March 1991

M91-19

Joshua D. Guttman

A Proposed Interface Logic for Verification Environments

MITRE

Bedford, Massachusetts

March 1991

M91-19

Joshua D. Guttman

A Proposed Interface Logic for Verification Environments

CONTRACT SPONSOR RADC
CONTRACT NO. F19628-89-C-0001
PROJECT NO. 4030
DEPT. G117

MITRE

The MITRE Corporation
Bedford, Massachusetts

Department Approval: [illegible]

MITRE Project Approval: Dale M. Johnson

Abstract

This report proposes adoption of an *interface logic* for verification environments, namely, a logic with a syntax that is simple for machines to generate and parse, and which has a standard semantics. It is intended to codify logical presuppositions that are common to a considerable number of efforts in specification and verification, thereby allowing a range of work to be shared among them rather than duplicated. The practicality of the proposals were tested by implementing them in a program called IMPS.

Acknowledgments

I am grateful to colleagues at MITRE for comments and conversations at many stages of this work. In particular, W. M. Farmer and F. J. Thayer helped to design the logic as well as to design and implement the IMPS software that were used to test the proposals made here. L. G. Monk was the designer of a previous version, from which many ideas derived. They, together with M. E. Nadel, J. D. Ramsdell, and J. G. Williams, also read drafts and strengthened it greatly.

This work was supported by RADC/COTC, under Air Force contract F19628-89-C-0001. Almost all of the implementation of the IMPS software was supported by the MITRE-Sponsored Research program (projects 90770 and 91280).

Table Of Contents

Section 1
Introduction

The goal of this report is to advance the idea of an *interface logic* for verification environments (henceforth referred to as VEs). By this we mean a logic with a syntax that is simple for machines to generate and parse, and that has a standard semantics and a sufficient degree of expressiveness. It is intended to codify logical presuppositions that are common to a considerable number of efforts in specification and verification.

Our hope is that it will become possible largely to separate the work of developing formula-generators, such as verification-condition generators and specification-language processors, from the effort of developing theorem-proving software. Currently, research efforts whose primary emphasis is on formula generation often spend a great deal of time and effort developing theorem-proving software to demonstrate that the formulas they generate are susceptible to automated deduction. This effort is often redundant, as the theorem provers are often based on the same classical semantics for first order logic or simple type theory. In this report, we argue that a useful interface logic can be found, and propose a sequence of candidate logics.

We believe that many existing software environments for program or design verification can be easily adapted to generate assertions in the form proposed in this paper, and that many existing theorem provers can be adapted to take input in this form. They appear to include FDM [18], Gypsy [14], EHDM [28], m-EVES and EVES [7], Penelope [22], Ariel, and HOL [15]. [1] It is unfortunate that work done on automated theorem-proving for one of these projects can not be made available to others.

In addition, we expect that many research efforts in verification over the next few years will also be able to use this format. This is not to say that all will, as there is certainly reason to continue to examine non-classical or otherwise unusual logics. Current efforts involving logics substantially different from the classical predicate calculus discussed here include SDVS [23], NuPRL [8], and Romulus (formerly Ulysses) [27].[2] Our intent is not to put a dent in these or similar projects, but rather to define a framework to

[1]The Boyer-Moore theorem prover [2, 3] is an intermediate case: using the technique of skolemization, it may be able to handle the first logic in the sequence we propose.

[2]Possibly, using somewhat more complicated translations, some of these efforts might also be able to benefit from the interface logic.

serve the needs of the large collection of verification environments that are based on similar logics.

1.1 Properties of an Interface Logic

We believe that any logic suited to serve as an interface logic must have three characteristics:

- a simple syntax;
- a widely accepted semantics;
- a sufficient degree of expressiveness.

The syntax of the interface logic needs to be simple so that programs can easily translate between it and whatever syntax may be preferred by a particular formula generator or theorem proving system. Moreover, because the interface logic is intended as a medium of communication between programs, there is no obligation to choose a syntax that will be easy for human users to read and write. Indeed, one advantage of a simple, machine-oriented syntax is that it is then easy to experiment with a range of programs to translate between it and various "user-oriented syntaxes." The choice of a good human interface is thus separated from the selection of a logic to serve as an interface among programs.

A semantics that is already widely accepted is a necessity for two main reasons. First, the logic is intended to serve as a bridge among independently designed components, some of them already existing. It will more readily serve this purpose if its theoretical commitments are similar to those of the majority of the components it will mate. Second, since the logic intended to serve as a *lingua franca*, a large number of researchers will need to have a thorough understanding of its semantics. Happily, the standard semantic approach to predicate logic is suited to play this role, being so well understood, and so widely understood. We would think it was uniquely suited to the role.

The degree of expressiveness of an interface logic is the most delicate of these issues, because it must suffice for a range of formula generators, without being excessively burdensome on theorem provers. One half of the problem is to determine what characteristics are needed so that realistic formula generators will be able to express candidate theorems and the axioms from which they are to be proved. The other half is to determine what cost this expressiveness imposes on theorem provers. Naturally, the less regular

the set of formulas to be used in reasoning, the more carefully designed a theorem prover must be. In order to appraise the usefulness of different kinds of expressiveness, we discuss four general semantic issues. These are:

- Partially defined functions: should they be expressed directly, with the concomitant non-denoting terms, or represented indirectly?
- Overlapping sorts (data types): should they be permitted, or should sorts be assumed disjoint?
- Functions and operators on them: should objects such as functions belong to syntactically distinct sorts of "higher type"? Should there be variable-binding operators such as λ to introduce expressions of these sorts? To what extent are polymorphic operators (operators of *variable type*) needed?
- Simplification and decision procedures: Most theorem provers will supply special simplification and decision procedures, and such a procedure is normally applicable in all theories satisfying certain properties. How can they be organized so that all theories satisfying the properties can access them, independent of vocabulary?

In addition, to evaluate first-hand the difficulties that these cause in the course of theorem-proving, we have implemented some of these elements within a prototype theorem-proving environment called IMPS. We will discuss the issues in detail in in Section 2.

On the basis of our experience in this area, we have drawn four conclusions:

- if possible, the logic should allow partial functions and overlapping sorts;
- it should contain the λ operator to make "higher-order objects" easy to introduce and manipulate;
- the logic should provide notation appropriate for declaring theories, together with theorems, definitions, and theory-specific (partial) procedures for simplification and deciding validity;
- to support the λ operator and partial functions, without excluding too many existing theorem-proving systems, it is desirable to define a succession of logics starting with a first order syntax and introducing the more controversial features in sequence.

Hence, we have devoted the bulkiest part of this report—contained in Sections 3–4—to the syntax and semantics of a sequence of three logical systems. The intent of this portion of the report is to demonstrate, by example, that useful interface logics can be defined without elaborate formal research. We do not consider these logics to be the only reasonable candidates, and hope that this report will stimulate alternative proposals, and suggestions for revisions.

Of the three logics, the first is many-sorted classical first order predicate logic, expressed with a simple, machine-oriented syntax. This version of first order logic is somewhat unusual in that it contains second order free variables, which we have added so that axioms like induction can be expressed without any special machinery for schemas (see also Section 4.5.2). However it has no bound higher order variables. We refer to it as **FQ**, to emphasize that quantification is purely first order in this system.

The second logic, called **ST**, is many-sorted classical simple type theory, which is predicate logic equipped with constants and variables of higher (function) types; in this logic all variables may appear bound. The third logic presented, called **PF**, differs from the second in that it allows terms to be undefined, and functions to be partial; also, sorts may overlap.

First-order logic is included because some quite usable theorem proving systems have very weak support for variable-binding operators, and thus cannot be extended to support the simple type theory we describe. Similarly, we have included a version of simple type theory where functions are total and all terms are defined, even though we believe that partial functions and non-denoting terms are desirable, and that the majority of the needed checking for definedness can be automated. Nevertheless, we recognize that many theorem-proving systems will not be suited to these techniques. Thus it would be not be desirable for all projects to support partial functions.

Not all details have been resolved. Before these logics could be considered a well-specified medium of communication between programs, many detailed questions would need answers. For instance, how are software versions to be specified? How are the lexical roles of individual characters (what in Lisp are called read-tables) to be specified? What particular command names are to be used for a host of necessary operations? This report makes no attempt to answer questions at this level.

The three logics to be described form a naturally ordered sequence in that first order logic is simplest, simple type theory[3] is intermediate, and simple type theory with partial functions and undefined terms is the most complex. Each is a sublogic of the next in the sense that a theory in one logic can be faithfully transferred to the next logic (in a sense to be made precise in Section 4.5.3) by a simple syntactic process.

Hence, it is desirable for projects focusing on formula generators, such as specification processors or VCGs, to stay within the simplest of the three that is consistent with their research and development goals. Conversely, if a project is developing a theorem prover, it should aim at the richest of the three interface logics that can be supported with the approach selected. This strategy will increase the set of compatible pairs of formula generators and theorem provers.

1.2 Method

Our method for reaching the conclusions defined in this paper involved two parts. First, we have consulted a series of papers including [9, 10, 24, 25] on logical issues written at MITRE.

Second, we have considered the lessons of developing an automated reasoning testbed under the funding from the MITRE-Sponsored Research program. This system, called IMPS, an Interactive Mathematical Proof System, is under continuing development [11, 19]. It is based on a version of simple type theory allowing partial functions; its semantics were studied in [10]. Software allows creation of languages and theories (with facilities for reading and printing expressions), and extension of existing theories by definitions. Several mechanisms for reasoning are provided. They include simplification, support for user-invoked tactics on expressions, and a facility for building tree-like deductions interactively or as directed by procedures called strategies. IMPS is equipped with a highly informative user interface for deductions, based on GNU Emacs [29] and TeX[21]. We have used IMPS to ensure that automated deduction systems can implement those logics efficiently. It is now a sophisticated and effective implementation of **PF**, the richest of the logics we will discuss. In addition, it now implements the method of theory interpretations as a substitute for polymorphism. We have found that good support for theory interpretation seems to make it unnecessary to

[3] "Simple type theory" is so-called not because it is intrinsically simple in any sense, but for a historical reason: it was a simplified version of Russell's original "ramified" type theory.

complicate the logic with explicit polymorphism, a point that has long been made by Goguen and his colleagues [4, 12, 13] in reference to programming languages (see Section 2.5).

Section 2
Issues of Expressiveness

In order to explain why we have selected the three logics **FQ**, **ST**, and **PF**, we will with discuss five issues:

- Partially defined functions;
- Overlapping base-sorts;
- Functions and operators on them;
- Simplification and decision procedures;
- Polymorphism and theory interpretation.

2.1 Partially Defined Functions

In this section, we will argue that it is desirable for an interface logic to support partially defined functions and non-denoting terms, as incorporated in the logic **PF**. We have studied how to organize a theorem prover so that non-denoting terms will not be an intolerable burden, and we recommend that the approach be pursued [11]. However, for much current work non-denoting terms would create an intolerable burden, and this report also proposes the more traditional logics **FQ** and **ST**, in which all terms are defined.

Partially defined functions and non-denoting terms are ubiquitous in both mathematics and computing. Mathematical examples of non-denoting terms come in many flavors:

- $x/0$;
- 0^0;
- $d(|x|)/dx$ at 0;
- $\lim_{x\to\pi/2} \tan x$.

Partially defined functions are also, in some modeling approaches, natural ways of representing various "bad" behaviors of computer programs, such as non-termination or abnormal termination for certain values of parameters.

Thus any system for reasoning about computer programs or rigorous mathematics must either allow partial functions, or else provide some alternative mechanism for coping with these issues. (See [10] for a more detailed and inclusive discussion of the range of options.)

We shall consider a version of the partial functions approach in which *terms* may be undefined, but *formulas* are always either true or false. Our work indicates that this is a quite natural compromise, in that it delivers the advantages of a direct treatment of partial functions, while causing a minimal disruption to the patterns in reasoning familiar from classical predicate logic and standard mathematical practice. Indeed, much standard informal mathematics can be formalized quite smoothly using this framework. We will refer to this as "the direct approach" to the partial functions problem; we hope that this terminology will not be thought contentious.

It is clear that the rules for using existential and universal quantifiers must be modified to be sound on the direct approach. For instance in a single-sorted context, using $t\downarrow$ to express the assertion that t is defined, we would have the rule of existential generalization in the form:

$$\frac{\phi(t) \qquad t\downarrow}{\exists x\, \phi(x)} \quad \text{instead of} \quad \frac{\phi(t)}{\exists x\, \phi(x)}.$$

Similarly, substitution depends on definedness, in that $\phi(s,t)$ follows from $\phi(x,y)$ by applying the substitution $\langle x \to s; y \to t\rangle$ only on the additional assumption that $s\downarrow \wedge t\downarrow$. However, few rules need changes, and the notion of model is also not much affected.

In traditional mathematical logic, two techniques are used to avoid partial functions and non-denoting terms. One is to ensure that a function that would be partial is not represented by a function symbol. Instead, its graph, a predicate with one extra argument place, may be used. This is perfectly satisfactory for metalogic, but it is inconvenient when concrete formulas must actually be used in reasoning. The reason for this is that in most contexts, the extra argument place must be filled with a variable, which is quantified. Thus rewriting $x/y = z$ by its graph $\mathrm{div}(x,y,z)$, we must transform $(x/y)/w = 3$ into a formula such as:

$$\forall z_1, z_2\, \mathrm{div}(x,y,z_1) \wedge \mathrm{div}(z_1,w,z_2) \Rightarrow z_2 = 3.$$

Clearly, this translation method becomes too cumbersome for deeply nested terms. The other common technique is to introduce a function symbol which represents some "totalization" of a partial function. Normally, it is unspecified which totalization is meant. Thus, one might introduce a *function*

symbol $\mathrm{div}(x,y)$ with the axiom $y \neq 0 \Rightarrow y * \mathrm{div}(x,y) = x$. It is then unspecified what the value of $x/0$ is, although it is something. This technique is more useful in practice than the first.

Comparison: Mathematical Examples. How does this "total functions with unspecified values" approach compare to the direct approach? In simple cases, they seem to be interchangeable. In the direct approach, one has the axiom

$$\forall x, y, z\, [x = z/y \Rightarrow x * y = z];$$

however, one must ensure that z/y is defined before using the axiom to simplify $(z/y) * y$ to z. In effect, y must be shown to be non-zero. In the "total functions with unspecified values" approach, the corresponding axiom is stated:

$$\forall y, z\, [\neg(y = 0) \Rightarrow z/y * y = z].$$

Here again, y must be shown to be non-zero.

Nevertheless, we think that in mathematically (or computationally) more complex cases, the direct approach is far superior. For instance, if f is a real function, consider f', the first derivative of f, evaluated at some real number x. Is $f'(x)$ defined? There are familiar ways of answering this kind of question. But, is there any good way to write down a necessary and sufficient condition, comparable to $\neg(y = 0)$ in the example above? Well, perhaps one could use the definition of derivative:

$$f'(x) = \lim_{t \to x} \frac{f(t) - f(x)}{t - x}.$$

But the question is whether this limit exists. Would it be fruitful to write out the necessary and sufficient condition for such a limit to exist? It seems unworkable to include such conditions in every formula that asserts something about the value of $f'(x)$. Certainly it is far better to be able to write:

$$f'(x)\downarrow \Rightarrow \phi(f'(x)).$$

When it comes time to check that the formula is applicable for a particular f and x, then the usual facts about the definedness of derivatives can be applied.

