expressions as primitive objects, thus moving editing into the realm of the language. However, LISP can refer to only one universe, that of lists and atoms, and does not allow for any meta-reasoning.

LCF is closer than LISP to PL/CV3, in that it consciously implements a logic for programming. However its universe allows for only one level of meta-reasoning, so there is a real separation between meta-language types and "real" types. The universe that the user sees cannot be folded down to be a part of the theory. LCF is not intensional at the meta-language level. although it does show some intensional properties in its treatment of "forms".

5. Examples

In this section, we illustrate the theory by presenting familiar examples in it, starting wih the predicate calculus, then number theory, analysis and set theory.

5.1. Predicate calculus examples

Suppose we want to prove that

SOME $y \in A$.ALL $x \in A$.p(x,y) $\Rightarrow$ ALL $x \in A$. SOME $y \in A$.p(x,y)

This involves building a function.

1. In this case the function delimiters are PROOF and QED for mnemonic reasons.

```
PROOF;
   ASSUME SOME y ∈ A.ALL x ∈ A.p(x,y);
   /* the goal is to build a function, an element of the
      type  ALL x ∈ A. SOME y ∈ A. p(x,y)    */
   ALL x ∈ A.SOME y ∈ A. p(x,y) BY INTRO,
   PROOF;
      ARB x ∈ A;
      CHOOSE y ∈ A WHERE L1: ALL x ∈ A.p(x,y);
      L2: p(x,y) BY ALLEL, L1(x);
      SOME Y ∈ A.p(x,y) BY INTRO, y, L2;
   QED;
QED;
```

2. An equivalent way to write this is

```
(ARB z ∈ Vy ∈ A.Λx ∈ A.p(x,y);
     (ARB x ∈ A; pair(z.1,z.2(x,z.1))))
∈ Vy ∈ A.Λx ∈ A.p(x,y) ⇒ Λx ∈ A.Vy ∈ A.p(x,y)
```

These are simply two different ways of spelling the same mathematical object. The typings and definitions used in the first form are matters of linguistic style.

3. It is also possible to write this proof as a procedure which manipulates a "state." For example

```
⋁y ∈ A.⋀x ∈ A.p(x,y) ⇒ ⋀x ∈ A.⋁y ∈ A.p(x,y) BY
PROC;
    ARB Z ∈ (⋁y ∈ A.⋀x ∈ A.p(x,y));
    DCL Y ∈ A;
    DCL w ∈ ⋀x ∈ A.⋁y ∈ A.p(x,y);
    DCL w2 ∈ ⋀x ∈ A.p(x,Z.1);
    Y := Z.1;
    w2 := Z.2;
    w := (ARB x ∈ A; w2(x));
  END
```

This form of "procedure" is such that the result extracted depends on which components are selected of the implicit parameterized dependent product of the form

```
PROD(y:A, w: ⋀x ∈ A.⋁y ∈ A.p(x,y), w2: ⋀x ∈ A.p(x,Z.1))
```

4. We have expressed the previous propositions as typings because we were interested in the proof. We can also write a predicate which defines the proved (small) propositions.

```
FOR x ∈ TYPE(1) DEFINE Proved(x) = SOME p ∈ x.true
DEFINE TRUE(1) = {x ∈ TYPE(1) : Proved(x)}.
```

Now whenever we have the typing "A BY proof", we also have the proposition Proved(A). The system will automatically make this assertion from the typing. We can make this concept more familiar by using the notation $\vdash$(A) for Proved(A).

5.2. An example from arithmetic

The theory of arithmetic from PL/CV2, described in [Constable, O'Donnell 70], is used here also. We show how to define the least prime factor of a positive integer. assuming certain basic facts about arithmetic which are proved in detail in [Constable. Johnson. Eichenlaub 81]. called the source document in the proof.

