A CALCULUS FOR THE CONSTRUCTION OF MODULAR PROLOG PROGRAMS*

D. T. SANNELLA AND L. A. WALLEN

▷ We present a module language for PROLOG based on the theory of modularity underlying the Standard ML module system. The language supports the construction of hierarchically structured programs from parametrized components and provides a form of structural data abstraction. A formal semantics is given for the system which translates modular programs into conventional programs. ◁

1. INTRODUCTION

Module systems and data abstraction are powerful methods for managing the complexity of large programs. Most standard PROLOG systems lack both facilities. As efficient implementations of PROLOG become widely available, the lack of such facilities becomes a serious hurdle to the use of the language for large projects. The majority of previous proposals for module systems in a logic programming context require significant extensions of the PROLOG language and heavily modified interpreters/compilers (see, e.g., [9, 4]).

In this paper we present a module system for the *standard* PROLOG language that also supports structural data abstraction. In its role as a metalanguage, the module system is almost completely decoupled from the underlying PROLOG language, a separation which facilitates its implementation within existing PROLOG systems. What interaction there is occurs, as expected, via the extralogical predicates and the method of referencing predicate and function constants. The language supports the construction of programs from generic, or parametrized, components, and provides a notion of program well-formedness which correctly excludes many common PROLOG programming errors. The integration of PROLOG's extralogical facilities into the module system supports a hierarchical view of the PROLOG database.

Address correspondence to Dr. L. A. Wallen, Computing Laboratory, University of Oxford, Oxford OX1 3QD, England, U.K.

*Research supported in part by SERC grants GR/D/44874 and GR/D/44270.

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/92/$3.50

The module language is based on the theory of modularity underlying the Standard ML module system [7]. A deliberate effort has been made to maintain consistency with that system for the following (positive) reasons:

The theory of modularity on which the system is based is practically independent of the underlying programming language; to instantiate the theory for a particular language we need only decide on the ways in which programs in that language can interact. (This we do for PROLOG in Section 1.3.)

Program development methods based on the module language have been extensively investigated (e.g., [14]). Such methods are deliberately couched so as not to prescribe the underlying programming language and logic [3, 13]. As a consequence, this work is directly applicable to the modular PROLOG language presented here, and this fact justifies the use of the term "calculus" for the language presented below.

The module system for ML has been implemented and is well liked.

Since the language encourages the construction of programs in a hierarchical manner, an approach must be formulated for the sharing of common subcomponents. The approach outlined here is based on MacQueen's notion of a *sharing specification*.

In the rest of this introduction we present the functional approach to modularity on which the module language is based and introduce some PROLOG-specific terminology. We also discuss the choice of the appropriate program unit to form the basic component of PROLOG programs in the large. The second section contains details of the module language and examples.

While the PROLOG language itself is not a pure logic programming language, interest in the language stems from its proximity to this ideal. It is important therefore that any notion of module preserve the logical interpretation of a program. In Section 3 (and in the appendix) we give a translation semantics for our system by showing how a structured program written using the module language is equivalent to a unstructured program. Such a semantics stresses the metalogical nature of the module language and, perhaps more importantly, leads directly to an implementation. Moreover, any logical semantics given to programs in the unstructured PROLOG language lifts via this translation to a semantics for structured programs.

Finally, in Section 4, we integrate PROLOG's extralogical predicates into the proposed module system.

1.1. A Functional Approach to Modularity

A standard technique for managing complexity arising from interaction is *functional abstraction*: a component is abstracted from concrete contexts by providing a specification of its

requirements: the components it requires in order to function correctly, and its

results: the components it produces when supplied with the components it requires.

The internal details of the abstracted component are defined in terms of the components declared in the requirement specification.

The most obvious example of this technique in operation is of course in the design of functional programming languages, where such abstractions form the basic program unit. The function body defines how to compute the output (results) in terms of the input (requirements). Typed programming languages increase the power of the language available for specifying requirements and results.