Our case is strengthened if we let the situation be even slightly more complex. Suppose that f is itself defined as the (pointwise) limit of a sequence $\langle f_i \rangle$ of functions. There is now a question of where f is defined. We would like to say that if the limit of this sequence of functions is not defined for some open interval, then $f'(x)$ is not defined for x in that interval. However, on the "total functions with unspecified values" approach, we cannot be sure. Perhaps, in some model, the unspecified values of the (artificially total) function f are constant in that interval. Then in fact f' is defined (and $= 0$) within the interval. This is not an easy way to do mathematics.

Comparison: Programming Language Example. A similar argument also applies to formal reasoning about program behavior. Many programming languages currently of interest contain the higher order operators on programs that would make this line of argument apply. We will give a simple example, expressed with the notation and terminology of Scheme. We will use the word *thunk* to mean a procedure with no parameters, so that if the Scheme expression `p` represents a thunk, then `(p)` returns the value of executing `p` without arguments in the current state (and variable-binding environment, etc.).

Consider a function `if*` which takes three arguments, assumed to be thunks.[4] It behaves as an if-then-else operator. `(if* test conseq alt)` first evaluates `(test)` in the current state s_0. If that terminates in state s_1, returning value v, then either `(conseq)` or `(alt)`—depending on whether v is distinct from `'#f`—is evaluated in s_1.

Formally, we will consider a thunk to be a (mathematical) function of the state in which it is executed; the values are pairs of the form $\langle s, v\rangle$ meaning that execution terminates in state s and returns value v. The meaning of an expression such as `(if* test conseq alt)` is given in the same way, and if `e` is an expression, then we will use $[\![\texttt{e}]\!]$ to refer to its denotation, which is a function from states to pairs $\langle s, v\rangle$. We will think of non-termination in terms of partial functions, so that if `(test)` does not terminate when executed starting in s, then its denotation is a function that is undefined for the argument s.

On the direct approach, we would formalize the semantics of `if*` as follows.

1. If $[\![\texttt{(test)}]\!]s_0$ is undefined, then $[\![\texttt{(if* test conseq alt)}]\!]s_0$ is undefined. Otherwise, suppose $[\![\texttt{(test)}]\!]s_0 = \langle s_1, v_1\rangle$.

[4] I.e., some sort of error arises if any argument is not a thunk. However, for present purposes, let us suppose that no procedure ever raises an error.

2. Suppose $v_1 \neq$ `'#f`. If $[\![\texttt{(conseq)}]\!]s_1$ is defined and $= \langle s_2, v_2\rangle$, then:

$$[\![\texttt{(if* test conseq alt)}]\!]s_0 = \langle s_2, v_2\rangle.$$

Otherwise, $[\![\texttt{(if* test conseq alt)}]\!]s_0$ is undefined.

3. Suppose $v_1 =$ `'#f`. If $[\![\texttt{(alt)}]\!]s_1$ is defined and $= \langle s_2, v_2\rangle$, then:

$$[\![\texttt{(if* test conseq alt)}]\!]s_0 = \langle s_2, v_2\rangle.$$

Otherwise, $[\![\texttt{(if* test conseq alt)}]\!]s_0$ is undefined.

By contrast, on the "total functions with unspecified values" approach, we cannot represent the semantics of `if*` properly.

To prove this, consider the case in which $[\![\texttt{(test)}]\!]s_0$ is intuitively undefined. In each model, it must be supplied with some arbitrary value $\langle s_1, v_1\rangle$. Unfortunately, in any particular model v_1 is either `'#f` or not. If not, then in this model:

$$\exists s_1 \,.\, [\![\texttt{(if* test conseq alt)}]\!]s_0 = [\![\texttt{(conseq)}]\!]s_1.$$

If $v_1 =$ `'#f` in this model, then

$$\exists s_1 \,.\, [\![\texttt{(if* test conseq alt)}]\!]s_0 = [\![\texttt{(alt)}]\!]s_1.$$

Hence, the semantics predicts that the disjunction of these two cases is valid. However, this is not correct. For instance, suppose both `conseq` and `alt` are:

```
(lambda () 1).
```

Then it would follow that $[\![\texttt{(if* test conseq alt)}]\!]s_0 = 1$ holds true in every model. But this is absurd, because `(if* test conseq alt)` is non-terminating when started in s_0, so that its denotation may be arbitrarily chosen.

Conclusion. These examples suggest that, if partial functions are to be avoided, it may be preferable to add a new "bottom element," or possibly several new "erroneous elements," to the models under consideration. Then, rather than "completing" a partial function by giving it *unspecified* values outside its natural domain, one would complete it by giving it these new values. However, in the case where only one new "bottom element" is added, there is a simple translation between this and the direct approach. Moreover, the latter has the advantage that a theorem prover can "know how to reason" with the "bottom element," and can thus hide some of the routine checking that expressions are defined/non-bottom from the user.

For these reasons, we advocate research on effective theorem proving in logics supporting partial functions along the lines we have termed the "direct approach." However, because this is not compatible with much valuable current work, we also present the more traditional logics **FQ** and **ST**. Our goal in defining them was to do so in such a way that verification environments using these logics would be able to migrate to **PF**, our logic with partial functions, as theorem proving systems supporting it develop toward maturity.

2.2 Overlapping Sorts

Our second question concerns sorts: should we regard them as being disjoint, or should they be allowed to overlap? By a *sort* we mean a set of objects that are treated as belonging to "a single kind" for the purposes of some theory. Natural numbers, real numbers, and real functions are some familiar examples of sets of objects that are frequently treated as sorts. Similarly, the data types of programming languages will be regarded as sorts.

The question whether sorts should be allowed to overlap is tied to the treatment of partial functions. For instance, consider the natural numbers $\mathcal{N}$ and the real numbers $\mathcal{R}$. The untutored view is that $\mathcal{N}$ is included in $\mathcal{R}$. Now the function *gcd* makes sense only on natural numbers (or integers, but in any case not outside the integers). If i, j are variables ranging over $\mathcal{N}$, then all is well and good: $\gcd(i, j)$ should be well-defined. But what about the variables x, y, ranging over $\mathcal{R}$? If the logic does not support undefined terms, then the expression $\gcd(x, y)$ is undesirable: it is almost always undefined. However, if $\mathcal{N}$ is included in $\mathcal{R}$, then we cannot reasonably prohibit expressions of the form $\gcd(s, t)$ where s and t behave syntactically as real valued terms. For, their values may in fact be in $\mathcal{N}$, so that $\gcd(s, t)$ is well-defined. For this reason, we allow overlapping sorts in **PF**, which is suited to reasoning with possibly undefined terms. On the other hand, in **FQ** and **ST**, we take the view that sorts do not overlap.

Very frequently, when two sorts σ and τ overlap, one, say σ, is actually included within the other. In this situation, we expect a theorem prover for **PF** to be able to recognize efficiently that if a term is defined with a value of sort σ, then it also has a value of sort τ. It was easy to ensure this in IMPS.

However, ther are also cases where two sorts may overlap, although neither is a subsort of the other. Consider a language with a basic sort *set*. That is, sets are among the individuals treated by this language. Moreover, some (but not all) sets represent partial functions from sets to sets. However, not all partial functions from sets to sets are represented by sets. Suppose we think of the "representation" relation here as being simply identity, as there is no reason why we should not. Then the sorts *set* and *set→set* overlap, although neither is contained in the other.

Two fine points should be added. First, we do not in fact restrict the *semantics* of **FQ** and **ST** to avoid overlapping sorts. Instead, we have arranged their *syntax* so that assertions that would be sensitive to whether sorts really overlap or not simply do not occur in the logics. This point is discussed in detail in Section 4.5.1. Second, when a user wants to discuss the relation between $\mathcal{N}$ and $\mathcal{R}$ in **FQ** and **ST**, he can do so using coercion functions. A coercion function between two sorts is a one-one function defined on the included sort, taking values in the other. By defining a suitable coercion, statements about the relations between, say, the multiplication operations in $\mathcal{N}$ and $\mathcal{R}$ can be expressed. The relations between sorts are more cumbersome to express in these logics when there is merely an overlap between them, without either including the other. However, they require no special machinery in **PF**.

2.3 Functions and Operators on Functions

We refer to a sort τ as a *higher sort* if it consists of the n-ary functions taking arguments in sorts $\sigma_1, \ldots, \sigma_n$ (its *domain sorts*) and yielding values in a sort σ_0 (its *range sort*). A higher-order operator (or an operator on functions) is a function one of whose domain sorts is itself a higher sort.

We will argue that functions and operators on them are ubiquitous, not only in mathematics, but also in computing. For this reason, it is essential to be able to construct expressions referring to functions easily, and to be able to apply complex expressions built using operators on functions. The familiar λ constructor serves this purpose. Our experience with IMPS indicates that it is no harder to simplify expressions involving the λ constructor than expressions involving other variable-binding constructors. For this reason, we would conclude that a theorem-proving system that handles nested quantification in the straightforward way should extend to handle the λ constructor. System that eliminate nested quantification in favor of Skolem-functions, however, may not be extensible to **ST**. In essence, we have included the logic **FQ**, which lacks the λ constructor, so that theorem provers lacking genuine variable-binding constructors would be able to support a form of the interface logic. The decision is reinforced by the fact that some approaches to verification, such as Hoare logics (or VCGs) for certain programming languages, do not make use of expressions of higher types.

We can point to a large class of higher-order operators; to use them effectively, we naturally need to be able to refer to the objects we want to apply them to. Examples would include:

- Sum and product: The operators $\sum$ and $\prod$ have several usages. According to one usage, they apply to a predicate (e.g. of integers) and a function (taking integer arguments). The value is determined by summing (or multiplying) the values of the function for all arguments passing the test expressed in the predicate. For instance:

$$\lambda n \,.\, \prod_{\lambda j \,.\, 1 \leq j \leq n} \lambda j \,.\, j.$$

  This expression defines the factorial function. In this usage, the value is undefined if infinitely many j satisfy the predicate. According to another usage (most commonly involving $\sum$), the operators take a single argument, a sequence (e.g. of reals). In this usage, $\sum f$ is the limit, as n increases, of the partial sums $f(0) + \cdots + f(n)$. It can thus be defined in terms of limit and the first usage of $\sum$.

- Limit, differentiation and integration.

- In computing, list-oriented and stream-oriented [1] mapping functions.

- Higher order procedures, such as those in ML [16], or operations, such as those in T [26], which are applicable to procedures and other operations.

- Fixed point and direct limit operations prevalent in denotational semantics.

A logic that is intended to be useful for sophisticated reasoning in mathematics and the theory of computation needs to be able to express these operators. Moreover, it needs a uniform and clear way to build up expressions for the higher-type objects to which they apply. While λ is by no means the only way to express these objects, it is familiar and comprehensible, and, in our experience, not difficult to reason with.

What is involved in reasoning with expressions constructed with the λ operator? In addition to the need to be able to apply β-conversion (the rule normally stated as $(\lambda x \,.\, t)t' = t[t'/x]$), and to recognize when two expressions differ by α-conversion, it is also important to be able to simplify the insides of λ expressions. We have found, in working with IMPS, that simplifying the body of a λ expression presents no essential problem. The main requirement is the following. When v is a variable bound by the λ operator, then information about v, known in the context of simplification, must not be applied to occurrences of v inside the λ expression. But this requirement that the scope of bound variables must be respected is exactly the same requirement that must be observed for any other variable-binding operator, such as $\forall$ or $\exists$.

2.4 Simplification and Decision Procedures

We have discussed a number of desirable properties of an interface logic. However, do these properties make its use in a verification system impractical? One of the features that we expect from a verification system is that it recognize simple inferences that depend for their validity on the theory in use. Reasoning with partial functions or higher order functions is more delicate, possibly even to the extent that it might be unreasonable to expect verification systems to incorporate these features in the foreseeable future.

In order to test these ideas we attempted to provide our testbed IMPS system with a class of procedures that can carry out theory-dependent inferences for the user; we have called these procedures *tactics*. A particularly important tactic is simplification; this tactic serves to "tidy up" both the logical and term structure of an expression. One of the main contributions here has been the way we handle possibly undefined expressions which might cancel out in the course of a simplification. Despite this apparently stringent restriction, the IMPS simplifier is able to get its job done quickly and effectively. It is no surprise that allowing partial functions in theories introduces a difficult problem of checking definedness of expressions. However, one of the significant lessons that we have learned from the IMPS testbed is that these difficulties can be overcome.

In part, the success we are reporting is a result of the specific design of the IMPS system, mainly the way algebraic simplification is organized and the way theory-specific information concerning definedness is handled. Consequently, our conclusions may not be applicable to systems with fundamentally different designs.

However, a broadly relevant contribution is the design of the IMPS simplifier itself, which is essentially vocabulary independent. This means that the same simplification procedures can be used for theories with (e.g.) ring-like operations independently of the underlying language. This is clearly useful in situations (such as those that motivate the need for an interface logic in the first place) in which the exact structure of the language to be dealt with by the simplifier is not known beforehand. For additional information, consult [11].

2.5 Polymorphism or Theory Interpretation

The logics in this report do not contain any explicit mechanism for defining theories with *polymorphic* sortings and expressions. Our opinion is that it will not be necessary to do so, and that the version presented here will be adequate. We believe that the mechanism of theory interpretation, long advocated by Goguen in the case of programming languages [4, 12, 13], and examined by us in [9], is a theoretically appealing and practically efficient alternative. Moreover, we have implemented a mechanism for theory interpretation in IMPS, our testbed system, and find it to be highly effective.

In essence, this problem arises out of the need to have *generic* theories and specifications. Consider the datatype *"list of integers"*. It is easy to specify the behavior of the associated primitives, such as `cons`, `car`, and `cdr` in Lisp terminology. However, the datatype *"list of floating-point numbers"* is just the same, except that the elements of the list are floating-point numbers rather than integers. Thus, there is a need to be able to construct a datatype *"list of elements"*, where *elements* form some other datatype; it will be determined later *which* other datatypes will be in question.

This situation is extremely common; examples include the relation between:

- graphs and their vertices;
- matrices and the members of the ring that supplies their elements;
- statements about limits and continuity, and the underlying topological space.

In addition, polymorphism within a single theory is also familiar; think of the predicate "is a total function," which can be applied in a uniform way to functions of many sortings within a theory.

One approach, developed in ML and proposals for specifications based on ML [16], is to designate certain sortings as variable sortings or polymorphic sortings. An "instantiation" of the theory allows one to specify concrete sortings to associate with the variables. The same polymorphic theory can be used in many instantiations, and expressions of the theory with polymorphic types can be instantiated by other expressions of the same theory to support polymorphism within a single theory.

An alternative, which we find simpler and consider to be adequate, is to allow theories to have only "straightforward" sortings. To apply the theory to instantiations, a map is set up from it, as the source theory, into a second theory. This map translates formulas in the source theory to formulas in the target theory. The requirement on a theory interpretation is that the image of any axiom, under the translation, should be a theorem of the target theory.

Typically, the translation assigns "concrete meanings" in a different vocabulary to a few primitives of the source theory; the remainder of the source theory is then transferred unchanged. The supplementary vocabulary normally belongs to the theory of a concrete data type, so we will call the theory that supplies it the concrete theory. When there is no conflict of vocabulary, the target theory can be constructed directly from the source theory and the concrete theory. The language of the target theory is formed by adding the vocabulary of the concrete theory to that part of the vocabulary of the source theory that is transferred unchanged. The axioms of the target theory are those of the concrete theory together with the translations of the axioms of the source theory. When a theory is intended to be used as the source of interpretations, we call it a "generic theory." However, we emphasize that generic theories are not a special kind of theory; they are just intended to be used in special way.