1. Assume there is a definition of "x divides y" as "SOME z ∈ N.(y=n*x)". and a definition of prime(x) as "x > 1 & ALL z ∈ INT WHERE 1 < z < x . ¬(z divides x)". Now we prove

```
Theorem:  ALL x ∈ N WHERE x>1 . ALL L ∈ N .
                 (  L divides x & prime(L)
                  & ALL d ∈ INT.(d divides x & d > 1 ⇒ L ≤ d))
BY INTRO.

    Least: PROC;

    /* define a function which finds the least factor */
    divisor:  PROC
         ARB Y ∈ INT;
         ATTAIN  divisor(y) divides x. divisor(y) > 1.
             ALL i ∈ INT WHERE 2 ≤ i < divisor(y) . ¬(i divides x);
         DCL (I.J) ∈ INT;
         ALL K ∈ INT WHERE 2 ≤ K<I . ¬(K divides y) BY INTRO;
         DO J = 2 to Y;
            ASSUME D:  ALL K ∈ INT WHERE 2 ≤ K<I . ¬(K divides y);
            IF MOD(Y.J) = 0
            THEN RETURN(J);
            MOD(Y.J) ≠ 0;
            E:  ¬ (J divides Y);
            ALL K ∈ INT WHERE 2  ≤  K < I+1 . ¬ (K divides Y)
                BY ALLEL. domain_ext.D.2.E;
         END;
         RETURN(Y);
    END divisor;
L1: divisor(x) divides x BY FUNCTION. divisor(x);
L2: ALL k ∈ INT WHERE 2 ≤ k < division(y) .
        ¬(k divides x) BY FUNCTION. divisor(x);
    ALL z ∈ INT WHERE 1 < z < divisor(x) .
        ¬(z divides divisor(x))  BY INTRO.
    ¬(z divides divisor (x))  BY CASES. D.
        PROOF;
           CASE z divides divisor(x);
           CASE z divides x
                      BY (lemma in divisors. see complete source);
                      ¬(z divides x)  BY ALLEL.L2.z;
        QED;
    QED;
    divisor(x) > 1  BY FUNCTION. divisor(x);
L3: prime(divisor(x));
```

```
L4: ALL d ∈ INT . (d divides x & d > 1 ⇒ divisor(x) ≤ d) BY INTRO,
    PROOF;
       L:  d < divisor(x) ∨ divisor(x) ≤ d BY ARITH;
           divisor(x) ≤ d BY CASES,L;
           PROOF;
              CASE  d < divisor(x);
                 1 < d < divisor(x);
                 d divides x;
                 ¬(d divides x)  BY ALLEL,L2,0;
           QED;
    QED;

/* take as L of the conclusion, divisor(x);
   then L1, L3 and L4 establish the theorem */
END least
```

2. A critical lemma in this proof is domainext used after statement E in the DO loop. Here is a proof of that lemma which illustrates how the structural form of propositions can be used in statements.

```
domainext:  ALL L ∈ INT . ALL P ∈ (INT ⇒ TYPE(1·))
            WHERE (P=(ARB n; ALL x ∈ INT WHERE L ≤ x<n . Q(x)))
            . (ALL n ∈ INT . (P(n) & Q(n) => P(n+1))  BY INTRO, INTRO,
            PROOF;
               ALL n ∈ INT . (P(n) & Q(n) => P(n+1))  BY INTRO,
               PROOF;
                  ALL x ∈ INT WHERE L ≤ x [illegible] BY INTRO,
                  PROOF;
                     x < n ∨ x = n   BY ARITH , x < n+1;
                     Q(x)  BY CASES,
                     PROOF;
                        CASE x < n;
                              Q(x)  BY ALLEL, P(n),x;
                        CASE x=n; Q(x);
                     QED;
                  QED;
               QED;
            QED;
```

3. Here is another way to express the work of finding a divisor; we can iterate the function e defined below, starting with k=2.

```
ALL n ∈ INT .  ALL k ∈ N.
    (ALL i ∈ {i ∈ INT : 2 ≤ i< k } . ¬(i divides n) ⇒
```

```
      ALL i ∈ {i ∈ INT : 2 ≤ i < k+1} . (¬(i divides n) v (k divides n))

  BY INTRO.
     e: PROC  ARB n ∈ INT;
        f: PROC ARB k ∈ INT;
             PROC
                ASSUME  L: ALL i ∈ {i ∈ INT : 2 ≤ i<k} . ¬(i divides n);
                IF decide (k divides n)
                THEN proof (k divides n)
                ELSE PROC ARB j ∈ {k ∈ INT : k ≤ j < k} . ¬(j divides n)
                        D:  k=j v j < k    BY ARITH;
                        ¬ (j divides n)  BY CASES, D.
                        PROOF;
                           CASE k=j; ¬ (k divides n)
                           CASE j<k; ¬ (j divides n)  BY ALLEL.L.j;
                        QED;
                     END;
                FI;
             END;
          END f;
     END e;
```

We assume that the relation "k divides n" is proved automatically by a routine called decide, which the user writes. The function "proof" in the THEN clause supplies the proof for automatically generated proofs.

5.3. A second order arithmetic example

1. We can easily prove the corollary to Cantor's theorem, that N→N is not enumerable.

```
  ¬SOME E ∈ (N→(N→N)) . ALL f ∈ (N→N) . SOME n ∈ INT . E(n)=f
  BY INTRO, PROOF;
     /* suppose such an E exists and derive a contradiction */
     CHOOSE E ∈ (N→(N→N)) WHERE ALL f ∈ (N→N)
                                        . SOME n ∈ INT . E(n)=f
     SOME d ∈ INT . (E(d) = D:(ARB x ∈ INT; E(x)(x)+1))
     BY ALLEL, ALL f ∈ (N→N) . SOME n ∈ N . (E(n)=f), D;
     /* D is the diagonal function */
     CHOOSE d IN N WHERE E(d) = (ARB x ∈ N; E(x)(x)+1);
  L1: D(d) = E(d)(d)+1;
  L2: E(d)(d) = D(d);
       false BY ARITH,L1,L2;
  QED;
```

2. Cantor's theorem can easily be proved for any type on which we can find a different element for each element.

```
FOR A ∈ TYPE DEFINE
    SEP(A) = ALL x ∈ A . SOME y ∈ A . (¬(x=y)).
```