For programming "in the large", the program units that we might consider abstracting are collections of primitive program components: predicate definitions in the case of PROLOG, functions and types in the case of ML, procedures and global variables in the case of Pascal, etc. Such abstractions are in effect program-valued functions. When applied to program units that satisfy their input specification (imports), they result in a program unit that satisfies their output specification (exports). Large programs can be constructed in stages by applying these program-valued functions successively to previously constructed program units. The coherence of the resulting program is ensured by the requirement that the arguments supplied to such functions satisfy the input specification of the function.

It is important to realize that this approach to complexity management is functional *in essence*, irrespective of the internal nature of the program units themselves. Following MacQueen [7], we shall call abstracted program units *functors*, parameter/result specifications *signatures*, and the results of functor applications *structures*.

A program therefore is a structure. Functors are parametrized structures used to manage the dynamics of program construction. Provided that its parameter signature contains sufficient information about the class of structures to which a functor may be correctly applied, the functor may even be compiled separately (e.g., [6]). The application of the compiled functor then performs the job of linking the components of the input structures into the compiled functor body.

1.2. Terminology

We adopt standard logical terminology to refer to various PROLOG constructs. Since the module language draws a distinction between predicate and function constants, we cannot safely use the PROLOG terminology which confuses the two.

A predicate constant consists of a predicate symbol and a natural number called an *arity*. Similarly for function constants. We write such constants as *symbol* : *arity*. A *term* is a variable, or a function constant together with a sequence of terms. The length of the sequence must equal the arity of the constant for the term to be well formed. An *atom* is a predicate constant together with a sequence of terms. Again the length of the sequence must equal the arity of the constant for the atom to be well formed. A program *clause* consists of a *head* and a *body*. The head is an atom. The body is a finite sequence of atoms. A *predicate definition* is a finite sequence of program clauses the heads of which have the same predicate constant. A *program* is a finite set of predicate definitions.

1.3. Well-Formed PROLOG Programs

In this section we choose the basic component of a PROLOG program "in the large", and decide on the ways in which such components may legitimately

interact. As we mentioned above, this is the step of instantiating the general theory of modularity to a concrete programming language. The particular choices we make are designed to:

prevent certain PROLOG programming errors that are common in the program development cycle;

support structural data abstraction, and

maintain a logical interpretation of the program component.

Interactions in PROLOG programs arise by means of references to predicate and function constants. References to predicates are made within the bodies of program clauses. References to functions are made within individual atoms for the purposes of unification.

The key to managing such interactions and limiting the possibility of error lies in controlling the language (set of constants) available for writing program clauses. There are two elements to this language: the predicate language and the function language. We consider that large PROLOG programs are formed from individual components, or *structures*, each of which consists primarily of a set of predicate definitions. A structure is well formed with respect to predicates if every predicate constant used within the bodies of predicate definitions in the structure are either declared within that structure or "imported" via the mechanisms of the module language. In effect, we are building cross-referencing facilities into the environment. Use of the module language catches spelling and arity errors earlier than postprocessing systems.

We also introduce an explicit function constant declaration which extends the function language available for writing program clauses. This gives the structural data abstraction we have referred to above. The program clauses within a structure are forbidden to reference individual function constants for the purposes of explicit unification unless those constants are declared within the structure or "imported" via the mechanisms of the module language. If programmers wish to hide the representation details of data structures, they refrain from exporting the particular function constants used from the defining structure. The mechanisms available to achieve this effect are discussed below.

2. THE MODULE LANGUAGE

In this section we introduce the major elements of the module language for pure PROLOG programs. A more formal definition of syntax and semantics can be found in the next section. In Section 4 we describe the integration of the extralogical predicates of PROLOG with the module language. The details of the PROLOG module system differ slightly from the corresponding constructs in the ML system [7], but the overall feel is the same. Consequently, [5] provides a good introduction to the actual use of the module language.