One implementation idea serves to make interpretations more effective. There are a number of generic theories that are applicable to every concrete theory, and typically in very many ways. For instance, consider the theory of pairs with first element chosen from sort σ and second element chosen from sort σ'. There are a few basic facts about pairs that should be chosen once and for all, and should then be available to be applied to *any* σ and σ'. It might seem cumbrous to have to define vocabulary and justify an interpretation for every pair of sorts that one might want to pair together. However, as is well known [15], pairs can be represented in simple type theory in a purely logical form. The pair of a of sort σ and a b of sort σ'

can be identified with the predicate *pair(a,b)* defined to be:

$$\lambda x : \sigma \ y : \sigma' \, . \, x = a \wedge y = b.$$

The first component of such a pair ϕ can then be represented as:

$$\iota x' : \sigma \, . \, \exists y' : \sigma' \, . \, \phi = \mathrm{pair}(x', y').$$

Using such ideas, we can represent all the basic operators involving pairs in pure type theory. Thus, the expressions needed always exist in any concrete theory. Thus, we may define patterns for expressions. Given two expressions a and b, the *pair* pattern chooses appropriately sorted variables x and y, and builds the λ-expression above. Naturally, it can be arranged that theorems about generic pairs are available in the course of building up proofs involving concrete pairs. Similarly, theorems with special forms can be treated as rewrite rules and installed in a simplifier.

An additional source of a weak polymorphism available in **PF** derives from the existence of overlapping sorts. This so-called *inclusion polymorphism* [5] is very convenient, as operators can be defined an an inclusive sort—and theorems proved—and then transferred to other sorts that are contained within it.

Section 3
Syntax and Informal Semantics

Although the syntax has been developed in connection with a Lisp implementation, it is intended to be easy for programs of many kinds to generate, read, and otherwise manipulate. We will refer to it as an "s-expression syntax," because expressions have the appearance of nested lists, which in Lisp parlance are referred to as s-expressions. Also as in Lisp, all operators are placed in prefix position.

It should be emphasized from the start that the s-expression syntax is intended to be used as a medium of communication between programs. Thus, we do not consider it an objection that this syntax is not aesthetically pleasing to certain tastes. On the contrary, one advantage of an s-expression syntax of the kind we will describe is that it is easy to write programs that translate between it and "user-oriented syntaxes."

We have in fact implemented procedures to translate between this syntax and a more familiar mathematical syntax for expressions, which we call a "string syntax." The string syntax supports infix and prefix operators, and settable precedence relations between different operators. However, it represents less than half a week's labor.

We have also implemented procedures to generate TEX output from expressions in the s-expression syntax. Using these procedures, mathematically meaningful expressions can be viewed on a bit-mapped screen in the form that a mathematician would most normally expect to see them. The TEX interface represents a similarly small investment of effort. Thus we would argue that the s-expression syntax represents a valuable "canonical form;" a variety of pleasing alternative forms can easily be generated from it or reduced to it.

Indeed, in the following sections we do not attempt to express all formulas in the canonical s-expression syntax. Whenever readability is a primary concern, we use familiar conventions to make expressions more compact. However, the rules for converting between our informal "user-oriented syntax" and the official s-expression syntax are very simple.

We assume that implementations will be able to group characters into lexical units in a reasonable way. In what follows, we shall use the word *symbol* to refer to a string of characters making up a lexical item used as name of a constant or variable. A string of characters that represents a number will be called a numeral.

Logical expressions are to be represented in the form of s-expressions, where the class of s-expression is defined inductively:

- A symbol or a numeral is an s-expression;
- If $s_1, \ldots s_n$ (where $0 \leq n$) are all s-expressions, then so is the result of enclosing them within parentheses, separated by whitespace: $(s_1 \ldots s_n)$.

The syntax must represent items of four different kinds:

- *sortings*, indicating the type of values that expressions have;
- *languages*, indicating the vocabulary of a formal language, and the sortings of the constants of the language;
- *variable-lists*, declaring a sequence of variables and indicating their sortings, which are used to allow a variable name to appear in formulas without an explicit indication of its sorting;
- *expressions*, the meaningful linguistic units in the logic: they may be constants or variables, or else compound expressions built up by *constructors*, the logical constants of the system; and

3.1 Types and Sorts

A *type* is a non-empty set of objects, which are treated as being "all of a kind" in a particular theory. In **FQ** and **ST**, the set of objects that a variable ranges over is always a type. In **PF**, a type may have smaller sets within it, called *sorts*, that are also of special significance, and have variables ranging over them. A *syntactic type* or *sorting* will be a *syntactic* form that specifies either a type or a sort. We will try to use the words *type*, *sort*, *syntactic type*, and *sorting* consistently. The former two will always denote something semantic, namely a set or "domain;" the last two will always denote something syntactic, which is used to specify a domain.

3.1.1 Type Structures

The types available in **ST** form a structure familiar from simple type theories [6], except that we allow any number of base types of individuals. The types available in **FQ** are built up in the same way, except that only the lowest

layers are available. **PF** will make use of these type structures, adding additional ingredients to accomodate sorts properly included in types.

A type structure consists of three main ingredients.

1. A finite number of base types of individuals. These will be considered the domains of the structure.
2. The set of truth values $\{T, F\}$, which we shall refer to as `prop`.
3. The *function types* of the form $\tau_0, \ldots, \tau_{n-1} \rightarrow \tau_n$, whenever all of the τ_i are types. This is construed as the set of n-ary functions taking arguments from $\tau_0 \times \ldots \times \tau_{n-1}$ and yielding values in τ_n.

A type is a base type if it is either `prop` or else one of the base types of individuals. It is said to be of *kind* `prop` only if it is `prop`; otherwise it is said to be of kind `ind`. A function type is said to be of kind `ind` or `prop`, depending as the range type τ_n is of kind `ind` or `prop`. We sometimes use the phrase "prop-sorted" to mean "of kind `prop`".

In **FQ** and **ST**, the function types are assumed to contain all total functions from their domains to their range. In **PF**, a function type of kind `ind` contains all partial functions from its domains to its range (including those functions that happen to be total). However a function type of kind `prop` contains only the total functions from its domains to its range. We believe that it is very difficult to preserve laws similar to the familiar rules of classical logic if (properly) partial functions are included in the function types of kind `prop` [10].

3.1.2 Sort Structures

Sorts (that is, subtypes) in **PF** are organized into *sort structures*. A sort structure consists of a type structure together with:

1. A finite set of *named sorts*, each a non-empty subset of a designated type in the type structure;
2. The *function sorts* of the form $\sigma_0, \ldots, \sigma_{n-1} \rightarrow \sigma_n$, whenever all of the σ_i are sorts.

As above, a function sort of kind `ind` contains all partial functions from its domains to its range (including those functions that happen to be total), while a function sort of kind `prop` contains only the total functions from its domains to its range. Thus in particular, suppose σ_0 is a subsort of σ_0',

$f : \sigma_0 \to \sigma_1$, and $a \in \sigma_0' \setminus \sigma_0$. Then $f(a)$ is undefined if σ_1 is of kind `ind` (see Section 4.4 for the treatment of the case in which σ_1 is of kind `prop`).

If σ is a sort, we define the type of σ as follows:

1. If σ is a type then the type of σ is σ;
2. If σ is a named sort, then its type is the designated type in which it is included;
3. If σ is a function sort $\sigma_0, \ldots, \sigma_{n-1} \to \sigma_n$, and the type of σ_i is τ_i for each i, then the type of σ is $\tau_0, \ldots, \tau_{n-1} \to \tau_n$.

We write $\tau(\sigma)$ for the type of σ.

3.2 Specifying Sortings

As a sort is either a type (or named sort) or a function sort, a linguistic entity representing a sort—which we will refer to as a *sorting*—will be either a symbol or a list representing domains and range. Thus the syntactically atomic sortings are symbols such as `RR`, `ind`, or `prop`. We shall call the sortings representing function sorts *higher sortings*. They have the form:

$$(s_0 \ldots s_n \; s_{n+1})$$

This represents the sort of functions taking arguments from the sorts represented by $s_0 \ldots s_n$ and yielding values in the sort represented by s_{n+1}. In this we assume that $0 \leq n$. The syntax makes no provision for 0-ary functions: we see no reason to introduce a 0-ary function with a value in sort σ as an object distinct from its sole value.

To specify a particular type structure for some formal theory, we must give a finite number of symbols to refer to base types of individuals. The type symbol `prop` is always reserved for the type of propositions, containing just the values *truth* and *falsehood*.

For **PF**, to introduce the syntactic machinery needed for a formal theory we must also specify a sort structure. In specifying formal languages and reasoning about sort-inclusion, we have found it valuable to have syntactically fixed information about the order of various sorts within a type (under inclusion). Thus, we associate an *enclosing sort* of the same type with each named sort. The enclosing sort is required to include the named sort.

To specify sorts and their relations of inclusion, we introduce a sequence of pairs of the form:

(*named_sorting enclosing_sorting*).

Each *named_sorting* is a symbol, and each *enclosing_sorting* is a sorting, i.e., a type symbol, a previously introduced atomic sorting or else a higher sorting in which only previously introduced atomic sortings appear. Naturally, the type associated with a named sorting is the type associated with its enclosing sorting.

Since the enclosing sort must have been previously introduced, there can be no cycles, and the relation generates a finite partial ordering. We extend this partial ordering to function sorts by stipulating that

$$\sigma_1, \ldots, \sigma_n \to \sigma_0 \quad \text{is below} \quad \sigma_1', \ldots, \sigma_m' \to \sigma_0'$$

if $m = n$ and for each i where $0 \leq i \leq n$, σ_i is below σ_i'. If two sorts have the same type, then that type is an upper bound for them in this ordering. Moreover, because each atomic sort has a single enclosing sort, an inductive argument shows that any two sorts of the same type have a least upper bound.

We divide expressions into a few different categories depending on their kind and height:

- An expression having lowest syntactic type is a *formula* if it is of kind `prop` and a *term* otherwise;
- An expression of higher syntactic type is a *predicator* if it is of kind `prop` and a *function* otherwise;
- A predicator with range sorting `prop` is also called a *predicate*.

Note that by a predicate we do not necessarily mean a single formal symbol, a constant; on the contrary, we will also call complex expressions of this kind predicates.

3.3 Languages

Languages are of two species, namely basic languages and compound languages. Moreover, basic languages are divided into two varieties, namely self-extending languages and non-self-extending languages. For the moment, we will ignore self-extending languages.

A (non-self-extending) basic language has a name, and declares a number of base sorts and constants. A compound language cites one or more already-defined languages to be included as a starting-point, and then declares a number of additional base sorts and constants.

Thus, we need clauses of four kinds to express language declarations.

- An *embedded-language* clause has the form `(embedded-languages` *language-name1 ... language-nameN*`)`, where each *language-name* is a symbol.
- A *base-types* clause has the form `(base-types` *type-name1 ... type-nameN*`)`, where each *type-name* is a symbol.
- A *named-sorts* clause has the form `(named-sorts (`*sort-name1 enclosing-sort1*`) ... (`*sort-nameN enclosing-sortN*`))` where each *sort-name* is a symbol and each *enclosing-sort* is a sorting containing only `prop`, `ind` (in the case of **PF**), and previously introduced type and sort names.
- A *constant* clause has the form `(constants (`*constant-name1 sorting1*`) ... (`*constant-nameN sortingN*`))`, where again all of the *constant-names* are symbols and the *sortings* are sortings.

A language declaration for a non-self-extending language then has the form `(`*language-name clauses*`)`, where *language-name* is a symbol, which will serve as a name for the language being declared, and *clauses* is a set of embedded-language, sort, and constant clauses, containing at most one of each.

The language thus declared has, as its set of sorting symbols, the union of those in all embedded languages, together with those declared in the sort clause and `prop`. The set of constants consists of all those in any embedded language, together with those declared in the constant clause. It is an error for the same symbol name to appear as a constant with two different sortings.

Self-extending languages are an attempt to support numerical types smoothly. The problem with numerical types—such as, say, the integers, or the integers mod 61, or the 3×4 matrices of reals—is that there are a large (or infinite) number of constants in the language. A language for the integers mod 61 should be able to read the constant 67 mod 61, or print it if it is generated in the course of simplification. However, the specifier cannot be expected to declare all of these constants at the time that he introduces the language. What he can do is to associate a numerical type with a base sorting. Whenever the implementation reads a new, undeclared token, the token is checked against the numerical types. If it satisfies one, then a new constant having the corresponding sorting is created.

Although we have used the notion of a numerical type, but there is no unique meaning in common currency. For our purposes, we will let the exact meaning be determined by the various implementations. All that is needed now is that every numerical type in the relevant sense should be denoted by some symbol. If the implementation reads a token that would normally generate a value of that type, as for instance a normal Lisp system, reading the token `12/56`, would generate a rational number, then the token is associated with the numerical type `rational`. For instance, if we stipulate the association `((non-negative-integer NN) (integer JJ) (rational QQ))`, then a new constant written `-12` will automatically be read into sort `JJ`, while `12/56` will be read into sort `QQ`.

A *self-extending-language clause* has the form `(extensible (`*num-type1 sorting1*`) ... (`*num-typeN sortingN*`))`. All of the *num-types* and *sortings* are assumed to be symbols.

A language declaration for a self-extending language then has the form `(language-name` *clauses*`)`, where `language-name` is a symbol, which will serve as a name for the language being declared, and *clauses* is a set of self-extending-language, sort, and constant clauses, containing at most one of each.

Self-extending languages never have embedded languages. Instead, we require that the self-extending part of a language be declared as a separate unit, which can then be included with other languages of any kind in further languages. This convention does not reduce the power of the self-extending language mechanism, and we found that it simplified the checking done by the implementation when languages are to be declared.

Any language declaration as defined above is acceptable for **ST** and **PF**. A language declaration is acceptable for **FQ** if the sorting associated with each constant is either a base sorting or else a list of base sortings. Nested sortings are not permitted in **FQ**.

3.4 Expressions and Variable Lists

Expressions are built up from constants and variables by constructors. Constants are determined by declaring a language, and language declaration allows us to determine the sorting of any symbol serving as a constant. As to variables, we require that a variable have two attributes, a *name*, represented by a symbol and a *sorting*. A single variable occurs at two different occurrences in an expression if and only if the symbol is the same and the sorting is the same. But how is the sorting associated with a variable name at some occurrence in a formula to be indicated?

It would be too restrictive, and ultimately unworkable, to have to state the sortings associated with all variable names once and for all when a language is declared. For instance, many basic properties of logic depend on the fact that there are available an unbounded number of distinct variables of each sorting. One solution to the problem is to display, at every occurrence of a variable, not only its name but its sorting also. However, this makes formulas too unwieldy in practice. We will in fact choose to do it by associating the variable names with sortings in a structure we call a *variable list* at the start of a relevant syntactic unit. However, we will introduce our syntax for variable lists in Section 3.4.2. Until then, we will represent variables and their sortings in an explicit form. Thus, we will write:

[*sorting name*]

for the variable with name *name* and sorting *sorting*. For instance, a variable named `f` intended to range over real functions will be written:

```
[(rr rr) f].
```

Similarly, when we need to present a list of variables—for instance, the list of variables bound by a quantifier or by λ—we will present it simply as a list of variables with explicit sortings. An example would be:

```
(lambda ([(rr rr) f] [rr x])
  (apply-operator [(rr rr) f] [rr x]))
```

This way of associating sortings and variable names should be regarded as an abstract syntax, and the syntax of variable lists introduced in Section 3.4.2 as a corresponding concrete syntax. We will show in Section 3.4.11 how to eliminate the concrete syntax in favor of the abstract syntax.