Note, $=_A$ is easily inferred for = by the type checker in this context.

```
DEFINE S = {x ∈ TYPE : SEP(x)}

ALL A ∈ S . ¬(SOME E ∈ A → (A→A) .
        ALL f ∈ (A → A) . SOME x ∈ A. (E(x)=f)).
BY INTRO,INTRO,
PROOF;
     CHOOSE E ∈ A → (A → A) WHERE L: ALL f ∈ A → A . SOME x ∈ A . (E(x)=f);
     CHOOSE sep ∈ ALL x ∈ A . SOME y ∈ A . (¬(x=y));
     /* apply phrase L of the hypothesis
        to the diagonal function, named D*/
     DEFINE D = (ARB y ∈ A; sep(E(x)(y)));
     SOME x ∈ A . (D = E(x))  BY ALLEL,L,D;
     CHOOSE d ∈ A WHERE E(d)=D;

     D(d) = sep(E(d)(d)); /*by definition of D*/
     E(d)(d) = D(d); /*by definition of E and d*/

     false; /*since ¬(sep(E(d)(d)) = E(d)(d))*/

QED;
```

5.4. Set theory

1. Sets are built as "equivalence classes" of types. The technique is illustrated by examining the small subsets built on a type A. First we define the "small subtypes", P(A).

```
FOR A ∈ TYPE(1)
DEFINE P(A) = {x ∈ TYPE(1) : SOME Q ∈ A → TYPE(1) . (x = {y ∈ A : Q(y)})}
```

So the small subtypes of A have the form {x ∈ A : Q(x)} for some proposition Q. We now define an extensional equivalence relation on these types to make them sets.

```
FOR (S1.S2) ∈ P(A)
DEFINE INFIX eqe = ALL x ∈ A . ((S1.2)(x) ⇔ (S2.2)(x))
```

Recall that S1.2 and S2.2 select the second (propositional) part of the type {x ∈ A : Q(x)}.

```
DEFINE Pow(A) = P(A) / eqe
```

Officially, before we can define this, we must show that eqe is an equivalence relation, a step we omit here. Pow is the type of (small) subsets of A. On these, we can define the usual set operations: union, intersection, and difference as follows.

2. The intersection, union and complement of subsets X and Y are defined as

```
FOR (X.Y) ∈ P(A)
DEFINE INFIX cap = {z ∈ A : (X.2)(z) & (Y.2)(z)},
             cup = {z ∈ A : (X.2)(z) v (Y.2)(z)};
FOR X ∈ P(A)
DEFINE bar = {z ∈ A : ¬(X.2)(z)};
```

Now we show that "cap" respects the "eqe" equivalence and hence induces a cap operation on Pow(A).

```
Prop: ALL (x1,x2,y1,y2) ∈ P(A) .
          ((x1 eqe x2 & y1 eqe y2) ⇒ ((x1 cap y1) eqe (x2 cap y2))
      BY INTRO, INTRO, PROOF;
          ALL z ∈ A . ((x1 cap y1).2)(z) ⇔ ((x2 cap y2).2)(z))
          BY INTRO, PROOF;
             ((x1 cap y1).2)(z) ⇒ ((x2 cap y2).2)(z) BY INTRO;
             /* This step is a sequence of equivalences which can be
                automatically deduced from the hypotheses
                x1 eqe x2 and y1 eqe y2 */
             ((x2 cap y2).2)(z) ⇒ ((x1 cap y1).2)(z);
          QED;
      QED
```

Now it is possible to lift cap to the Pow type. The official name for it is q(cap), but the system will allow the use of cap since the type will always be unambiguous.

To prove X cap Y = Y cap X for all X, Y in Pow(A), we return to P(A), where it is necessary to show

(X cap Y) eqe (Y cap X)   for X, Y in P(A)

This is trivial by the commutativity of "&". In fact, arguments of this type are so typical, that we want to build general methods of carrying them out automatically.