Structures form the basic program unit of PROLOG programs in the large. They may be organized hierarchically and consist of encapsulated sets of predicate, function, and substructure declarations. *Signatures* are specifications of structures and serve as interfaces. They specify the constants a structure provides for the outside world. *Abstractions* are a special kind of structure; they are formed by

hiding some of the contents of a structure, declaring the structure with a smaller signature than it would otherwise have had. Finally *functors* are parametrized structures with explicit interfaces; they are structure-valued functions.

It is important to realize that sensibly structured programs will be written almost exclusively using functors. A functor enables a programmer to isolate a component of his or her program and provide an explicit interface indicating the environments in which the component will function correctly, and how it enhances that environment. A concrete program is constructed by applying functors to suitable parameters to build the program in stages. This is the process of *linking*, similar to "consulting" in current PROLOG terminology. If a debugged component can be relinked successfully, the module language ensures that the changes made did not render the overall program incoherent.

Another point to note is that functors (program-valued functions), structures (programs), and signatures (interfaces) can all be declared and explicitly named. The names are considered as residing in a global environment. This is in contrast to the hierarchy of encapsulated name spaces in which function and predicate constant reside. This global name space should be compared with the filestore from which PROLOG programs are currently constructed using `consult` and `reconsult` operations.

Although modules are for constructing large programs, we are forced (due to space limitations) to use small examples to get the ideas across. We beg our readers' indulgence and hope that they can see how the same principles work in the large.

2.1. *Primitive Declarations*

The module language admits two primitive declaration forms: function constant declarations and predicate constant declarations. Function declarations come in two flavors: one for introducing new function constants, and one for defining a new function constant in terms of an existing constant (renaming); e.g.,

```
fun leaf:0.
fun tree:3.
```

declares the nullary function constant `leaf:0` and the function constant `tree:3` to be available within the scope of the declaration. (These may be interpreted as data constructors for labeled binary trees.) We can introduce two new constants by renaming the above constants using declarations of the form

```
fun nulltree:0=leaf.
fun node:3=tree.
```

Such declarations are allowed within the scope of the previous declarations. The new constants are considered identical to the old ones for purposes of unification. The arity specification can also be omitted provided no ambiguity arises. Natural abbreviations are supported such as

```
fun leaf:0, tree:3.
```

As mentioned above, function constant declarations determine the language avail-

able for the terms occurring within predicate definitions. They may be viewed as data constructors.

Predicate constants are declared by their definitions. For example, the program clauses

```
isleaf(leaf).
isnode(tree(_,_,_)).
```

serve to declare the predicate constants `isleaf:1` and `isnode:1` for use within the current structure.

2.2. *Structures*

Structures are the basic building blocks for programs in the large. They consist of an encapsulated set of declarations which define the language available within the structure and which the structure makes available to the outside world. The encapsulation limits the scope of the primitive declarations.

Structures are named with declarations of the form

`structure` $S = \langle$struct expr$\rangle$.

There are four types of *structure expression*:

encapsulated declarations,

structure names (possibly qualified),

⟨structure, signature⟩ pairs, and

functor applications.

As an example of the first form, here is a structure implementing some operations on labeled binary trees:

```
structure BtreeData0 =
  struct
    fun leaf:0, tree:3.
    isleaf(leaf).
    isnode(tree(_,_,_)).
    mkleaf(leaf).
    mknode(A, Left, Right, tree(A,Left,Right)).
    label(tree(A,_,_), A).
    leftchild(tree(_,Left,_), Left).
    rightchild(tree(_,_,Right), Right).
  end.
```

The `struct...end` are simply brackets. Such brackets should be compared with the use of `[user]` within existing PROLOG systems. Ignoring the function constant declarations, the major difference is that the environment to which the definitions are added can be explicitly named (in this case `BtreeData0`). Structure names are considered global.