Constructors are the recursively applicable items used to build complex expressions. They serve to represent the logical connectives. Examples of constructors include equality, the propositional connectives, and the quantifiers. Another important constructor is `apply-operator`. It serves to build up "applications," formed by applying an *operator*—either a function or a predicator—to arguments. As an example of a compound expression in **PF**, we would offer the equation expressing the core of the binomial theorem; formatted automatically using IMPS facility for TeX typesetting, this expression reads:

$$\textbf{with } m,b,a: \quad (a+b)^m = \sum_{k=0}^{m} comb(m,k)\cdot a^{m-k}\cdot b^k.$$


```
(= (apply-operator ^ (apply-operator + [rr a] [rr b]) [zz m])
   (apply-operator
    sum 0 [zz m]
    (lambda ([zz k])
      (apply-operator
       *
       (apply-operator
        *
        (apply-operator comb [zz m] [zz k])
        (apply-operator ^ [rr a]
                        (apply-operator sub [zz m] [zz k])))
       (apply-operator ^ [rr b] [zz k])))))
```

Figure 1: The Binomial Equation with Explicit Variable Sortings

This equation is valid only under the assumptions that a and b are non-zero, and m is positive. In the s-expression syntax with explicit variable sortings at every occurrence, this has the form given in Figure 1. The constructors appearing here, besides `apply-operator`, are =, `lambda`, and `and`. The constants, which in this case belong to a language for real arithmetic, are 0, `+`, `*`, `^`, `<=`, and `sum`. This last is a higher order operator of sorting `(zz zz (zz rr) rr)`. That is, two integers i and j, serving as lower and upper bounds, and a function f from integers to reals, it returns a real number as value when defined. The value is intended to be the sum of the values $f(k)$ for all values k such that $i \leq k \leq j$. The cumbrousness of this example will also make clear the desirability of avoiding explicit sortings for every occurrence of variables; in the somewhat more compact notation that will be introduced in Section 3.4.2, this equation reads as given in Figure 2. However, for the purposes of defining the syntax of the interface logic, it will be best to start with the most explicit form, in which the sorting of each variable is explicitly presented.

We should emphasize that even in form with implicit sortings, the interface logic is by no means compact. However, because it is so simple, it is very easy to write programs that manipulate it, for instance programs to print and parse formulas in appealing syntaxes. The IMPS system has two separate parsers and three printers. The user can switch between them with a single command, even while in the middle of a proof. Hence, our

```
(with ((zz m) (rr b a))
      (= (apply-operator ^ (apply-operator + a b) m)
         (apply-operator
          sum 0 m
          (lambda ((zz k))
            (apply-operator
             *
             (apply-operator
              *
              (apply-operator comb m k)
              (apply-operator ^ a (apply-operator sub m k)))
             (apply-operator ^ b k))))))
```

Figure 2: The Binomial Equation with Implicit Variable Sortings

conclusion is that the user-interface *improves* when the kernel of the system is built around an extremely simple and regular syntax.

The richest system, **PF**, contains nineteen operators. Of these, three are concerned with definedness, and therefore have no role in **FQ** or **ST**. Another, the definite description operator `iota`, could be introduced into **FQ** and **ST**, but its semantics are somewhat irregular in the case where the definite description is not uniquely satisfied. Hence we will include it only in **PF**. Finally, the `lambda` constructor is used to construct expressions belonging to higher types. Hence it appears in **ST** and **PF** but not in **FQ**.

The full collection of constructors is presented in Table 1. We have used the notation $\sigma(e)$ to indicate the sorting of e, when e is an expression, and $\tau(e)$ to refer to its type. Except in **PF**, $\sigma(e) = \tau(e)$.

There is considerable redundancy in this set of constructors. However, it would be inconvenient to omit any of them. Moreover, implementations should be structured so that it is easy to add constructors, as there are various other candidates that might be desirable.

3.4.1 A Type-Checking Algorithm

To summarize the discussion of the previous pages, we present an algorithm expressed in the Scheme language which, given a well-formed expression, will return its syntactic type. Given a non-well-formed expression, it will call a

Constructor	Sort of Result	Argument Restrictions
`(the-true)`	`prop`	
`(the-false)`	`prop`	
`(and p ... q)`	`prop`	
`(or p ... q)`	`prop`	
`(if-form p q r)`	`prop`	τ(p) = ... = τ(q)
`(implies p q)`	`prop`	= τ(r) = prop
`(iff p q)`	`prop`	
`(not p)`	`prop`	
`(= e1 e2)`	`prop`	τ(e1) = τ(e2)
`(forall var-list body)`	`prop`	
`(forsome var-list body)`	`prop`	τ(body) = prop
`(apply-op o a1 ... aN)`	range(σ(o))	Note 1
`(if p c a)`	lub(σ(c),σ(a))	Note 2
`(lambda var-list body)`	Note 3	
`(iota var-list body)`	$\sigma(v)$	
`(iota-p var-list body)`	$\sigma(v)$	Note 4
`(undefined var-list)`	$\sigma(v)$	
`(defined-in e1 e2)`	`prop`	τ(e1) = τ(e2)
`(is-defined e)`	`prop`	

Notes:

1. τ(o) must be a higher type. The `i`-th domain of τ(o) must equal τ(ai). The name `apply-op` shortens `apply-operator` for the sake of the margins.

2. τ(p) must be `prop`; τ(c) must equal τ(a).

3. If `var-list` declares variables named $v_1, \ldots, v_n$ associated, respectively, with the sorts $\sigma_1, \ldots, \sigma_n$, then the sorting of the result is $(\sigma_1 \ldots \sigma_n\, \sigma(\texttt{body}))$.

4. `var-list` declares a single variable of the form (v); for `iota-p`, v is required to be of kind `prop`, while for the other two constructors, it is required to be of kind `ind`.

5. The constructor `lambda` belongs only to **ST** and **PF**. The constructors `iota`, `iota-p`, `undefined`, `defined-in`, and `is-defined` belong only to **PF**.

Table 1: The Constructors of **FQ**, **ST**, and **PF**

procedure *signal-error* on a "message" explaining the problem; the value of *signal-error* can be chosen so that its result is a Scheme value distinct from any possible syntactic type. In practice, one would choose *signal-error* to be an escape procedure that would avoid any work attempting to compute the type of enclosing expressions.

We assume here that there is a procedure `language-type-constant` that, when given a language and a formal constant in that language will return the type of the constant in the language. If its second argument is not a constant in the language given in the first argument, then it returns the Scheme false value `#f`. For numerical objects (in self-extending languages), we need a procedure `type-numerical-object`. When given a language and an s-expression, it either returns the type of the numerical object that the s-expression represents (according to that language), or, if there is none, calls the *signal-error* with an appropriate argument.

We also need a procedure `sorting->type` for use in **PF**. Given a sorting, it returns the syntactic type that the sorting is included within. Naturally, it is the identity on a syntactic type.

```
(define (typecheck-top sexp language)
  (typecheck sexp language (lambda (var-name) '#f)))

(define (typecheck sexp language var-name-typer)
  (cond ((numerical-object? sexp)
         (type-numerical-object language sexp))
        ((symbol? sexp)
         (or (language-type-constant language sexp)
             (var-name-typer sexp)
             (signal-error
              (list
               sexp
               "symbol neither constant in language
                nor typable variable"))))
        ((pair? sexp)
         (typecheck-pair sexp language var-name-typer))
        (else
         (signal-error
          (list
           sexp
           "sexp neither numerical-object,
            symbol nor pair")))))
```



```
(define (typecheck-pair sexp language var-name-typer)
  (case (car sexp)
    ((the-true the-false)
     (typecheck-truth-value sexp))
    ((and or)
     (typecheck-and-or sexp language var-name-typer))
    ((if-form)
     (typecheck-fixed-length-propositional-constructor
      sexp
      language
      var-name-typer
      3))
    ((implies iff)
     (typecheck-fixed-length-propositional-constructor
      sexp
      language
      var-name-typer
      2))
    ((not)
     (typecheck-fixed-length-propositional-constructor
      sexp
      language
      var-name-typer
      1))
    ((apply-operator)
     (typecheck-apply-operator sexp language var-name-typer))
    ((=)
     (typecheck= sexp language var-name-typer))
    ((if)
     (typecheck-if sexp language var-name-typer))
    ((forall forsome)
     (typecheck-quantifier sexp language var-name-typer))
    ((lambda)
     (typecheck-lambda sexp language var-name-typer))
    ((iota)
     (typecheck-definite-descriptor sexp language
                                    var-name-typer ind))
    ((iota-p)
     (typecheck-definite-descriptor sexp language
```

```
                                        var-name-typer prop))
   ((undefined)
    (typecheck-undefined sexp language var-name-typer))
   ((defined-in)
    (typecheck-defined-in sexp language var-name-typer))
   ((is-defined)
    (typecheck-is-defined sexp language var-name-typer))
   ((with)
    (typecheck-with sexp language var-name-typer))
   (else
    (signal-error
     (list
      sexp
      "bogus constructor" (car sexp))))))
```

The individual typechecking procedures for the different contructors are defined in the appropriate places below.

3.4.2 Variable Lists and Implicit Sortings

As we pointed out at the beginning of Section 3.4, it is preferable not to represent a variable by the pair consisting of its sorting and its name at every occurrence. To do so makes formulas unnecessarily cumbersome. Hence we introduce the notion of a *variable list*; a variable list "declares" the sorting to be associated with a collection of variable names in a syntactically determined stretch.

We define variable lists in two stages:

- A *variable sublist* is an s-expression of the form:

  $$(sorting \quad v_1 \ldots v_n),$$

  where $n > 0$ and each v_i is a symbol (the name of the corresponding variable), and *sorting* is a sorting (which will be associated with the variable name in the occurrences governed by this variable list).

- A *variable list* is an s-expression of the form:

  $$(sublist_1 \ldots sublist_n),$$

  where n may be 0. It is required that no v occur twice in the sublists.

Thus, we would represent the assertion that for every scalar s and vector v, there exists a vector v' such that $s * v = v'$ in the form:

```
(forall ((scalar s)(vector v))
        (forsome ((vector v^prime)) (= (* s v) v^prime))).
```

Similarly, the assertion that for all vectors v_1 and v_2, there exists a vector v_3 such that $v_1 + v_2 = v_3$ would be written in the form:

```
(forall ((vector v_1 v_2))
        (forsome ((vector v_3)) (= (+ v_1 v_2) v_3))).
```

A variable list for **FQ** is a variable list in which the sorting associated with each variable is either a base sorting or else a list of base sortings. Nested sortings are not permitted in variable lists for **FQ**.

In the typechecking algorithm, we will need two procedures to manipulate variable lists. The first, `decode-variable-list`, takes two arguments. Suppose the first argument is a function g that, given a variable name, returns either a syntactic type or else `#f`, and the second argument is a variable list ℓ. Then `decode-variable-list` returns a function which, given a variable name, again returns either a syntactic type or else `#f`. However, if the variable name occurs in ℓ, then the syntactic type of the sorting that it is associated with in ℓ is given. If it does not appear in ℓ, then the value of g for that name is returned.

Speaking algorithmically, we define how to augment an association g according to a variable list as follows. Suppose g is an association of variable names to sortings, and `var-list` is a variable list of the form $((s_1 v_{11} \ldots v_{1n}) \ldots (s_m v_{m1} \ldots v_{mk}))$. By the assumption that a variable name occurs at most once in a variable list, we know that $v_{ij} = v_{i'j'} \Rightarrow i = i'$. Hence, we can let the augmented association g' be defined as follows:

$$g'(v) = \begin{cases} s_i & \text{if } v = v_{ij} \text{ for some } i \text{ and } j \\ g(v) & \text{if } v \neq v_{ij}\text{, and } g(v) \text{ is a type} \\ \texttt{\#f} & \text{otherwise} \end{cases}$$

In Scheme code we would express this algorithm as follows:

```
(define (decode-variable-list g var-list)
  (lambda (var-name)
    (letrec
        ((loop
```

```
        (lambda (var-list)
          (if (null? var-list)
              (g var-name)
              (let ((sub-list (car var-list)))
                (if (memq var-name (cdr sub-list))
                    (sorting->type (car sub-list) )
                    (loop (cdr var-list)))))))
    (loop var-list))))
```

A procedure `variable-list->type-list` is also needed. When given a single variable list as argument, it returns the types of the variables declared in the variable list (listed in the same order).

3.4.3 Truth and Falsehood

In order to ensure that there is a uniform way of referring to the two truth values in all languages, we introduce `the-true` and `the-false`. Because only constructors are specified by the logic—all constants belong to the individually specified language—`the-true` and `the-false` will be syntactically null-ary constructors. The result of applying them to no arguments—written `(the-true)` and `(the-false)` are formulas denoting the truth values. An implementation will normally introduce programming language constants to refer to these objects; IMPS uses `truth` and `falsehood`.

```
(define (typecheck-truth-value sexp)
  (if (null? (cdr sexp))
      prop
      (signal-error (list sexp "too many components"))))
```

3.4.4 Propositional Constructors

Six propositional operators are included, namely `and`, `or`, `if-form`, `implies`, `iff`, and `not`. In these logics, `and` and `or` are n-ary, meaning that they takes as arguments any number of formulas, and return a formula. A formula `(and p q ...)` is true if all of `p`, `q`, ... are, while `(or p q ...)` is true if at least one of them is. The constructor `if-form` is ternary; it takes a *conditional*, a *consequent*, and a *alternative*—all formulas. It returns a formula which is true if and only if either both the conditional and the consequent are true, or else the conditional is false and the alternative is true. The constructors

`implies` and `iff` are binary, and `not` is unary. They have their normal truth-functional meaning.

```
(define (typecheck-and-or sexp language var-name-typer)
  (if (every?
        (lambda (component)
          (eq?
           prop
           (typecheck component language var-name-typer)))
        (cdr sexp))
      prop
      (signal-error (list sexp "non-formula argument"))))
```

```
(define (typecheck-fixed-length-propositional-constructor
         sexp
         language
         var-name-typer
         lth)
  (if (= (length (cdr sexp)) lth)
      (if (every?
           (lambda (component)
             (eq?
              prop
              (typecheck component language var-name-typer)))
           (cdr sexp))
          prop
          (signal-error
           (list sexp
             "non-formula argument")))
      (signal-error
         (list sexp
               "wrong number of components"))))
```

3.4.5 The Apply-Operator Constructor

The apply-operator constructor is used to build a complex expression by applying an "operator"—that is, either a function or a predicator—to arguments. The well-formedness condition for the application depends only on the syntactic *types*, not the *sortings*, of the components. Thus if `op` is an expression with sorting $(s_1 \ldots s_n s_{n+1})$, and $e_1, \ldots, e_n$ are a sequence of the same number of expressions. Then (apply-operator op arg1 ... argN) is a well-formed expression of sorting range if $\tau(e_i) = \tau(s_i)$ for each i.