3. The general fact behind the commutativity of cap is this; If an operation "op" on subtypes is defined by a commutative operation on the proposition, then "op" is commutative. This can be stated as a theorem as follows.

```
FOR F ∈ TYPE → (TYPE → TYPE), op ∈ P(A) → (P(A) → P(A))
DEFINE Def = ALL (X,Y) ∈ P(A) .
                    (  op(X, Y) = {x ∈ A : F((X.2)(x), (Y.2)(x))}
                     & op(Y, X) = {x ∈ A : F((Y.2)(x), (X.2)(x))}
                    );
ALL F ∈ TYPE → (TYPE → TYPE)
WHERE (ALL (X,Y) ∈ TYPE . (F(X, Y) ⇔ F(Y, X))) .
     ALL op ∈ P(A) → (P(A) → P(A)) WHERE Def(F, op) .
         ALL (S1,S2) ∈ Pow(A) . (op(S1, S2) = op(S2, S1))
BY INTRO, INTRO, PROOF;
    ALL (S1,S2) ∈ Pow(A) . (op(S1. S2) = op(S2, S1))
    BY INTRO, PROOF;
        /* since Pow(A) is a quotient type, we can choose
           arbitrary representatives */
        ARB (X,Y) ∈ P(A);
        op(X, Y) = {x ∈ A : F((X.2)(x), (Y.2)(x))}
                 BY ALLEL, Def(F, op), X, Y;
        op(Y, X) = {x ∈ A : F((Y.2)(x), (X.2)(x))}
                 BY ALLEL, Def(F, op), X, Y;

        /* goal: show (X op Y) eqe (Y op X) */
        ALL x ∈ A . (F((X.2)(x), (Y.2)(x)) ⇔ F((Y.2)(x), (X.2)(x))
        BY INTRO, PROOF;
            L1; ALL (X,Y) ∈ TYPE . (F(X, Y) ⇔ F(Y, X));
            F((X.2)(x), (Y.2)(x)) ⇔ F((Y.2)(x), (X.2)(x))
                      BY ALLEL, L1, X.1(x), X.2(x);
```

```
        QED;
        {x ∈ A : F((X.2)(x), (Y.2)(x))} eqe
        {x ∈ A : F((Y.2)(x), (X.2)(x))} /* from the definition of eqe */
        op(X, Y) = op(Y, X); /* by the definition of op */
    QED;
QED
```

5.5. Recursive types

Let us consider the beginnings of the theory of formal languages. Starting with a finite alphabet $A = \{a_1,\ldots,a_n\}$, the type A^* of all finite sequences of elements of A, including the empty sequence, is defined. Here is how that is done in this theory.

First we define the family of finite discrete types as follows. Let Nat = $\{x \in INT : x \geq 0\}$. We also assume the notation [1,n] stands for the type SOME $x \in INT$. $1 \leq x \leq n$.

```
/* a type is finite iff there is a bijection onto an interval 1,....,n */
FOR X ∈ TYPE
DEFINE Finite(X) = SOME n ∈ Nat .
                   SOME e ∈ [1,n] → X .
                   SOME p ∈ X → [1,n] .
                        ( ALL i ∈ [1,n] . (p(e(i)) = i)
                        & ALL x ∈ X . (e(p(x)) = x));
/* a discrete type is one with a decidable equality */
FOR X ∈ TYPE
DEFINE Discrete(X) = ALL (x,y) ∈ X . (x = y v ¬(x = y));
```

Now define the type A^+ of all nonempty finite sequences from A.

```
FOR (A,X) ∈ TYPE
DEFINE seq(A, X) = A + X × A;

FOR A ∈ {X ∈ TYPE : Finite(X) & Discrete(X)}
DEFINE Seq(A) = R((ARB Y ∈ TYPE; seq(A, Y)));

/* this is equivalent to the domain equation
   Seq(A) = A + Seq(A) × A               */
```

The language of Seq(A), denoted Lang(Seq(A)), is essentially the type consisting of A, A + A × A, A + (A + A × A) × A, etc. The type Seq(A) is isomorphic to the union of all of these types, i.e. x ∈ Seq(A) iff x is the image of an element of one of these types. Given x ∈ Seq(A), the function Basis(x) maps x to the elements of Lang(Seq(A)) from which it arose.

We want sequences that include the empty sequence, which is simply some special element of an enumeration type.

```
FOR A ∈ {X ∈ TYPE : Finite(X) & Discrete(X)}
DEFINE SeqStar(A) = Seq(A) + {nil};
```

We can now define functions on SeqStar(A). We want to define

```
Nil(x)  iff x = nil
Atom(x) iff x ∈ A
hd(xa) = a, hd(a) = a
tl(xa) = x, tl(a) = a for a ∈ A
```

The definitions will be

```
Nil(x) = (Case(x) = 2)
Atom(x) = (Case(x) = 1 & "x comes from A + VOID × A")
hd(x) = IF Basis(x) = A
        THEN x
        ELSE x.2
        FI
```

Officially, we cannot write "x.2" but must instead map x back into a type which is a product. So we take "inj(x).2" where inj is an appropriate injection from Seq(A) back into one of the elements of Lang(A).

```
tl(x) = IF Basis(x) = A
        THEN x
        ELSE x.1
        FI
```

Again, officially we cannot use "x.1" but must use injections. The actual form of these definitions is:

```
FOR x ∈ SeqStar(A)
DEFINE Nil(x) = (Case(x) = 2),
       Atom(x) = (Basis(x.1) = seq(A, VOID));

FOR x ∈ SeqStar(A)
DEFINE hd(x) = IF Nil(x)
               THEN x
               ELSE IF Atom(x)
                    THEN x
                    ELSE in(out(x).2)
                    FI
               FI,
       tl(x) = IF Nil(x)
               THEN x
               ELSE IF Atom(x)
                    THEN x
                    ELSE in(out(x).1)
                    FI
               FI;
```

for appropriate definitions of "in" and "out", e. g.

$$out(x) = inj^{-1}_{BASIS(x) \rightarrow Seq(A)}(x).$$

Finally we can define a recursive concatenation operation as

```
FOR (x,y) ∈ SeqStar(A)
DEFINE x con y = IF atom(y)
                 THEN in(pair(out(x), out(y)))
                 ELSE in(pair(out(x con tl(y)). out(hd(y))))
                 FI
```

again, for appropriate definitions of "in" and "out".