Structure declarations may appear inside other structures. This represents an explicit dependence of one structure on another and is the mechanism by which hierarchically structured programs are formed. Structure names are used to reference existing structures. For example, the following structure extends the previous one by defining a membership relation over binary trees:

```
structure BtreeMem0 =
  struct
    structure B = BtreeData0.
    member(A, B/tree(A,_,_)).
    member(A, B/tree(_,Left,_)) :-
      member(A, Left).
    member(A, B/tree(_,_,Right)) :-
      member(A, Right).
end.
```

Constants in the substructure `BtreeData0` are referenced via *qualified* names such as `B/tree`. This structure is well formed because `BtreeData0` does contain a function constant of the appropriate arity. A substructure declaration is considered to make all qualified names accessible to the outer structure. So for instance, the predicates `isleaf:1`, `isnode:1`, `mkleaf:1`, `mknode:4`, `label:2`, `leftchild:2`, and `rightchild:2` may be used to define new predicates within `BtreeMem0`. They are referenced via the qualified names `B/isleaf` etc. `BtreeMem0` is said to be a *derived* structure.

The qualifications serve to distinguish different instances of the same constants which in the case of predicate constants could have different definitions associated with them. In the absence of any possibility of confusion, the declaration

```
open BtreeData0.
```

could have been used. This makes the constants of `BtreeData0` directly accessible within the derived structure, i.e., no qualification is required.

2.3. Signatures

Signatures specify the contents of structures. They achieve this via a specification of the language declared within structures. For example, the language declared within the structure `BtreeData0` is the following:

```
sig
  fun leaf:0, tree:3.
  pred isleaf:1, isnode:1, mkleaf:1, mknode:4,
                label:2, leftchild:2, rightchild:2.
end.
```

Again, the `sig ... end` are merely brackets. Signatures can be *inferred* from a well-formed structure declaration in an obvious way. Let us call this signature

BTREEDATA0. The signature inferred from the derived structure BtreeMem0 is

```
sig
  structure B:BTREEDATA0.
  pred member:2.
end.
```

indicating the dependence on a structure specified in turn by the signature BTREEDATA0. Indeed BtreeMem0 could have been declared explicitly with a signature SIG thus:

```
structure BtreeMem0:SIG =
  struct
    ...
  end.
```

Such a declaration is well-formed provided the signature inferred from the structure *matches* the explicit signature SIG. A signature matches another when the former is a superset (i.e., superlanguage) of the latter. In case the inferred signature of the structure is larger than the explicit signature; the additional names are hidden in the resulting structure.

The syntax of signatures should be obvious. Signatures are named and manipulated in a similar manner to structures. For example,

```
signature BTREEDATA1 =
  sig
    pred isleaf:1, isnode:1, mkleaf:1, mknode:4,
                label:2, leftchild:2, rightchild:2.
  end.
```

declares a signature that specifies structures containing the predicate constants mentioned. Notice (for future reference) that this signature is a specification of structures that do not necessarily contain the function constants leaf:0 and tree:3. This should be compared with the signature BTREEDATA0. Again, signature names exist in a global name space.

2.4. Information Hiding and Abstraction

Data abstraction is a powerful method of limiting the interactions between program components and easing program development. PROLOG sorely lacks an environmentally supported facility along these lines.

Notice that the structure BtreeMem0 explicitly references the function constants used as data constructors for the implementation of binary trees provided by its substructure BtreeData0. Any alteration to the function constants in BtreeData0 will require alteration to the predicate definitions in BtreeMem0 for the latter to be coherent. We can ensure that all structures that make use of our data implementation do not make such assumptions about the underlying representation of binary trees by using an *abstraction* declaration.

The declaration

```
abstraction BtreeData1:BTREEDATA1 =
  struct
    fun leaf:0, tree:3.
    isleaf(leaf).
    isnode(tree(_,_,_)).
    mkleaf(leaf).
    mknode(A, Left, Right, tree(A,Left,Right)).
    label(tree(A,_,_), A).
    leftchild(tree(_,Left,_), Left).
    rightchild(tree(_,_,Right), Right).
  end.
```

results in a structure with signature BTREEDATA1 rather than BTREEDATA0 as before. (Recall that BTREEDATA1 does not contain the function constants, whereas BTREEDATA0 does.) The implementation details of the data structure have been effectively hidden from the rest of the program.