In **FQ** and **ST**, this is exactly the condition one would expect. In **PF** it represents a decision not to use subsorting information to determine well-formedness. Our grounds are that an expression such as gcd(3, (2.2/1.1)) should be defined (and equal to 1) even though a crude syntactic analysis gives a syntactic sorting of RR rather than ZZ for the subexpression 2.2/1.1.

```
(define (typecheck-apply-operator sexp
                                  language
                                  var-name-typer)
  (if (= (length operator-type)          ;don't forget range
         (+ (length (cddr sexp)) 1))     ;type is there!
      (let ((operator-type
             (typecheck (cadr sexp) language var-name-typer)))
        (if (every?
             (lambda (expected-type argument)
               (equal?
                expected-type
                (typecheck argument language var-name-typer)))
             (all-but-last operator-type)
             (cddr sexp)))
            (last operator-type)
            (signal-error
             (list sexp
                   "arg mismatch")))
      (signal-error
       (list sexp
             "wrong number of args"))))
```

3.4.6 Equality

Equality (written =) is a constructor taking two arguments. As with the apply-operator constructor, the condition on the arguments concerns only their types, which must agree.

```
(define (typecheck= sexp language var-name-typer)
  (if (= (length (cdr sexp)) 2)
      (if (equal?
```



```
           (typecheck (cadr sexp) language var-name-typer)
           (typecheck (caddr sexp) language var-name-typer))
          prop
          (signal-error
           (list sexp
                 "arg mismatch")))
      (signal-error
       (list sexp
             "wrong number of args"))))
```

3.4.7 If

By analogy with the propositional constructor if-form, the logic offers a constructor if, which builds conditional expressions of any type. It takes three arguments, the first of which is a formula. The second and third arguments must be of the same type. In **PF**, the sorting of the resulting expression is the least upper bound of the sortings of the second and third arguments (with respect to the partial ordering defined on page 23). In **FQ** and **ST** the type of the conditional expression is the (common) type of the second and third arguments.

The value of the expression (if p s t) is the value of s if p is true, and the value t if p is false.

```
(define (typecheck-if sexp language var-name-typer)
  (if (= (length (cdr sexp)) 3)
      (let ((test (cadr sexp))
            (conseq (caddr sexp))
            (alt (cadddr sexp)))
        (let ((result-type
               (typecheck conseq language var-name-typer)))
          (if (and
               (equal?
                prop
                (typecheck test language var-name-typer))
               (equal?
                result-type
                (typecheck alt language var-name-typer)))
              result-type
              (signal-error
```

```
          (list
           sexp
           "mistyped components"))))
    (signal-error
     (list sexp
           "wrong number of components")))))
```

3.4.8 Variable-Binding Constructors

The variable-binding constructors common to the three logics are **forall** and **forsome**. Each of them requires a list of variables and a formula. These are called, respectively, the newly bound variables and the body of the expression. The resulting expression is a formula. In **FQ**, it is required that the newly bound variables must have base sortings.

The resulting formulas are true if for every [respectively, at least one] assignment of values to the newly bound variables, the body is true.

```
(define (typecheck-quantifier sexp language var-name-typer)
  (if (= (length (cdr sexp)) 2)
      (let ((var-list (cadr sexp))
            (body       (caddr sexp)))
        (if (var-list? var-list)
            (if (eq? prop
                     (typecheck
                      body
                      language
                      (decode-variable-list
                       var-name-typer
                       var-list)))
                prop
                (signal-error
                 (list
                  sexp
                  "non-prop body
                      to quantified expression")))
            (signal-error
             (list
              sexp
              "bad variable-list"))))
```



```
      (signal-error
       (list sexp
             "wrong number of components"))))
```

ST and **PF** also contain the constructor **lambda**. It requires a variable-list and an expression, which may have any sorting s_0. If all of the variables in the variable-list are $v_1, \ldots, v_n$, and they appear in that order, and with sortings $s_1, \ldots, s_n$, then the sorting of the resulting expression is $(s_1 \ldots s_n s_0)$. The resulting expression denotes the function whose value for the argument list $a_1, \ldots, a_n$ is the denotation of the body, under the assumption that the newly bound variables have the values the values $a_1, \ldots, a_n$ respectively.

```
(define (typecheck-lambda sexp language var-name-typer)
  (if (= (length (cdr sexp)) 2)
      (let ((var-list (cadr sexp))
            (body       (caddr sexp)))
        (if (var-list? var-list)
            (append
             (variable-list->type-list var-list)
             (list
              (typecheck
               body
               language
               (decode-variable-list
                var-name-typer
                var-list))))
            (signal-error
             (list
              sexp
              "bad variable-list"))))
      (signal-error
       (list sexp
             "wrong number of components"))))
```

PF contains the definite description operators **iota** and **iota-p**. These bind a single variable, which must be of kind **ind** in the first case and of kind **prop** in the second. An expression built using **iota** is undefined if its body is not uniquely satisfied. An expression built using **iota-p** cannot be, bacause it is of kind prop. We stipulate that its value is "false-like" in the sense defined in Section 4.4.

```
(define (typecheck-definite-descriptor sexp
                                       language
                                       var-name-typer
                                       kind)
  (if (= (length (cdr sexp)) 2)
      (let ((var-list (cadr sexp))
            (body     (caddr sexp)))
        (if (var-list? var-list)
            (let ((type-list
                   (variable-list->type-list var-list)))
              (if (= (length type-list) 1)
                  (if (kind-matches? kind (car type-list))
                      (car type-list)
                      (signal-error
                       (list sexp
                             "result of wrong kind")))
                  (signal-error
                   (list sexp
                         "wrong number of variables"))))
            (signal-error
             (list
              sexp
              "bad variable-list"))))
      (signal-error
       (list sexp
             "wrong number of components"))))

(define (kind-matches? goal-type result-type)
  (if (kind-ind? goal-type)
      (kind-ind? result-type)
      (kind-prop? result-type)))
```

3.4.9 Constructors Concerning Definedness

Three constructors deal with definedness, and occur only in **PF**. They are `is-defined`, `undefined`, and `defined-in`.

The first, `is-defined`, takes a single term or function as argument. The result, a formula, is true just in case the expression given as argument has a value.

```
(define (typecheck-is-defined sexp language var-name-typer)
  (if (= (length (cdr sexp)) 1)
      (let ((ignore
             (typecheck (cadr sexp) language var-name-typer)))
        prop)
      (signal-error
       (list sexp
             "wrong number of components"))))
```

The second, `undefined`, is a variable-binding constructor. As such, it takes a list of variables containing exactly one variable, but this followed by no term or function as its body. The identity of the variable is irrelevant, as the variable occurs only to indicate the relevant sorting.[5] The resulting expression has no free variables, and its variable of quantification is its sole bound variable. It has the syntactic sorting of the variable, but its value is undefined. This constructor is useful in the course of simplification. For instance, suppose that t is some complex expression of sorting RR, and it is known to be defined. We may then want to reduce the quotient t/t on the assumption that t's value is different from 0. The `undefined` constructor allows us to replace t/t with the expression:

```
(if (not (= t 0))
    1
    (undefined ([rr x]))).
```

It can also be used to state—say, in the definition of a function—that the function will be undefined whenever some condition is not met.

```
(define (typecheck-undefined sexp language var-name-typer)
  (if (= (length (cdr sexp)) 2)
      (let ((var-list (cadr sexp)))
        (if (var-list? var-list)
```

[5] The reason we do not simply use the sort itself in the formula is that sorts are not first-class objects in these logics. There are no variables over sorts; sorts cannot be quantified; there are no functions from sorts to sorts. Hence, we stick to the convention that sorts also cannot be constituents of expressions.

```
(let ((type-list
       (variable-list->type-list var-list)))
  (if (= (length type-list) 1)
      (if (kind-ind? (car type-list))
          (car type-list)
          (signal-error
           (list sexp
                 "wrong kind variable (ind)")))
      (signal-error
       (list sexp
             "wrong number of variables"))))
(signal-error
 (list
  sexp
  "bad variable-list"))))
(signal-error
 (list sexp
       "wrong number of components"))))
```

The third constructor, `defined-in`, takes as arguments two expressions that must agree in type. The identity of the second expression is irrelevant, and only its syntactic sorting matters. The constructor returns a formula that is true just in case its first argument is defined and has a value in the sort denoted by the sorting of its second argument. In IMPS it is customary to apply this constructor with second argument of the form `(undefined ([s x]))`, because the latter is an expression of sorting `s` that is conveniently constructed and independent of the constants available in any particular formal theory.

```
(define (typecheck-defined-in sexp language var-name-typer)
  (if (= (length (cdr sexp)) 2)
      (if
       (eq?
        (typecheck (cadr sexp) language var-name-typer)
        (typecheck (caddr sexp) language var-name-typer))
       prop
       (signal-error (list sexp "arg mismatch")))
      (signal-error
       (list sexp
             "wrong number of components"))))
```


3.4.10 The *With* Constructor

We use variable-lists, as described above, to indicate the sortings of free occurrences of variables as well as bound occurrences of them. In order to have available an appropriate variable list for a *free* occurrence of a variable, we introduce a constructor, `with`, which takes the same form as the variable-binding operators. However, it does not bind occurrences, nor does it have any truth-conditional meaning. Its only significance is that it makes the sortings of variables explicit. Thus, for instance, in the following expression, which represents the induction schema:

```
(with (((nn prop) p))
      (implies (and (p 0)
                    (forall ((nn n))
                            (implies (p n) (p (1+ n)))))
               (forall ((nn n)) (p n)))),
```

`p` is a free variable ranging over sets of natural numers (or, more precisely, unary predicates of objects of the sort denoted by `nn`). This is of course precisely the significance attached to the expression:

```
(implies (and ([(nn prop) p] 0)
              (forall ((nn n))
                      (implies ([(nn prop) p] n)
                               ([(nn prop) p] (1+ n)))))
         (forall ((nn n)) ([(nn prop) p] n))),
```

which simply makes explicit that every occurrence of `p` is to be associated with the sorting `(nn prop)`. Indeed, this is the only significance of `with`.

We emphasize that `with` is not a variable-binding constructor. Occurrences of the variable names it governs are free if it is the outermost constructor in an expression. Moreover, it does not introduce a scope in any logically meaningful sense. For instance, in:

```
(or (with ((prop p)) p)
    (with ((prop p)) (not p))),
```

the variables in the two disjuncts are identical, and it would be unsound to rename the variable `p` in one conjunct but not the other.

```
(define (typecheck-with sexp language var-name-typer)
  (if (= (length (cdr sexp)) 2)
```

```
(let ((var-list (cadr sexp)))
  (if (var-list? var-list)
      (typecheck
       body
       language
       (decode-variable-list var-name-typer var-list))
      (signal-error
       (list sexp "bad variable-list"))))
(signal-error
 (list sexp "wrong number of components"))))
```

3.4.11 Eliminating *With* and Variable Lists

We give next an algorithm for the process of expanding variable names. At any stage in the expansion process, there is a current s-expression e, and an association g between variable names and sortings. Initially g is the null function; that is, no variable name is associated with a sorting. Every time that the algorithm traverses a constructor using a variable list, it *augments* the association g by decoding the variable list in the sense defined in Section 3.4.2.

1. If e is a variable name v, and it has sorting s in g, then return $[s\ v]$.

2. If e is a variable name v, but v has no associated sorting in g, return v.

3. If e is a constant, return e.

4. Suppose e is of the form `(constr comp1 ... compN)`, where `constr` is not `with` or a variable-binding constructor. Let $\overline{\texttt{comp1}}, \ldots, \overline{\texttt{compN}}$ be the results of (recursively) executing the expansion process on `comp1`, ..., `compN` respectively, with the association g. Then return `(constr` $\overline{\texttt{comp1}}$ `...` $\overline{\texttt{compN}}$`)`.

5. Suppose e is of the form `(with var-list body)`. Let g' be the result of augmenting g by decoding the variable/sorting associations in `var-list`. Return the result of (recursively) executing the expansion process on `body` and g'.

6. Suppose e is of the form `(constr var-list body)`, where `constr` is a variable-binding constructor.

   (a) Let g' be the result of augmenting g with the variable/sorting associations in `var-list`.

   (b) Let $\overline{\texttt{body}}$ is the result of (recursively) executing the expansion process on `body` and g'.

   (c) Let $\vec{V}$ be the list of items of the form $[s_i\ v_i]$ where the v_i are the variable names occurring in `var-list` taken in the same order, and the s_i are the associated sortings.

   (d) Return `(constr` $\vec{V}$ $\overline{\texttt{body}}$`)`.

If e is an expression in the concrete syntax, we say that the *sorting of an occurrence* of a variable name v in e is s if s is the sorting associated with that variable name in the variable list of the smallest enclosing variable-binding constructor or `with` constructor whose variable list contains v. Otherwise, we say that occurrence has undetermined sorting. An expression in the concrete syntax is *readable* if no variable occurrence has undetermined sorting. Clearly, e is readable if and only if case 2 in the expansion algorithm never occurs. Hence an expression is readable if and only if, in the result of the expansion process, every variable name has been replaced by a pair $[s\ v]$.

The `with` constructor is essentially a tool for allowing us to transform any expression into a readable expression without changing its intended meaning; all that is done is to make explicit the intended sortings of the variable name occurrences in it. As will be seen in Section 4, no semantics are assigned to the `with` constructor, because the semantics are assigned to the abstract syntax in which every occurrence of a variable name is immediately associated with a sorting.

Section 4
Formal Semantics

Suppose that $\mathcal{L}$ is a language whose set of type and sort symbols (exclusive of prop) is $S = \{s_1, \ldots, s_n\}$, and whose constants form a set $\mathcal{C}$. We assume that for any $c \in \mathcal{C}$, the sorting of c is given by a function σ; that is, if $c \in \mathcal{C}$, then $\sigma(c)$ is the sorting of c in $\mathcal{L}$.

For the purposes of this section, we finesse the issue of self-extending languages: $\mathcal{C}$ may be infinite, and we assume that all potential new constants are already included in it, with the appropriate sortings. Note, however, that only finitely many symbols are in $\mathcal{C}$, the remaining members being strings that are treated (or "read") not as symbols but as numerical objects. We also assume that all "possible" variables are in use, in the sense that we will take a variable to be a pair consisting of any symbol not naming a constant, together with a sorting. As there are infinitely many symbols (of unrestricted length), this is consonant with the normal approach to logic, which requires that there be infinitely many variables of each sorting. We will refer to the set of variables as $\mathcal{V}$. As $\mathcal{V}$ is disjoint from $\mathcal{C}$, we may assume that, for $v \in \mathcal{V}$, $\sigma(v)$ returns the second component of v.

First consider the logics **FQ** and **ST**. If $\mathcal{L}$ is as described above, then a *frame* for $\mathcal{L}$ is a type structure together with a map from S to the basic types of individuals. We can write a frame for these logics as an indexed family of the non-empty sets of individuals. If $\mathcal{F} = \{F_s : s \in S\}$ is a frame for $\mathcal{L}$, then the *interpretation* of a sorting s in $\mathcal{F}$, which we will write $I_{\mathcal{F}}s$, is defined by induction on the structure of s:

- $I_{\mathcal{F}}(s) = F_s$ for a base sorting $s \in S$.
- $I_{\mathcal{F}}(\texttt{prop}) = \{T, F\}$.
- If s is a higher sorting $(s_1 \ \ldots \ s_m \ s_{m+1})$, then $I_{\mathcal{F}}(s)$ is the set of all total m-ary functions $f : I_{\mathcal{F}}(s_1) \times \ldots \times I_{\mathcal{F}}(s_m) \to I_{\mathcal{F}}(s_{m+1})$.

In **FQ** this inductive definition could be "clipped off" after the first induction step, because every variable or constant has a sorting that is either a base sorting or else a list of base sortings. In **ST**, all the levels are used. We do not require that the base sortings for a frame in these logics be disjoint. That would be unnecessary, because the syntax of the logics prevents us

from expressing any formula that would be sensitive to whether sorts are disjoint or not. We will make this claim more precise below in Section 4.5.1.