If we want to treat SeqStar(A) as an abstract type, we can form the dependent product

AbsrecStar(A) = Prod(SeqStar(A), Nil, Atom, hd, tl, con)

REFERENCES

Aczel, P., The Type Theoretic Interpretation of Constructive Set Theory, Logic Colloquium, North-Holland, Amsterdam, 1978, pp. 55-66.

Aczel, P., Frege Structures and the Notions of Proposition, Truth and Set, The Kleene Symposium, North-Holland, Amsterdam, 1980, pp. 31-60.

Allen, John, Anatomy of Lisp, McGraw-Hill, New York, 1978.

Banachowski, L., A. Kreczmar, G. Mirkowska, H. Rasiowa, and A. Salwicki, An Introduction to Algorithmic Logic, Banach Center Publications, Vol. 2, Warsaw, 1977.

Backus, J. W., Can programming be liberated from the von Neuman style? A functional style and its algebra of programs, CACM, August 1978.

Bates, J.L., A Logic for Correct Program Development, Ph.D. Thesis, Department of Computer Science, Cornell University, 1979.

Bishop, E., Foundations of Constructive Analysis, McGraw Hill, New York, 1967, 370 pp.

Bishop, E., Mathematics as a Numerical Language, Intuitionism and Proof Theory, J. Myhill, ed., North-Holland, Amsterdam, 1970, pp. 53-71.

Bishop, E., H. Cheng, Constructive Measure Theory, American Math. Society Memoir, 116, 1972.

Boehm, H., A.J. Demers, J.D. Donahue, An Informal Introduction to Russell, Department of Computer Science TR80-430, Cornell University, 1980.

Bourbaki, N., Elements of Mathematics, Vol I, Theory of Sets, Addison-Wesley, Reading, 1968.

Boyer, R.S. and J.S. Moore, A Computational Logic, Academic Press, New York, 1979, 397 pp.

Bridges, D.S., Constructive Functional Analysis, Pitman, London, 1979.

Burstall, R. and J. Darlington, Some Transformations for Developing Recursive Programs, Proc. of 1975 International Conference on Reliable Software, 1975, pp. 465-472.

Cartwright, R. and J. McCarthy, First Order Programming Logic, Proc. Sixth Annual Symp. on Principles of Programming Languages, ACM, Jan. 1979, pp. 68-80.

Cartwright, R., A Constructive Alternative to Axiomatic Data Type Definitions, Computer Science Department TR 80-427, Cornell University, 1980.

Cohn, A.J., Machine Assisted Proofs of Recursion Implementation. Doctoral Thesis, University of Edinburgh, April 1980.

Constable, R.L., Constructive Mathematics and Automatic Program Writers, Proc. of IFIP Congress, Ljubljana, 1971, pp. 229-233.

Constable, R.L., A Constructive Theory of Recursive Functions, Computer Science Department TR 73-186, Cornell University, 1973.

Constable, R.L., On the Theory of Programming Logics, Proceedings of the 9th Annual ACM Symposium on Theory of Computing, Boulder, May 1977, pp. 269-285.

Constable, R.L. and M.J. O'Donnell, A Programming Logic, Winthrop, Cambridge, 1978.

Constable, R.L. and S.D. Johnson, A PL/CV Precis, Conf. Record of the ACM Symposium on Principles of Programming Languages, ACM, New York, 1979, pp. 7-20.

Constable, R.L., FORMAL: a family of languages for communicating formal reasoning, Computer Science Department, Cornell University, 1980.

Constable, R.L. and D. Zlatin, Report on the PL/CV3 Verification System, Computer Science Department, Cornell University, 1980.

Constable, R.L., S.D. Johnson, and C.D. Eichenlaub, Introduction to the PL/CV2 Programming Logic, Department of Computer Science, Cornell University, January 1981.

Conway, R. and D. Gries, An Introduction to Programming, A Structured Approach Using PL/1 and PL/C-7, Second Edition, Winthrop, Cambridge, Massachusetts, 1975, 509 pp.

Curry, H.B., R. Feys, and W. Craig, Combinatory Logic, Vol. 1, North-Holland, Amsterdam, 1968.

Davis, M. and J.T. Schwartz, Mathematical Extensibility for Theorem Verifiers and Proof Checkers, Comp. and Math. w. applications, 5, 1979, pp. 217-230.

de Bruijn, J.G., The mathematical language AUTOMATH, its usage and, some of its extensions, Symposium on Automatic Demonstration Lecture Notes in Math, Vol. 125, Springer-Verlag, New York, 1970, pp. 29-61.

de Bruijn, N.G., A Survey of the Project AUTOMATH Essays on Combinatory Logic, Lambda Calculus and Formalism, Academic Press, 1980, pp. 579-606.

Demers, A.J. and J. Donahue, Parameters and Type Checking, Proc. of Seventh ACM Symposium on Principles of Programming Languages, January, 1980.