If we now try to define our membership structure in the same way as before—namely,

```
structure BtreeMem0 =
  struct
    structure B = BtreeData1.
    member(A, B/tree(A,_,_)).
    ...
  end.
```

—the resulting structure is not well formed, since the language available for writing clauses within the structure does not include the function constants B/leaf:0 and B/tree:3. The following declaration is well formed:

```
structure BtreeMem1 =
  struct
    structure B = BtreeData1.
    member(A, Tree) :-
      B/label(Tree, A).
    member(A, Tree) :-
      B/leftchild(Tree, Left),
      member(A, Left).
    member(A, Tree) :-
      B/rightchild(Tree, Right),
      member(A, Right).
end.
```

The `abstraction` construct is not a separate construct; it can be derived via a special use of functors. (See Section 3.2.)

2.5. *Structure Equality*

We consider structure expressions to be *generative*, i.e., the declaration of the same structure expression twice gives two different structures. Structure declaration is therefore like a `consult` as opposed to a `reconsult`, except that the constants of one instance are not considered identical to the constants of the other. One reason for this is that we want to formalize and support the process of program construction. If the definitions within a structure are altered (e.g., debugged), the changes must be reflected in the rest of the program. Functors (Section 2.6) enable this rebuilding to be effected with a minimum of effort. The new structure is simply linked in by repeating the functor applications. Only those parts of the program that depend on the updated structure need be relinked.

The equivalent of the `reconsult` operation is considered to be an implementation level operation. In some circumstances it is possible to replace a substructure of a program without affecting the code which depends on it. (This is made easier by liberal use of the `abstraction` operation.) In such circumstances a more efficient *implementation* of relinking can be achieved which is equivalent to a `reconsult`.

2.6. *Parametrized Structures and Functors*

The dependence of the derived structure `BtreeMem1` on the structure `BtreeData1` is explicitly represented by the fact that the latter is a substructure of the former. Although we have insulated the membership code from dependence on the actual PROLOG data structure used to implement trees, we have built in a particular implementation nonetheless. This is because the representation is included in a substructure (and because structure declarations are generative). If our program utilized two different representations of binary trees, we would have to write the membership code twice, once for each representation.

We really want to be able to define the membership code so that it works correctly on any implementation of binary trees that provides the predicates `label:2`, `leftchild:2`, and `rightchild:2`. We can then obtain two instances of the code by applying such an abstraction twice, once to each representation, rather than actually writing the code twice.

We lambda-abstract the body of the structure `BtreeMem1` on its substructure to form a structure-valued function called a *functor*. The functor may be *applied* to another structure that contains the predicates on which the (now abstract) code depends (i.e., `label:2`, `leftchild:2`, and `rightchild:2`).

To ensure that such an application results in well-formed code, we need to be able to specify the class of parameters the functor accepts. Signatures, of course, provide us with just this sort of specification. Functors are thus "typed" structure-valued functions, signatures being the types.

Returning to our example, we can abstract the structure defining `member:2` from its dependence on a particular substructure to make the functor `AbsB-`

treeMem by the functor declaration

```
functor AbsBtreeMem(X:BTREEDATA1) =
  struct
    structure B = X.
    member(A, Tree) :-
      B/label(Tree, A).
    member(A, Tree) :-
      B/leftchild(Tree, Left),
      member(A, Left).
    member(A, Tree) :-
      B/rightchild(Tree, Right),
      member(A, Right).
  end.
```

AbsBtreeMem takes as a parameter any structure which matches its parameter specification and returns a structure with the appropriate elements of the parameter "linked" in.

The functor application

```
AbsBtreeMem(BtreeData1).
```

produces a structure that contains an implementation of member:2 over the particular binary tree data structure implemented by BtreeData1. In order to obtain an implementation of member:2 which works over some different implementation, say BtreeData2, we simply apply the functor to the alternative structure thus:

```
AbsBtreeMem(BtreeData2).
```

The coherence of the resulting structure is ensured by requiring the actual parameter to the functor to match the parameter signature.