Turning next to **PF**, we again let $\mathcal{L}$ be a language with type symbols (exclusive of prop) $S = \{s_1, \ldots, s_n\}$, and named sorts $S' = \{s_{n+1}, \ldots, s_k\}$. A *frame* for $\mathcal{L}$ is a family $\mathcal{F} = \{F_s : s \in S \cup S'\}$ of non-empty sets indexed by $S \cup S'$. $I_{\mathcal{F}}(s)$ is again defined by induction on the structure of s.

- $I_{\mathcal{F}}(s) = F_s$ for a type symbol or named sort $s \in S \cup S'$.
- $I_{\mathcal{F}}(\texttt{prop}) = \{T, F\}$.
- Suppose s is a higher sorting $(s_1 \ \ldots \ s_m \ s_{m+1})$, and s_{m+1} is not prop-sorted. Then $I_{\mathcal{F}}(s)$ is the set of all partial (and total) m-ary functions:
$$f : I_{\mathcal{F}}(s_1) \times \ldots \times I_{\mathcal{F}}(s_m) \rightharpoonup I_{\mathcal{F}}(s_{m+1}).$$
- Suppose s is a higher sorting $(s_1 \ \ldots \ s_m \ s_{m+1})$, and s_{m+1} is prop-sorted. Then $I_{\mathcal{F}}(s)$ is the set of all total m-ary functions:
$$f : I_{\mathcal{F}}(s_1) \times \ldots \times I_{\mathcal{F}}(s_m) \to I_{\mathcal{F}}(s_{m+1}).$$

It is required here that if s_i is a named sort with some sorting s as its enclosing sort, then $I_{\mathcal{F}}(s_i) \subseteq I_{\mathcal{F}}(s)$. From this it follows that if t is the syntactic type of any sorting s, then $I_{\mathcal{F}}(s) \subseteq I_{\mathcal{F}}(t)$.

4.1 Structures

A structure for the language $\mathcal{L}$ consists of a frame for the language, which serves to interpret the *sortings* of $\mathcal{L}$, together with an association between the *constants* of $\mathcal{L}$ and objects in the frame. The definition is uniform, and does not need to be stated independently for **PF**. However, a structure for $\mathcal{L}$ incorporates a frame for $\mathcal{L}$, so there is a hidden dependency on the choice of logical system.

When $\mathcal{L}$ is a language as described at the beginning of this section, we define an $\mathcal{L}$-structure $\mathcal{A} = \langle \mathcal{F}, I_{\mathcal{C}} \rangle$ to be a frame for $\mathcal{L}$ together with a function defined on C. The only requirement on $\mathcal{A}$ is that $I_{\mathcal{C}}$ take values in the range of $I_{\mathcal{F}}$ in a way consistent with $I_{\mathcal{F}}$'s treatment of sortings. In particular, for all $c \in C$, $I_{\mathcal{C}}(c) \in I_{\mathcal{F}}(\sigma(c))$. If the underlying logic is **PF**, then the values of $I_{\mathcal{C}}$ may be partial functions; however, $I_{\mathcal{C}}$ is not a partial function, because $I_{\mathcal{C}}(c)$ is always some object in the frame $\mathcal{F}$.

We will also need the notion of a variable assignment, which is a total function α mapping the variables $\mathcal{V}$ into $\mathcal{F}$ such that $\alpha(v) \in I_{\mathcal{F}}(\sigma(v))$. We will use Greek letters from the beginning of the alphabet to refer to variable assignments. If $V \subset \mathcal{V}$, then we use the relation $\alpha{\sim_V}\beta$ to mean that $v \notin V$ implies $\alpha(v) = \beta(v)$. If $v \in \mathcal{V}$, then $\alpha{\sim_v}\beta$ means $\alpha{\sim_{\{v\}}}\beta$.

In the next three subsections, we will give the clauses in the definitions of denotation and satisfaction that apply to the three logics. For a given $\mathcal{A}$ and α, this means extending $I_{\mathcal{C}}$ to a function I that is applicable to all expressions. In **PF**, this function is not a total function, but is defined for a particular expression just in case that expression has a denotation. However, I is always defined for variables, constants, and expressions of kind `prop`.

We will follow standard terminology in saying that a formula ϕ is *valid in a structure* $\mathcal{A}$ (written $\mathcal{A} \models \phi$) if, for every variable assignment α, $I_{\mathcal{A}}(\alpha, \phi) = T$. The formula ϕ is *valid* (written $\models \phi$) if it is valid in every structure $\mathcal{A}$. A structure *satisfies a theory* Γ (written $\mathcal{A} \models \Gamma$) if every formula (axiom) of the theory is valid in the structure. If Γ is a set of formulas and ϕ is a formula, then we say that ϕ is a *semantic consequence* of Γ just in case, for all $\mathcal{A}$, $\mathcal{A} \models \Gamma$ implies $\mathcal{A} \models \phi$.

We must also ensure that there is a sorting associated with every occurrence of a variable name in an expression to be interpreted by the function I. This would be problematic if were we genuinely interested in expressions in the concrete syntax that are not *readable* in the sense given in Section 3.4.11. However, our real concern is only with expressions where no variable occurrence has undetermined sorting. To simplify the semantic definition, we will work directly with expressions in the abstract syntax described in Section 3. Thus we shall assume that there are no *with*-constructors in an expression to which a denotation is being ascribed. In addition, in the first position after a variable-binding constructor, we will always find a list of pairs of the form $[s\ v]$. Finally, each variable occurrence will have this explicit form. We shall call a pair $[s\ v]$ a "decorated" variable.

Thus, we regard variable lists, with their specification of sorting, and `with` constructors, as being "mere syntax," to be removed before formal semantics are given for a more abstract syntax.

4.2 Denotation and Satisfaction for FQ

Suppose that $\mathcal{L}$ is a language for **FQ**, $\mathcal{A}$ is an $\mathcal{L}$-structure, and α is a variable assignment. Let σ be extended so that for any expression e in $\mathcal{L}$, $\sigma(e)$ is the sorting of e in $\mathcal{L}$. We proceed to define a function $I(\alpha, e)$ extending $I_{\mathcal{C}}$ and

α to all expressions e, in such a way that $I(\alpha, e) \in I_{\mathcal{F}}(\sigma(e))$. $I(\alpha, e)$ defines the denotation of e (or truth value of e if e is a formula), relative to $\mathcal{A}$. The definition is an induction on the structure of the expression e. The relation $\sim_V$ is used to handle variable-binding operators; this was an innovation of Tarski's.

The interpretation of variables and constants is determined directly from α and $\mathcal{A}$. If $[v\ s]$ is an occurrence of the variable named v with associated sorting s, then $I(\alpha, [v\ s]) = \alpha([v\ s])$. Similarly, if $c \in \mathcal{C}$ is a constant of $\mathcal{L}$, then $I(\alpha, c) = I_{\mathcal{C}}(c)$.

4.2.1 Truth, Falsehood, and the Propositional Constructors

the-true: $I(\alpha, \texttt{(the-true)}) = T$.

the-false: $I(\alpha, \texttt{(the-false)}) = F$.

and: $I(\alpha, \texttt{(and p ... q)}) = T$ if, for every formula ϕ among `p ... q`, $I(\alpha, \phi) = T$. Otherwise, its value is F. In particular, $I(\alpha, \texttt{(and)}) = T$.

or: $I(\alpha, \texttt{(or p ... q)}) = T$ if there is at least one formula ϕ among `p ... q` such that $I(\alpha, \phi) = T$. Otherwise, its value is F. In particular, $I(\alpha, \texttt{(or)}) = F$.

implies: $I(\alpha, \texttt{(implies p q)}) = T$ if either $I(\alpha, \texttt{p}) = F$ or $I(\alpha, \texttt{q}) = T$. Otherwise, its value is F.

iff: $I(\alpha, \texttt{(iff p q)}) = T$ if $I(\alpha, \texttt{p}) = I(\alpha, \texttt{q})$. Otherwise, its value is F.

not: $I(\alpha, \texttt{(not p)}) = T$ if $I(\alpha, \texttt{p}) = F$. Otherwise, its value is F.

if-form:

$$I(\alpha, \texttt{(if-form p q r)}) = \begin{cases} I(\alpha, \texttt{q}) & \text{if } I(\alpha, \texttt{p}) = T \\ I(\alpha, \texttt{r}) & \text{otherwise.} \end{cases}$$

4.2.2 Apply-Operator, Equality, and If

apply-operator: $I(\alpha, \texttt{(apply-operator op a1 ... aN)})$

$$= [I(\alpha, \texttt{op})](I(\alpha, \texttt{a1}), \ldots, I(\alpha, \texttt{aN})).$$

equality: $I(\alpha, \texttt{(= s t)}) = T$ if $I(\alpha, \texttt{s}) = I(\alpha, \texttt{t})$. Otherwise, its value is F.

if:

$$I(\alpha, \texttt{(if p s t)}) = \begin{cases} I(\alpha, \texttt{s}) & \text{if } I(\alpha, \texttt{p}) = T \\ I(\alpha, \texttt{t}) & \text{otherwise} \end{cases}$$

One comment is in order about **apply-operator**. Suppose an expression e is of the form (op a1 ... aN), where op is an operator of sorting $(s_1 \ldots s_N s_0)$, and a1, ..., aN are expressions of sortings $s_1, \ldots, s_N$, respectively. Then, inductively, we may suppose that $I(\alpha, \texttt{op}) \in I_{\mathcal{F}}[(s_1 \ldots s_N s_0)]$ and, for each i from 1 to N, $I(\alpha, \texttt{a}i) \in I_{\mathcal{F}}[s_i]$. Hence, the denotation of e, $[I(\alpha, \texttt{op})](I(\alpha, \texttt{a1}), \ldots, I(\alpha, \texttt{aN}))$, belongs to the required sort, namely $I_{\mathcal{F}}[s_0]$.

4.2.3 Variable-Binding Constructors

The clauses for forall and forsome are quite straightforward. In the abstract syntax, we may suppose that an expression e having one of these as its primary constructor has the form:

$$(\text{constr } V\ \phi),$$

where V is a list of decorated variables, and ϕ is the body of the expression.

forall: $I(\alpha, (\texttt{forall}\ V\ \phi)) = T$ if, for every variable assignment β such that $\beta \sim_V \alpha$, $I(\beta, \phi) = T$. Otherwise, its value is F.

forsome: $I(\alpha, (\texttt{forsome}\ V\ \phi)) = T$ if there exists a variable assignment β such that $\beta \sim_V \alpha$ and $I(\beta, \phi) = T$. Otherwise, its value is F.

4.3 ST: The Lambda Constructor

If $\mathcal{L}$ is a language for **ST** rather than for **FQ**, then there is one addition that must be made. We must give a clause defining the semantics for the constructor lambda, which does not occur in **FQ**. However, no changes to the wording of the other clauses are needed. Naturally, the clauses that are common to **FQ** and **ST** cover a much broader class of expressions in **ST**; nevertheless, the logical content expressed by the constructors is identical between the two systems.

In the abstract syntax, we may suppose that an expression e having lambda as its primary constructor has the form:

$$(\texttt{lambda}\ (v_1 \ldots v_n)\ e'),$$

where $(v_1 \ldots v_n)$ is a list of (distinct) decorated variables, and e' is the body of the expression. If each v_i has sorting s_i, and if the sorting of e' is s_0, then the sorting of e is $(s_1 \ldots s_n s_0)$, and its interpretation must be an n-ary function f of sort:

$$f : I_{\mathcal{F}}(s_1) \times \ldots I_{\mathcal{F}}(s_n) \to I_{\mathcal{F}}(s_0).$$

We define $I(\alpha, e)$ to be a function of that sort as follows. Let $a = \langle a_1, \ldots, a_n \rangle$ be an n-tuple of arguments of appropriate sorts, and let β_a be the variable assignment such that $\beta_a \sim_{\{v_1, \ldots, v_n\}} \alpha$ and $\beta_a(v_i) = a_i$.

lambda: $I(\alpha, \texttt{lambda}\ (v_1 \ldots v_n)\ e'))[a] = I(\beta_a, e')$.

By the inductive structure of the definition of I, this is always defined and of sort $I_{\mathcal{F}}(s_0)$.

4.4 Denotation and Satisfaction for PF

Suppose now that $\mathcal{L}$ is a language for **PF**, $\mathcal{A}$ is an $\mathcal{L}$-structure, and α is a variable assignment. Let σ be extended so that for any e in $\mathcal{L}$, $\sigma(e)$ is the sorting of e in $\mathcal{L}$. We again inductively define a function $I(\alpha, e)$ extending $I_{\mathcal{C}}$ and α to include complex expressions e, in such a way that $I(\alpha, e) \in I_{\mathcal{F}}(\sigma(e))$ whenever the former is defined. $I(\alpha, e)$ gives the denotation of e (or truth value of e if e is a formula), relative to $\mathcal{A}$. Thus, I, regarded as a function of α and e, may not be a total function. As long as e is not a prop-sorted expression, there is no need for $I(\alpha, e)$ to yield a value. Indeed, if e results from applying an operator of kind ind to arguments, and the value of the operator is a partial function, then $I(\alpha, e)$ will be undefined if the arguments are not in the domain of definition. Similarly, the constructor iota may cause $I(\alpha, e)$ to be undefined.

In **PF**, an atomic formula is false if any of its immediate components is undefined. This requirement (in combination with other basic principles like β-reduction) leads to a corresponding condition on higher type, prop-sorted expressions. For instance, consider the complex predicate:

$$\lambda x : \mathbf{R} \,.\, x \leq 3/0.$$

The result of applying this to any argument t must be equivalent to $t \leq 3/0$, which is false. Thus, the value of $\lambda x : \mathbf{R} \,.\, x \leq 3/0$ is equal to the constant function $\lambda y : \mathbf{R} \,.\, F$.

A similar phenomenon occurs at higher types. If we abstract the relation $\leq$ from the previous example, we get the expression:

$$\lambda f : (\mathbf{R}\,\mathbf{R}) \,.\, \lambda x : \mathbf{R} \,.\, x \leq 3/0.$$

This expression must have the same value as the higher typed constant function:

$$\lambda f : (\mathbf{R}\,\mathbf{R}) \,.\, \lambda x : \mathbf{R} \,.\, F.$$

These examples motivate the idea of the *false-like* object of any propositional sort in a frame $\mathcal{F}$, defined inductively. The false-like object belonging to the sort of propositions is F. If s is a propositional sort with the false-like object F_s, then the false-like object of sort $s' = (s_1 \ldots s_n s)$ is the function:

$$F_{s'} : s_1 \times \cdots \times s_n \rightarrow s$$

which takes the value F_s for all tuples of arguments. We will also define a set of *expressions* of **PF** which we will also write F_s, where s is a prop-sorting; we use the inductive stipulations:

- $F_{\mathrm{prop}} =$ `(the-false)`.
- $F_{s_1 \ldots s_n s_0} = \lambda v_1 : s_1 \ldots v_n : s_n \,.\, F_{s_0}$, when s_0 is prop-sorted.

The clauses for truth, falsehood, and the propositional constructors are identical in wording to those given in Section 4.2.1.