Demers, A.J. and J. Donahue, Revised Report on Russell, Department of Computer Science TR 79-389, Cornell University, Septemer, 1979.

Derschowitz, N. and Z. Manna, Proving Termination with Multiset Orderings, Comm. ACM, Vol. 22, No. 8, pp. 465-476.

Downey, P., R. Tarjan, and R. Sethi, Variations on the Common Subexpression Problem, JACM Vol. 27, No. 4, Oct. 1980, pp.758-771.

Dummett, M., Elements of Intuitionism, Oxford Logic Series, Clarendon Press, Oxford, 1977.

Engeler, E., Formal Languages: Automata and Structures, Markham, Chicago, 1968.

Feferman, S., A language and axioms for explicit mathematics, Algebra and Logic, Lecture Notes in Mathematics, 450, Springer-Verlag, New York, 1975, pp. 87-139.

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

Friedman, H., Set theoretic foundations for constructive analysis, Annals of Math, 105, 1977, p. 1-28.

Gerhart, S., Corectness - Preserving Program Transformations, Symposium on Principles of Programming Languages, ACM, New York, 1975, pp. 54-66.

Goad, C., Proofs as Descriptions of Computations, Proc. of the 5th Conference on Automated Deduction, Les Arcs, France, Lecture Notes in Computer Science, Vol. 87, pp. 39-52.

Goad, C., Computational uses of the manipulation of formal proofs, Ph.D. Thesis, Stanford University, 1980.

Goguen, J.A., J.W. Thatcher, and E.G. Wagner, Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, Current Trends in Programming Methodology, Vol. IV, (ed. R. Yeh), Prentice-Hall, Englewood Cliffs, 1978.

Goldblatt, R., Topoi: The Categorical Analysis of Logic, North-Holland, Amsterdam, 1979.

Gordon, M., R. Milner, and C. Wadsworth, Edinburgh LCF, A Mechanized Logic of Computation, Lecture Notes in Computer Science, Springer-Verlag, 1979.

Guttag, J.V., E. Horowitz, and D.R. Musser, Abstract Data Types and Software Validation, Comm. ACM, 21, 12, 1978, pp. 1048-1064.

Harel, D., First-Order Dynamic Logic, Lecture Notes in Computer Science, 68, Springer-Verlag, New York, 1979.

Hoare, C.A.R., Recursive Data Structures, International Journal of Computer and Information Sciences, Vol. 4, 2, June, 1975, pp. 105-132.

Ichbiah, J.D. et al, Preliminary ADA Reference Manual, SIGPLAN Notices, 14, 6, June 1979, Part A, ACM, New York.

Ichbiah, J.D. et al, Rationale for the Designe of the ADA Programming Language, SIGPLAN Notices, 14, 6, June, 1979, Part B, ACM, New York.

Igarashi, S., R.L. London, and D.C. Luckham, Automatic Program Verification I: A Logical Basis and its Implementation, Acta

Informatica, 4, 1975, pp. 145-182.

Jensen, K. and N. Wirth, *Pascal User Manual and Report*, (2nd ed.), Springer-Verlag, New York, 1975.

Johnson, S.D., *A Computer System for Checking Proofs*, Ph.D. Thesis, Cornell University, January 1981.

Jutting, L.S., Checking Landau's "Grundlogen" in the AUTOMATH System (Doctoral Thesis, Eindhoven University). *Math. Centre Tracts No. 83*, Math. Centre, Amsterdam, 1979.

Kennedy, K., and J.T. Schwartz, An Introduction to the Set Theoretical Language SETL, *Comp. and Math. with Applications*, Vol. 1, Permagon Press, 1975, pp. 97-119.

Kleene, S.C., *Introduction to Metamathematics*, D. Van Nostrand, Princeton, 1952, 550 pp.

Kozen, D.C., *Complexity of Finitely Presented Algebras*, Ph.D. Thesis, Cornell University, Computer Science Department, 1977.

Krafft, D.B., *An Interactive Proof Development System*, Ph.D. Thesis, Cornell University, Computer Science Department, August 1981.

Lewis, C.H. and B.K. Rosen, Recursively defined Data Types, Part 1, *ACM Symposium on Principles of Programming Languages*, Boston, October, 1973, pp. 125-138.

Liskov, B.H. and S.N. Zilles, Specification Techniques for Data Abstractions, *IEEE Trans. on Software Engineering*, Vol .SE-1, 1, 1975, pp. 7-19.

Manna, Zohar, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.

Manna, Z. and R. Waldinger, Is "sometime" sometimes better than always? Intermittent assertions in proving program correctness, CACM, Vol. 21, No. 2, February, 1978, pp. 159-172.

Martin-Lōf, P., Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions, *Proceedings of the Second Scandinavian Logic Symposium*, ed. J.E. Fensted, North-Holland, Amsterdam, 1971, pp. 179-216.

Martin-Lōf, P., An Intuitionistic Theory of Types: Predicative Part, *Logic Colloquium* '73, ed. H.E. Rose, J.C. Shepherdson, North-Holland, Amsterdam, 1975, pp. 73-118.