The signature of the structure resulting from the application of the functor AbsBtreeMem to any structure with signature BTREEDATA1 is

```
sig
  structure B:BTREEDATA1
  pred member:2
end.
```

As with structure declarations, the functor could have been declared with an explicit result signature. The signature inferred from the application would then have been required to match this signature. Such explicit specification of structures is a sort of coherency check and is a useful form of documentation. Names not mentioned in the explicit result signature are effectively hidden.

If we do not wish the parameter structure to be inherited by the resulting structure, we refrain from declaring it explicitly within the body of the functor.

2.7. *Sharing*

Interactions between program components occur via common substructures. For example, here is a functor implementing equality over labeled binary trees:

```
functor AbsBtreeEq(X:BTREEDATA1) =
  struct
    structure C = X.
    eqtree(Tree1, Tree2) :-
      C/isleaf(Tree1),
      C/isleaf(Tree1).
    eqtree(Tree1, Tree2) :-
      C/label(Tree1, Label),
      C/label(Tree2, Label),
      C/leftchild(Tree1, Left1),
      C/leftchild(Tree2, Left2),
      C/rightchild(Tree1, Right1),
      C/rightchild(Tree2, Right2),
      eqtree(Left1, Left2),
      eqtree(Right1, Right2).
  end.
```

If `BTREEMEM1` and `BTREEEQ` are the respective result signatures of the declarations

```
AbsBtreeMem(BtreeData1).
AbsBtreeEq(BtreeData1).
```

then a functor parametrized over such an equality structure and membership structures can be defined as follows:

```
functor AbsBtreeUtil(X:BTREEMEM1, Y:BTREEEQ) =
  struct
    structure U = X.
    structure V = Y.
    foobar(Elem, Tree1, Tree2) :-
      ...
      U/member(Elem, Tree1),
      V/eqtree(Tree1, Tree2),
      ...
      ...
  end.
```

Suppose we build our program in the following way:

```
structure BtreeMem = AbsBtreeMem(BtreeData1).
structure BtreeEq = AbsBtreeEq(BtreeData2).
structure BtreeUtil = AbsBtreeUtil(BtreeMem,BtreeEq).
```

where `BtreeData2` is, as before, a different representation of binary trees from that implemented in `BtreeData1`. Clearly the membership and equality predicates, used in the definition of `foobar:3` in `AbsBtreeUtil`, are supposed to work over the same data structure. However, they are defined using different representations (or versions) of binary trees.

We wish to classify the program built as above as being ill formed. To do this, following MacQueen [7], we use a so-called *sharing specification*. In the case of PROLOG, sharing specifications are equalities (or *path equations*) between substructures in the parameters of functors. For example, our parametrized tree utility structure should be written as follows.

```
functor AbsBtreeUtil(X:BTREEMEM,Y:BTREEEQ sharing
  X/B=Y/C) =
  struct
    structure U = X.
    structure V = Y.
    foobar(Elem, Tree1, Tree2) :-
      ...
      U/member(Elem, Tree1),
      V/eqtree(Tree1, Tree2),
      ...
      ...
    end.
```

The path equation `X/B=Y/B` indicates that the named substructures of the parameters must be identical. (Recall that structure expressions are generative.) Consequently, with this new functor, our previous attempt will be recognized as ill formed, whereas the following construction, using a single representation of binary trees, is well formed:

```
structure BtreeMem = AbsBtreeMem(BtreeData1).
structure BtreeEq = AbsBtreeEq(BtreeData1).
structure BtreeUtil = AbsBtreeUtil(BtreeMem,BtreeEq).
```

2.8. Restrictions

To enable static checking of well-formedness and sharing constraints (see Section 2.7) and to ensure decoupling of the module system from the underlying PROLOG language, a number of restrictions on structure, signature, functor (and abstrac-

tion) declarations are imposed. These are as follows (read either "structure", "signature", "functor", or "abstraction" for "module construct" below):

1. *Declaration before use*: a module construct must be declared before any reference to it is made.
2. *No dynamic declarations*: the declaration of a module construct may not appear in the body of a predicate definition.