4.4.1 Apply-Operator, Equality, and If

Suppose that e is of the form `(op a1 ... aN)`, where op is an operator of sorting $(s_1 \ldots s_N\ s_0)$; suppose `a1`, ..., `aN` are expressions, where $\tau(s_i) = \tau($`ai`. $I(\alpha, \mathrm{op})$ will yield a value for these arguments only if, for each i, $I(\alpha, \mathrm{a}i)$ is defined and belongs to $I_{\mathcal{F}}(s_i)$. The behavior of I depends on whether op is prop-sorted. With this in mind, we will divide the clause for `apply-operator` into two main cases, each with two subcases:

apply-operator: 1. op is prop-sorted:

(a) Suppose each $a_i = I(\alpha, \mathrm{a}i)$ is defined and belongs to sort s_i. Then:

$$I(\alpha, (\texttt{op a1 ... aN})) = (I(\alpha, \mathrm{op}))(a_1, \ldots, a_N).$$

(b) Otherwise, $I(\alpha, (\texttt{op a1 ... aN})) = F_{s_0}$.

2. op is not prop-sorted:

(a) Suppose each $a_i = I(\alpha, \mathrm{a}i)$ is defined and belongs to sort s_i, and suppose that $I(\alpha, \mathrm{op})$ is defined and yields a value for $a_1, \ldots, a_N$. Then:

$$I(\alpha, (\texttt{op a1 ... aN})) = (I(\alpha, \mathrm{op}))(a_1, \ldots, a_N).$$

(b) Otherwise, $I(\alpha, (\texttt{op a1 ... aN}))$ is undefined.

equality: $I(\alpha, (\texttt{= s t})) = T$ if $I(\alpha, \texttt{s})$ and $I(\alpha, \texttt{t})$ are both defined, and they have the same value. Otherwise, its value is F.

if: $I(\alpha, (\texttt{if p s t}))$

$$= \begin{cases} I(\alpha, \texttt{s}) & \text{if } I(\alpha, \texttt{p}) = T \\ I(\alpha, \texttt{t}) & \text{if } I(\alpha, \texttt{p}) = F \end{cases}$$

4.4.2 Variable-Binding Constructors

We consider next `forall`, `forsome`, `iota`, and `iota-p`, and will turn to `lambda` afterwards. The clauses for `forall` and `forsome` are identical to those in Section 4.2.

In the abstract syntax, an expression e having one of `forall`, `forsome`, and `iota` as its primary constructor has the form:

$$(\text{constr } V\ \phi),$$

where V is a list of decorated variables, and ϕ is the body of the expression. If *constr* is `iota`, then V is a singleton (v).

forall: $I(\alpha, (\texttt{forall}\ V\ \phi)) = T$ if, for every variable assignment β such that $\beta \sim_V \alpha$, $I(\beta, \phi) = T$. Otherwise, its value is F.

forsome: $I(\alpha, (\texttt{forsome}\ V\ \phi)) = T$ if there exists a variable assignment β such that $\beta \sim_V \alpha$ and $I(\beta, \phi) = T$. Otherwise, its value is F.

iota: If there exists a unique variable assignment β such that $\beta \sim_v \alpha$ and $I(\beta, \phi) = T$, then:

$$I(\alpha, (\texttt{iota}\ (v)\ \phi)) = \beta(v).$$

Otherwise $I(\alpha, (\texttt{iota}\ (v)\ \phi))$ is undefined.

iota-p: If there exists a unique variable assignment β such that $\beta \sim_v \alpha$ and $I(\beta, \phi) = T$, then:

$$I(\alpha, (\texttt{iota}\ (v)\ \phi)) = \beta(v).$$

Otherwise, if $s = I_{\mathcal{F}}(\sigma(v))$, and F_s is the false-like object of the appropriate sort, then:

$$I(\alpha, (\texttt{iota}\ (v)\ \phi)) = F_s.$$

Suppose now that expression e has **lambda** as its primary constructor, and is thus of the form:

$$(\texttt{lambda}\ (v_1 \ldots v_n)\ e'),$$

where $(v_1 \ldots v_n)$ is a list of (distinct) decorated variables, and e' is the body of the expression. If each v_i has sorting s_i, and if the sorting of e' is s_0, then the sorting of e is $(s_1 \ldots s_n s_0)$.

The interpretation of e is an n-ary function f of sort:

$$f : I_{\mathcal{F}}(s_1) \times \ldots I_{\mathcal{F}}(s_n) \rightharpoonup I_{\mathcal{F}}(s_0),$$

where f may be partial if s_0 is of kind `ind`. Again let $a = \langle a_1, \ldots, a_n \rangle$ be an n-tuple of arguments of appropriate sorts, and let β_a be the variable assignment such that $\beta_a \sim_{\{v_1,\ldots,v_n\}} \alpha$ and $\beta_a(v_i) = a_i$.

lambda: $I(\alpha, e)[a] =$

$$\begin{cases} I(\beta_a, e') & \text{if } I(\beta_a, e') \text{ is defined} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

Note that if e' is prop-sorted, $I(\beta, e')$ is defined (for all β). Thus the second case never occurs, and the denotation of the λ-expression is a total function, as it should be.

4.4.3 Constructors Concerning Definedness

undefined: $I(\alpha, (\texttt{undefined (v)}))$, is, naturally, undefined. Recall that this expression is well-formed only if v is of kind `ind`.

is-defined: $I(\alpha, (\texttt{is-defined}\ e)) = T$ if $I(\alpha, e)$ is defined, and otherwise is F.

defined-in: $I(\alpha, (\texttt{defined-in}\ e_1\ e_2)) = T$ if $I(\alpha, e_1)$ is defined and belongs to $I_{\mathcal{F}}(\sigma(e_2))$, and is otherwise F.

4.5 Comments

4.5.1 Overlapping Sorts

Although, as we described **FQ** and **ST**, overlapping sorts are not envisaged, we have nevertheless allowed structures for these logics to use frames where sorts do have non-null intersections. The reason for this is simple: the syntax of the logics ensures that overlapping sorts have no effect on the truth or falsehood of sentences.

More precisely, suppose that $\mathcal{A} = \langle \mathcal{F}, I_{\mathcal{C}} \rangle$, and there are two distinct sortings that have overlapping interpretations in $\mathcal{F}$. Let $\mathcal{F}'$ differ from $\mathcal{F}$ in that, for any sort symbol s,

$$I_{\mathcal{F}'}(s) = \{\langle x, s \rangle : x \in I_{\mathcal{F}}(s)\}.$$

Now we define a function π from $\mathcal{F}'$ to $\mathcal{F}$ by induction on the structure of sortings. If $\sigma(x)$ is a base sorting s, then $x = \langle y, s \rangle$, and we define $\pi(x) = y$. As for `prop`, let $\pi(T) = T$ and $\pi(F) = F$. Otherwise, $\sigma(x)$ is of some higher sorting $s_1 \ldots s_n s_0$. Define $\pi(x)$ to be the function that, when applied to arguments $\pi(a_1), \ldots, \pi(a_n)$, returns $\pi(x(a_1, \ldots, a_n))$. This stipulation defines a unique total function, because π is a bijection between base sortings $I_{\mathcal{F}'}(s)$ and $I_{\mathcal{F}}(s)$, and, moreover, th property of being a bijection between corresponding sorts is preserved as we ascend the hierarchy of sorts.

Let $\mathcal{A}' = \langle \mathcal{F}', I'_{\mathcal{C}} \rangle$, where $I'_{\mathcal{C}} = \pi^{-1} \circ I_{\mathcal{C}}$. Now clearly $\mathcal{A}'$ has no overlapping sorts.

Moreover, we can see that $I_{\mathcal{A}}(\pi \circ \alpha, e) = \pi(I_{\mathcal{A}'}(\alpha, e))$. It is clear from the definitions of π and $\mathcal{A}'$ that this property holds when $e \in \mathcal{C}$, and it is a triviality when $e \in \mathcal{V}$. Thus, we need to verify that it is preserved by each semantic clause. The most interesting clauses are those for `equality` and `apply-operator`.

If e is of the form $t_1 = t_2$ in **FQ** or **PF**, then t_1 and t_2 are expressions having some common sorting s. By induction, we may assume that

$$I_{\mathcal{A}}(\pi \circ \alpha, t_1) = \pi(I_{\mathcal{A}'}(\alpha, t_1))$$

and

$$I_{\mathcal{A}}(\pi \circ \alpha, t_2) = \pi(I_{\mathcal{A}'}(\alpha, t_2)).$$

Hence,

$$[I_{\mathcal{A}}(\pi \circ \alpha, t_1) = I_{\mathcal{A}}(\pi \circ \alpha, t_2)] \equiv [\pi(I_{\mathcal{A}'}(\alpha, t_1)) = \pi(I_{\mathcal{A}'}(\alpha, t_2))].$$

Moreover, as π is a bijection between $I_{\mathcal{F}'}(s)$ and $I_{\mathcal{F}}(s)$:

$$[I_{\mathcal{A}}(\pi \circ \alpha, t_1) = I_{\mathcal{A}}(\pi \circ \alpha, t_2)] \equiv [I_{\mathcal{A}'}(\alpha, t_1) = I_{\mathcal{A}'}(\alpha, t_2)].$$

Note that this line of argument breaks down in **PF**, where t_1 and t_2 may be of different syntactic sortings. This prevents us from "removing the πs" from the right hand side of the last formula.

The situation with `apply-operator` is similar. If e is the application of an operator to arguments, then the operator has some sorting $(s_1 \ldots s_n s_0)$ and the ith operator has sorting s_i. Thus, the action of π on the interpretation of the arguments is consonant with its action on the operator. Here also the situation is essentially different in **PF**, because a_i need not have s_i as its syntactic sorting.

From $I_{\mathcal{A}}(\pi \circ \alpha, e) = \pi(I_{\mathcal{A}'}(\alpha, e))$, it follows quickly that the truth value of any sentence is the same in $\mathcal{A}$ and $\mathcal{A}'$. For, π is the identity on $\{T, F\}$, and, when ϕ is a sentence, $I(\alpha, \phi)$ does not depend on α. Thus, for any structure $\mathcal{A}$, there is a corresponding structure $\mathcal{A}'$ which satisfies exactly the same sentences, but any two sorts have null intersection.

4.5.2 Full Semantics and General Semantics

The semantics described in this section is called the full semantics for higher order logic. Each structure is considered "full" because it contains all possible functions in every functional sort. It has long been known that there can be no complete deduction procedure relative to this semantics, because the set of formulas valid in a language $\mathcal{L}$ is not recursively enumerable unless $\mathcal{L}$ is almost trivial. However, there is an alternative semantics for simple type theory, due to Henkin, under which a simple deductive apparatus is complete [20, 10].

To adapt that idea to our context, we modify the definition of a *frame* given above. If $\mathcal{L}$ is a language, let S be the set consisting of all sortings for $\mathcal{L}$, instead of simply the set of sorting symbols for $\mathcal{L}$. A *general frame* for $\mathcal{L}$ will be a family of non-empty sets indexed by S. The inductive definition of $I_{\mathcal{F}}$ used above is now unnecessary, as $\mathcal{F}$ is defined directly on higher sortings. However, instead, we stipulate that a frame meet two corresponding conditions:

- $\mathcal{F}(\text{prop}) = \{T, F\}$.
- For the logics **FQ** and **ST**, if s is a higher sorting $(s_1 \ldots s_m s_{m+1})$, then $\mathcal{F}(s) \subseteq$ the set of all total m-ary functions $f : \mathcal{F}(s_1) \times \ldots \times \mathcal{F}(s_m) \to \mathcal{F}(s_{m+1})$.
- For the logic **PF**, if s is a higher sorting $(s_1 \ldots s_m s_{m+1})$, then $\mathcal{F}(s) \subseteq$ the set of all partial (and total) m-ary functions $f : \mathcal{F}(s_1) \times \ldots \times \mathcal{F}(s_m) \rightharpoonup \mathcal{F}(s_{m+1})$.

In the cases of **ST** and **PF**, we stipulate that $\mathcal{A} = \langle \mathcal{F}, I_{\mathcal{C}} \rangle$ is an interpretation only if the function f, defined in the clause for $I(\alpha, e)$, where e is of the form `lambda` $(v_1 \ldots v_n)$ e', always exists as a member of $\mathcal{F}(\sigma(e))$. In the case of **FQ**, we stipulate that $\mathcal{A}$ is an interpretation only if the corresponding function exists, whenever $v_1, \ldots, v_n$ are all variables of base sorting, and comprise all the variables free in e'.

Which semantics is the "right" semantics? We feel strongly that the full semantics, as presented originally, is the right one to use. We offer three reasons.

The first reason is defensive. The fact that no proof procedure can be complete relative to the full semantics does not appear to be a cogent objection. For, even if there exists a complete proof procedure for a logic, a practically useful theorem prover may prefer an incomplete proof procedure. No piece of software can efficiently derive formulas of unbounded complexity, and a theoretically incomplete method may have a wider range of practical applicability than a theoretically complete method. Moreover, when we consider axiomatic theories in semantically complete logics, they are characteristically inadequate to decide all relevant questions about their intended models. Hence, semantic completeness does not buy us what we want anyway, namely the power to decide all questions about a structure such as **N** or **R**.[6]

The second reason for preferring the full semantics is that it allows us to characterize a wide variety of important mathematical structures—**N** and **R** are examples—that are not defined by any axiomatic theory relative to the general semantics. Hence, the full semantics almost always corresponds to the intuitive mathematical meaning of an axiomatic theory. The exceptions to this principle, such as theories formulated for non-standard arithmetic or analysis, can be accomodated within the full semantics without too much trouble: typically, one introduces an explicit predicate of sets characterizing what it means for a set to be "internal" to the non-standard part of the

[6] We owe this line of reasoning to Leonard Monk.

model. Schemas such as induction are then restricted to sets satisfying this "internal" predicate.

The third reason for preferring the full semantics is that it interacts correctly with the process of combining theories. Suppose that T_1 and T_2 are two theories that share no vocabulary, neither constants of their languages nor sort symbols. Then, given two disjoint structures $\mathcal{A}_1$ and $\mathcal{A}_2$ that satisfy T_1 and T_2 respectively, it should be possible to "paste" $\mathcal{A}_1$ and $\mathcal{A}_2$ together to obtain a model of $T_1 \cup T_2$. However, if $\mathcal{A}_1$ and $\mathcal{A}_2$ are not models according to the full semantics, but only according to the general semantics, the result of pasting them together may not satisfy the joint theory. The explanation is that the enriched vocabulary of $T_1 \cup T_2$ creates new instances of schemas such as induction or the principle of definition by recursion. These new instances may not be satisfied in $\mathcal{A}_1$ and $\mathcal{A}_2$. We consider this argument highly relevant to the business of software verification. If the structures $\mathcal{A}_i$ are considered as "implementations" of the "specifications" T_i, then the problem with the general semantics is that implementations of independent specifications cannot be combined to produce an implementation of the whole.

Although we believe that the full semantics is most appropriate for automated deduction systems, we still believe that the general semantics is significant. The completeness theorems, relative to the general semantics, prove that there is a "reasonable" deductive apparatus for the systems we have defined. The theorems pick out, in a precise way, a large and important subset of the set of intuitively valid formulas whose truth is accessible to deduction. Given that there is no deductive procedure that establishes all intuitively valid formulas, it is important to have this supplementary property.[7]

4.5.3 Relations among FQ, ST, and PF

It is clear that for any choice of vocabulary for a language $\mathcal{L}$, $\mathcal{L}$ regarded as a language for **FQ** is a sublanguage of $\mathcal{L}$ regarded as a language for **ST**. The latter, in turn, is a sublanguage of $\mathcal{L}$ regarded as a language for **PF**.