Martin-Lōf, P., Constructive Mathematics and Computer Programming, *6th International Congress for Logic, Method. and Phil. of Science*, Hannover, August, 1979.

McCarthy, J., A Basis for a Mathematical Theory of Computation, *Computer Programming and Formal Systems*, eds. P. Braffort and D. Hirschberg, North-Holland, Amsterdam, 1963, pp. 33-70.

McGettrick, A.D., *Algol 68, a first and second course*, Cambridge University Press, Cambridge, 1978.

Milne, R. and C. Strachey, A Theory of Programming Language Semantics, Halsted Press, New York, 1976.

Milner, R., Implementation and Applications of Scott's Logic for Computable Functions, Proc. ACM Conf. on Proving Assertions about Programs, Las Cruces, New Mexico, 1972, pp. 1-6.

Mirkowska, G., On Formalized Systems of Algorithmic Logic, Bull. Acad. Pol. Sci., Ser. Math. Astrong. Phys., 18 (1971) pp. 421-428.

Myhill, J., Constructive Set Theory, JSL, 40, 1975, pp. 347-383.

Oppen, D.C., Reasoning about recursively defined data structures, Conf. Record of 5th Annual ACM Symposium on Principles of Programming Languages, ACM, New York, January, 1978, pp. 151-157.

Pratt, V.R., Semantical Considerations on Floyd-Hoare Logic, Proc. of 17th Annual Symp. on Foundations of Computer Science, Houston, October, 1976, pp. 109-121.

Pratt, V.R., On Specifying Verifiers, August, 1979.

Rasiowa, H., Algorithmic Logic, ICS PAS Reports 281, Institute of Computer Science, Polish Academy of Science, Warsaw, Poland, 1977.

Salwicki, A., Formalized Algorithmic Languages, Bul. Acad. Pol. Sci., Ser. Sci., Math. Astr. Phys., vol. 18, No. 5, 1970.

Schwartz, J., Correct Program Technology, Courant Computer Science Report #12, New York University, 1977.

Scott, D., Outline of a Mathematical Theory of Computation, Proc. 4th Annual Princeton Conference on Information Sciences and Systems, Princeton, 1970, pp. 169-176.

Scott, D., Data Types as Lattices, SIAM Journal on Computing, 5, 3, September, 1976.

Scott, D., Constructive Validity, Symposium on Automatic Demonstration, Lecture Notes in Mathematics, 125, Springer-Verlag, 1970, pp. 237-275.

Scott, D., and C. Strachey, Toward a Mathematical Semantics for Computer Languages, Proc. of the Symposium on Computers and Automata, Polytech Institute of Brooklyn, New York, 1971. pp. 19-46.

Shamir, A., and W.W. Wadge, Data Types as Objects, Automata, Languages, and Programming, 4th Colloquium, Lecture Notes in Computer Science No. 52, Turku, 1977, pp. 465-479.

Solomon, M.H., Theoretical Issues in the Implementation of Programming Languages, Ph.D. Thesis, Cornell University, May, 1977.

Stoy, J.E., Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, the MIT Press, Cambridge, 1977, 414 pp.

Stenlund, S., Combinators, Lambda-terms, and Proof-Theory, D. Reidel, Dordrecht, 1972, 183 pp.

Teitelbaum, R. and T. Reps, The Cornell Program Synthesizer: A Syntax-directed Programming Environment (to appear CACM, 1981).

Troelstra, A.S., Metamathematical Investigations of Intuitionistic Arithmetic and Analysis, Lecture Notes in Mathematics, 344, Springer-Verlag, New York, 1973.

van Daalen, Diederik T., The Language Theory of Automath, Doctoral Dissertation, Tech. Univ. of Eindhoven, 1980.

van Wijngaarden, A.B.J. et al, Revised Report on the Algorithmic Language ALGO 68, Supplement to ALGO BULLETIN, University of Alberta, 1974.

Weyhrauch, R., Prolegeoma to a Theory of Formal Reasoning, AIM-315, Stanford Artifical Intelligence Laboratory, December, 1978.

Williams, John H., On the Development of the Algebra of Functional Programs, Computer Science Research Report, IBM San Jose, 1980..

APPENDIX A

Another Construction for Recursive Types

While the method of introducing recursive types presented in section 3.5 is adequate to define the recursive types of modern programming languages, it is not adequate for certain kinds of transfinite induction used in constructive set theory. In particular [Aczel 78] and [Martin-Löf 79] introduce a concept of constructive well-ordering which goes far beyond the well-orderings we could define. Moreover, their treatment of this concept is quite simple, simpler than our notion of a recursive type. We are therefore led to introduce this more powerful concept and use it to define the recursive types of section 3.5 as outlined below.