The first restriction is not strictly necessary, since a two pass compiler could resolve references. We impose it for simplicity and to avoid recursive structure definitions. In addition, such a restriction enables the compiler to dereference qualified names at declaration time when structures are defined interactively.

The second restriction is crucial to minimize the impact of the module system on the standard language. It is possible to lift this restriction also and allow programs that dynamically construct and manipulate new module constructs. The result will of course be a new programming language, which will contain some of the facilities of object-oriented languages, a structure being viewed as the analogue of an object. Although this would be an interesting avenue for further research, it fails the criteria we set out in the introduction that the module system must be easily implementable on top of existing PROLOG systems; hence the restriction.

3. SYNTAX AND SEMANTICS

In this section we give a BNF syntax of the module constructs introduced in the preceding sections. Appendix A contains a complete formal semantics which explains in detail how to convert a program written in modular PROLOG into a program in ordinary PROLOG. Only PROLOG programs which do not make use of the predicates discussed in Section 4 are considered. The semantics is denotational in style; the denotation assigned to a program is a sequence of PROLOG clauses together with an environment which allows subsequent goals to be translated into ordinary PROLOG goals. A semantics for Standard ML including module constructs is given in [8]; it is presented in the form of a "natural" semantics.

3.1. Core Syntax

PROGRAMS *prog*

prog ::= *dec*

SIGNATURE BINDINGS *sigb*

sigb ::= *atid* = *sigexpr*

FUNCTOR BINDINGS *funb*

funb ::= *atid*(*plist*) = *strexpr*

plist ::= $atid_1 : sigexpr_1, \ldots, atid_n : sigexpr_n$ $[\texttt{sharing}\ patheq_1\ \texttt{and}\ \ldots\ \texttt{and}\ patheq_m]$ $n \geq 0, m \geq 1$

patheq ::= $id_1 = \cdots = id_n$ $n \geq 1$

STRUCTURE BINDINGS *strb*

strb ::= *atid* = *strexpr*

SIGNATURE EXPRESSIONS *sigexpr*

sigexpr ::= *atid*
sig *spec* end
spec ::= pred *atid* : *nat*.
fun *atid* : *nat*.
structure *specstrb*$_1$ and ... and *specstrb*$_n$
[sharing *patheq*$_1$ and ... and *patheq*$_m$]. $n \geq 1, m \geq 1$
spec spec'
specstrb ::= *atid* : *sigexpr*

STRUCTURE EXPRESSIONS *strexpr*

strexpr ::= *id*
struct *dec* end
strexpr : *sigexpr*
atid(*strexpr*$_1$, ..., *strexpr*$_n$) $n \geq 0$

DECLARATIONS *dec*

dec ::= *atid*(*term*$_1$, ..., *term*$_n$) [:- *atom*$_1$, ..., *atom*$_m$]. (a PROLOG clause)
fun *atid* : *nat*.
fun *atid* : *nat* = *id*.
open *id*.
structure *strb*$_1$ and ... and *strb*$_n$. $n \geq 1$
signature *sigb*$_1$ and ... and *sigb*$_n$. $n \geq 1$
functor *funb*$_1$ and ... and *funb*$_n$. $n \geq 1$
dec dec'

Brackets enclose optional items. The symbol *atid* denotes an atomic identifier (one which does not contain slashes), while *id* denotes an identifier in modular PROLOG, that is *id* ::= *atid* | *atid*/*id*.