There is, however, a surprisingly close relationship between a set of axioms Γ regarded as a theory in **FQ**, and Γ regarded as a theory in **ST**. Any structure $\mathcal{A}$ may be regarded as a structure for either logic depending on whether one decides to ignore objects of the higher sorts. Moreover, because

[7] William Farmer urged this point.

the semantic clauses for all the constructors of **FQ** are identical with those of **ST**, this operation cannot affect the interpretation of any expression in $\mathcal{L}$. Hence, the truth or falsehood of $\mathcal{A} \models \Gamma$ is independent of which logic is in question, so long as each axiom in Γ belongs to the first order language $\mathcal{L}$. It also follows that the question whether ϕ is a consequence of Γ is independent of whether the logic is **FQ** or **ST**.

Two remarks are in order. First, **FQ** presents a first order syntax, but has a higher order semantics. Second, this means that the difference between a sound theorem prover for **FQ** and one for **ST** is very small. The additions require only that the theorem prover support the syntax of nested sorting lists and also the variable binding λ operator. If automated simplification and heuristics for deduction involving higher type expressions are not needed, but simple proof checking will suffice, then precious little need be done.

The relationship between **ST** and **PF** is less direct, as a structure for **PF** contains a larger class of functions than the "corresponding" structure for **ST**. Because of this, the valid formulas of the two logics are different; to take the simplest example, $\forall f, x \cdot \exists y \,.\, y = f(x)$ is valid in **ST** but not **PF**. Suppose then that Γ is a theory in **ST**, and the axioms in Γ are all closed sentences, as can always be arranged by universally quantifying any free variables.

We will define a map on expressions $e \mapsto \overline{e}$, and a map on structures $\mathcal{A} \mapsto \mathcal{A}'$, and write $\overline{\Gamma}$ for the set $\{\overline{\phi} : \phi \in \Gamma\}$. Then:

$$\mathcal{A} \models_{\mathbf{ST}} \Gamma \text{ iff } \mathcal{A}' \models_{\mathbf{PF}} \overline{\Gamma}.$$

Within **PF**, we define a predicate, "extended hereditarily total" or *eht*, for each sorting s. This predicate picks out those objects of sort $\sigma(s)$ which correspond directly to an object in an **ST** structure. In the base sorts, this includes all objects. At the first level, for a sorting `(sym1 ...symN sym0)`, where `sym0` is not `prop`, it simply picks out the total functions. But at higher types, it is slightly more complex: it must pick out those functions which are total on arguments satisfying *eht*, and have some conventionally determined behavior elsewhere. We shall make the convention that if any argument does not satisfy *eht*, then the value will be undefined, if the range is not prop-sorted, and the false-like object in the range sort if it is prop-sorted.

If s is a base sorting, define eht_s to be $\lambda x : s \,.\, T$. If s is a higher sorting

$s = (s_1 \ldots s_n s_0)$, where s_0 is prop-sorted, let the predicate eht_s abbreviate:

$$\lambda f : s\, \forall v_1 : s_1, \ldots, v_n : s_n\ \neg \left[\bigwedge_{i=1 \text{ to } n} \mathrm{eht}_{s_i}(v_i) \right] \Rightarrow f(v_1,\ldots,v_n) = F_{s_0}.$$

If s is a higher sorting $s = (s_1 \ldots s_n s_0)$, where s_0 is not prop-sorted, let the predicate eht_s abbreviate:

$$\lambda f : s\,\forall v_1 : s_1, \ldots, v_n : s_n\,(\texttt{if-form}\ \begin{array}{l}[\mathrm{eht}(v_1)\wedge\ldots\wedge\mathrm{eht}(v_n)]\\ f(v_1,\ldots,v_n)\downarrow\\ \neg(f(v_1,\ldots,v_n)\downarrow))\end{array}$$

We will write $eht(t)$ to abbreviate $eht_{\sigma(t)}(t)$. Note that eht_s is an expression of *pure* **PF**; that is, no constant from a particular language $\mathcal{L}$ appears in it. Hence, as soon as a **PF** frame $\mathcal{F}$ is given, the interpretation of eht_s is determined. The choice of I_C is not relevant.

We define a map on expressions and formulas $e \mapsto \overline{e}$ inductively:

1. If e is a constant or variable, then $\overline{e} = e$;

2. If e is `forall` $(v_1 \ldots v_n)\,\phi$, then

$$\overline{e} = \texttt{forall}\ (v_1 \ldots v_n)\,[\mathrm{eht}(v_1)\wedge\ldots\wedge\mathrm{eht}(v_n)] \Rightarrow \overline{\phi}.$$

3. If e is `forsome` $(v_1 \ldots v_n)\,\phi$, then

$$\overline{e} = \texttt{forsome}\ (v_1 \ldots v_n)\,[\mathrm{eht}(v_1)\wedge\ldots\wedge\mathrm{eht}(v_n)] \wedge \overline{\phi}.$$

4. If e is `lambda` $(v_1 \ldots v_n)\,e'$, where e' is a formula, then $\overline{e} =$

$$\texttt{lambda}\ (v_1 \ldots v_n)\,(\texttt{if-form}\ \begin{array}{l}[\mathrm{eht}(v_1)\wedge\ldots\wedge\mathrm{eht}(v_n)]\\ [\overline{e'}]\\ (\texttt{the-false}))\end{array}$$

5. If e is `lambda` $(v_1 \ldots v_n)\,e'$, where e' is a predicator and has sorting s, then $\overline{e} =$

$$\texttt{lambda}\ (v_1 \ldots v_n)\,(\texttt{if}\ \begin{array}{l}[\mathrm{eht}(v_1)\wedge\ldots\wedge\mathrm{eht}(v_n)]\\ [\overline{e'}]\\ F_s)\end{array}$$

6. If e is `lambda` $(v_1 \ldots v_n)\,e'$, where e' is *not* prop-sorted and has sorting s, then $\overline{e} =$

$$\texttt{lambda}\ (v_1 \ldots v_n)\,(\texttt{if}\ \begin{array}{l}[\mathrm{eht}(v_1)\wedge\ldots\wedge\mathrm{eht}(v_n)]\\ [\overline{e'}]\\ (\texttt{undefined}((sx))))\end{array}$$

7. If e is built by applying any other constructor to a list of components $c_1, \ldots, c_k$, then $\overline{e}$ is the result of applying that same constructor to the components $\overline{c_1}, \ldots, \overline{c_k}$.

If Γ is a set of sentences, then let $\overline{\Gamma}$ be the set $\{\overline{\phi} : \phi \in \Gamma\}$.

Now, suppose that $\mathcal{A} = \langle \mathcal{F}, I_C \rangle$ is a **ST** structure such that $\mathcal{A} \models_{\mathbf{ST}} \Gamma$. We want to define a **PF** structure $\mathcal{A}'$ which corresponds to $\mathcal{A}$. Let $\mathcal{F}'$ be the **PF** frame having the same base sorts as $\mathcal{F}$. In order to define $I_{C'}$, we need to correlate objects in the higher sorts of $\mathcal{F}$ with objects in the higher sorts of $\mathcal{F}'$. We define a mapping $x \mapsto \tilde{x}$ taking arguments in $\mathcal{F}$ and values in $\mathcal{F}'$.

1. If s is a sorting symbol, then $\tilde{}$ is the identity on $I_{\mathcal{F}}(s)$.

2. If $s = (s_1 \ldots s_n s_0)$ is prop-sorted, and $f \in I_{\mathcal{F}}(s)$, and:

$$(v_1, \ldots, v_n) \in I_{\mathcal{F}}(s_1) \times \ldots \times I_{\mathcal{F}}(s_n) \text{ and } y = f(v_1, \ldots, v_n),$$

then $\tilde{f}(\tilde{v_1}, \ldots, \tilde{v_n}) = \tilde{y}$, while otherwise $\tilde{f}(x_1, \ldots, x_n) = F_{s_0}$.

3. If $s = (s_1 \ldots s_n s_0)$ is not prop-sorted, and $f \in I_{\mathcal{F}}(s)$, and:

$$(v_1, \ldots, v_n) \in I_{\mathcal{F}}(s_1) \times \ldots \times I_{\mathcal{F}}(s_n) \text{ and } y = f(v_1, \ldots, v_n),$$

then $\tilde{f}(\tilde{v_1}, \ldots, \tilde{v_n}) = \tilde{y}$, while otherwise $\tilde{f}(x_1, \ldots, x_n)$ is undefined.

Clearly, $\tilde{}$ is a bijection between $I_{\mathcal{F}}(s)$ and the part of $I_{\mathcal{F}'}(s)$ satisfying eht_s. Define I'_C by the condition $I'_C(c) = \widetilde{I_C(c)}$, and define $\tilde{\alpha}$ by the condition $\tilde{\alpha}(x) = \widetilde{\alpha(x)}$. Let $\mathcal{A}' = \langle \mathcal{F}', I'_C \rangle$.

It is a routine matter of checking each inductive semantic clause to assure oneself that:

$$I(\alpha, e) = y \Rightarrow I'(\tilde{\alpha}, \overline{e}) = \tilde{y}.$$

Hence, for any sentence ϕ, $\mathcal{A} \models_{\mathbf{ST}} \Gamma$ if and only if $\mathcal{A}' \models_{\mathbf{PF}} \overline{\Gamma}$.

Nevertheless, there is a more useful relationship between **ST** and **PF**. And this concerns the theorem provers for the two systems. Given a theorem prover for **PF**, it is very easy to construct a theorem prover for **ST**. In the case of IMPS, a switch would be added to the system to indicate whether the system is in **ST**-mode. The switch is only relevant at two points:

- If the user specifies **ST**-mode, then the constructors `iota`, `iota-p`, `undefined-of-sort`, `is-defined-in`, and `is-defined` should not be installed;
- When the system executes the test `necessarily-defined?`, if it is in **ST**-mode, the procedure should return `t`.

We believe that any rational design for a **PF** theorem prover would make it very easy to switch to **ST**. Thus, while a specifier may have to choose whether to write his specifications using one logic or the other, any theorem prover that can accomodate **PF** will still be available if he chooses **ST** instead.

Section 5
Conclusion

In this paper we have argued in favor of an *interface logic*. It would serve to allow a variety of projects, all attempting to apply formal methods to aspects of software or hardware correctness, to share tools and results. We do not expect that all research efforts would find it a suitable framework, but we think that it will be consistent with the goals of a large enough collection to substantially reduce the amount of duplicated effort.

In addition, we have defined a sequence of three closely connected logics, which we have called **FQ**, **ST**, and **PF**. We believe that they will serve the purpose of providing a common interchange format. In addition, as theorem proving systems become stronger, and can more effectively support **ST** and **PF**, we believe that the other components of VEs will benefit. Not only will it be easy for them to adapt to the logics, but they will also be able to exploit the richer expressiveness of **ST** and **PF** to provide far more effective verification. In particular, **ST** and **PF** seem to us far better suited to reasoning about:

- computations involving real numbers and other continuous domains;
- programs in languages such as Lisp, Scheme, and C, in which procedures are important data objects, and can serve as parameters or return values;
- the semantics of programming languages.

In addition, M. Gordon has argued for the appropriateness of higher-order logic (essentially, **ST**) as a formalism for hardware verification [17]. However, we believe that there will be immediate benefits even from adapting existing VE components to use the interface logic in the guise **FQ**, as it will enable some of the newer, and strongest, verification condition generators to be matched with some superior theorem provers. Indeed, we believe that the adoption of an interface logic will aid in producing a high-quality, well-integrated *user-oriented* verification environment, in a relatively short time, by effectively building onto the best currently available components.

We would like to end by emphasizing the method we have used in this paper. In particular, we have relied not only on a sequence of written studies [9, 10, 24, 25], but also on our first-hand experience in implementing the most complex of the logics proposed here, **PF**, in the IMPS system. This work, funded partly under the MITRE-Sponsored Research program, and partly under the present effort, guarantees that our proposals for interface logics are practical in the sense that currently existing ideas on how to structure theorem provers can lead to effective theorem provers for all three of the logics described.

List Of References

1. H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.
2. R. Boyer and J Moore. *A Computational Logic.* Academic Press, 1979.
3. R. Boyer and J Moore. *A Computational Logic Handbook.* Academic Press, 1988.
4. R. Burstall and J. Goguen. Putting theories together to make specifications. In *Fifth International Joint Conference on Artificial Intelligence*, 1977.
5. L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymophism. *Computing Surveys*, 17:471–522, 1985.
6. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
7. D. Craigen, S. Kromodimoeljo, I. Meisels, A. Neilson, W. Pase, and M. Saaltink. m-EVES: Collected papers. Technical report, I. P. Sharp Associates Ltd., 1988.
8. R. Constable et. al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, 1986.
9. W. M. Farmer. Abstract data types in many-sorted second-order logic. Technical Report M87-64, The MITRE Corporation, 1987.
10. W. M. Farmer. A partial functions version of Church's simple theory of types. *Journal of Symbolic Logic*, 55(3):1269–92, 1990. Also MITRE Corporation report M88-52 (revised 1990).
11. W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. Technical Report M90-19, The MITRE Corporation, 1990.
12. J. A. Goguen. Principles of parameterized programming. Technical report, SRI International, 1987.

13. Joseph A. Goguen and Timothy Winkler. Introducing OBJ3. Technical report, SRI International, August 1988.

14. D. I. Good. Revised report on Gypsy 2.1. Technical report, University of Texas, 1984.

15. M. Gordon. HOL: A proof-generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*. Kluwer, 1987.

16. M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

17. M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, University of Cambridge Computer Laboratory, 1985.

18. J. D. Guttman. The Ina Jo specification language: A critical study. RADC-TR 86-47, Rome Air Development Center, 1986.

19. J. D. Guttman, W. M. Farmer, and F. J. Thayer. IMPS: A proof system for a generic logic. Technical report, The MITRE Corporation, 1990. Submitted for publication.

20. L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15, 1950.

21. D. E. Knuth. *The TeXbook*. Addison-Wesley, 1984.

22. C. Marceau and C. D. Harper. An interactive approach to Ada verification. In *12th National Computer Security Conference*, 1989.

23. D. F. Martin and J. V. Cook. Adding Ada program verification capability to the state delta verification system (SDVS). In *11th National Computer Security Conference*, 1988.

24. L. G. Monk. PDLM: A proof development language for mathematics. Technical Report M86-37, The MITRE Corporation, 1986.

25. L. G. Monk. Inference rules using local contexts. *Journal of Automated Reasoning*, 4(4), 1988.

26. J. A. Rees, N. I. Adams, and J. R. Meehan. *The T Manual.* Computer Science Department, Yale University, fifth edition edition, 1988.

27. J. Seldin. Mathesis: The mathematical foundations of Ulysses. Technical report, Odyssey Research Associates, 1987.

28. SRI International Computer Science Laboratory. EHDM specification and verification system (version 4.1) preliminary definition of the EHDM specification language. Technical report, SRI International, 1988.

29. R. M. Stallman. GNU *Emacs Manual.* Free Software Foundation, sixth edition, version 18 edition, 1987.

DISTRIBUTION LIST

A010

R. D. Haggarty

G010

V. A. DeMarines
C. M. Sheehan

G110

H. A. Bayard
E. H. Bensley
E. L. Lafferty
L. J. LaPadula
J. K. Millen
P. S. Tasker

G116

F. Belvin
W. R. Gerhart

G117

J. D. Guttman (30)
Technical Staff

External

RADC/COAC
Griffiss Air Force Base
Rome NY 13441

John Faust (5)

Dale M. Johnson
D. M. Johnson
Project Leader, 4030