The **W** constructor is similar in syntax to the $\bigvee$ and $\bigwedge$ constructions. A **W** type can be introduced using the following rule.

$$\frac{\begin{array}{l} T \in \mathrm{TYPE}(i) \\ S \in T \rightarrow \mathrm{TYPE}(j) \end{array}}{\mathbf{W}x \in T.S(x) \in \mathrm{TYPE}(\max(i,j))} \qquad \mathbf{W}\ \text{intro}$$

$\mathrm{Nodes}(\mathbf{W}x \in T.S(x)) = T$

$\mathrm{OutDegree}(\mathbf{W}x \in T.S(x)) = S$

Elements of a well-ordering can be thought of as a tree formed such that each branch has finite length, but the number of branches to descendants in the tree from any node may be infinite. The nodes of one of these trees have values associated with them, taken from the type T. The fan-out of a node labelled with a value t in T can be put in one to one correspondence with the elements of S(t).

```
\|/ \|/  \|/ }  ← subtree f(b)
 o   o    o  }
  \  |.../ ← branches corresponding to elements of S(t);
   \ |  /     say b ∈ S(t)
    \| /
      o  ← node labelled with t ∈ T
```

Diagram of sup(t, f) $\in$ W$x \in T.S(x)$

Creating elements of W types closely corresponds to definition by induction. Given an element t of a type T, and a function from S(t) to the W type in question, the function **sup** creates an element of the W type.

$$\frac{\begin{array}{l} t \in T \\ f \in S(t) \rightarrow \mathrm{W}x \in T.S(x) \end{array}}{\mathrm{sup}(t,f) \in \mathrm{W}x \in T.S(x)} \qquad \text{W intro}$$

$$\mathrm{Label}(\mathrm{sup}(t,f)) = t$$

$$\mathrm{Predecessor}(\mathrm{sup}(t,f) = f$$

That is, an element of a well-ordering type is represented by the combination of an element t of T and a function f. When applied to an element b of S(t), f gives the element of W$x \in T.S(x)$ found by following the b^{th} edge out of the node labelled with t. This function is, in essence, a generalized predecessor function, as an element of a well-ordering can have as many predecessors as there are elements in the type specified by applying S to the label on the root node.

Given an element **sup**(t,f), we can operate on it to obtain the t and f objects by using the Label and Predecessor operations. The other form of elimination of W types corresponds to the definition of recursive functions on the W type. The Brec operation takes as argument a function f and produces as its result a function taking y in

$\mathbf{W}x \in T.B(x)$ to $C(y)$. The function f takes as argument and element y of $\mathbf{W}x \in T.B(x)$ and returns a function which given a predecessor z of y, returns an element of $C(z)$.

$$\frac{f \in \Lambda y \in (\mathbf{W}x \in T.B(x)).(\Lambda x \in \mathrm{Predecessor}(y)(\mathrm{Label}(y)).C(x))}{\mathrm{rec}(f) \in \Lambda x \in (\mathbf{W}x \in T.B(x)).C(x)} \quad \mathbf{W}\ \mathrm{elim}$$

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. For the predecessor function, equal means extensionally equal, rather than intensionally equal.

Our notion of recursion in section 3.5 was defined so that we would be able to define the integers from the primitive concepts of the theory as a recursive type. Clearly, we want the integers to be a TYPE(1) object. This meant that, after defining the language type of a recursive definition to be one level higher than the recursive type, we could not just use $\bigvee T \in L(T).T$ as our definition of the recursive type, as this would also be one level higher than desired. So we axiomatized the recursive types to be reflected down one level, and in doing so introduced the requirement that recursive types not be allowed to use intensionality functions in the function on which the recursion is performed. Otherwise, let f be given by

```
f(T) = IF IsRecursive(T)
         THEN RecFunc(T)(T)
         ELSE g(T)
```

This f is a mapping from types to types; hence we can build REC(f) as a recursive type. But then the application of f to the type REC(f) is a non-terminating computation.

Using the $\mathbf{W}$ combinator, we can build the integers from the assumption of the existence of an empty type, a type with one element, and a type with two elements. Furthermore, constructing the integers this way leaves them in TYPE(1) without complicated assumptions on what functions can be used in recursive definitions.

Let the type with one element be $\mathbf{1}$, and let its element be represented by "1". Thinking again of elements of a $\mathbf{W}$ type as trees, the integer 0 will be represented as a tree of exactly one node labelled with false; the successor of an integer n will be represented as a tree with a root labelled by true, and a single outward edge to the tree which represents n. So labels come from the type Bool, and there are either no edges out of a node if the label is false, or one edge if the label is **true**. So the number of outward edges corresponds to elements of VOID and $\mathbf{1}$, respectively. In order to build such a $\mathbf{W}$ type, we need a function f of type Bool $\rightarrow$ V such that f(false) = VOID, and f(true) = $\mathbf{1}$. Such a function is $\lambda x \in$ Bool.if x then VOID else $\mathbf{1}$ So the type

Int = $\mathbf{W}x \in$ Bool.if x then VOID else $\mathbf{1}$

describes the type of non-negative integers.

Given the integers, we can then form REC(f) by $\bigvee n \in$ INT.f^n(VOID), which also remains at the lower level. A version of L(f) can be constructed using similar techniques, but it is no longer a necessary notion.