3.2. Derived Forms

The functor binding

atid(*plist*) : *sigexpr* = *strexpr*

is equivalent to

atid(*plist*) = *strexpr* : *sigexpr*

The structure binding

atid : *sigexpr* = *strexpr*

is equivalent to

atid = *strexpr* : *sigexpr*

The declaration

inherit *atid*

is equivalent to

structure *atid* = *atid*

The specification

pred $atid_1 : nat_1, \ldots, atid_n : nat_n$.

is equivalent to

pred $atid_1 : nat_1$.

...

pred $atid_n : nat_n$.

The declaration

fun $atid = id$.

is equivalent to

fun $atid : n = id$.

provided that *id* unambiguously refers to a function constant with arity n.

The specification

fun $atid_1 : nat_1, \ldots, atid_n : nat_n$.

is equivalent to

fun $atid_1 : nat_1$.

...

fun $atid_n : nat_n$.

and equivalently for declarations of this form.

Evaluating a query (in the top-level "structure") after compiling the program

dec

abstraction *atid* : *sigexpr* = *strexpr*.

dec'

is equivalent to compiling the program

dec

functor $F(atid : sigexpr)$ = struct *dec'* end.

where F is an unused functor name, and then evaluating the query in the structure $F(strexpr)$.

4. RUN TIME MANIPULATION OF PROGRAMS

Practical PROLOG programs make use of various extralogical built-in predicates. In this section we propose an interface between the module language presented above and these predicates. Apart from completing the design of the module system, these proposals result in natural facilities for structuring the PROLOG database. This increases the natural appeal of the PROLOG language for use in advanced database applications.

4.1. Structure References and Constants

The module language draws a distinction between function and predicate constants. In addition, it introduces a distinct construct, the *structure reference*, which is a pointer or database reference to a structure.

4.1.1. Predicate and Function Constants. Constants are ⟨*Symbol*, *Arity*⟩ pairs from a logical point of view. Operationally however they are triples ⟨*Symbol*, *Arity*, *Ref*⟩, where *Ref* is a structure reference. Qualified names present in the source code are translated into such constructs at the time a structure is built (either explicitly, by means of an encapsulated declaration, or via a functor application).

Consequently, the construction of predicate and function constants is also affected by the module language. We replace the usual `functor:3` with

`predicate(`?*Atom*, ?*Symbol*, ?*Arity*`)`
?*Atom* is an atom with predicate constant ⟨?*Symbol*, ?*Arity*, *Ref*⟩, where *Ref* is a reference to the current structure.

`predicate(`?*Atom*, ?*Symbol*, ?*Arity*, ?*Ref*`)`
?*Atom* is an atom with predicate constant ⟨?*Symbol*, ?*Arity*, ?*Ref*⟩.

Similarly for functions, namely: `function:3` and `function:4`. These predicates have the usual restrictions as to which combinations of arguments may be uninstantiated.

Errors result if the constant is not part of the language of the referenced structure. The `arg:3` construct is unchanged.

4.1.2. Structure Names and References. Structure names are strings (or possibly terms) of the form `A/B/...` which can be used dynamically as relative references to structures. We propose the following built-in predicates to enable run time construction and manipulation of structure names and references:

`current_structure(`?*Ref*`)`
?*Ref* is the reference of the current structure.

`structure(`?*Ref*, +*Str*`)`
?*Ref* is the reference for the name +*Str* relative to the current structure.

`structure(`?*Ref*, +*Str*, +*Ref*`)`
?*Ref* is the reference for the name +*Str* relative to the structure +*Ref*.

4.2. Manipulation of Predicate Definitions

The extralogical predicates (in the family of) `assert:2` and `retract:2` enable the run time construction and manipulation of predicate definitions. In the terminology of the module language: they can alter the content of structures. The restrictions on functor, signature, and structure declarations ensure that the structure of the program "in the large" cannot be altered at run time. The ability to manipulate predicate definitions is however an extremely useful facility for database applications and the like.

The predicates `call:1` and `clause:2` do not manipulate the program explicitly, but nevertheless interact with the module language in a similar manner. The only issue is to determine which predicate constant is being referred to within a given call of such a primitive.

For that purpose we introduce new versions of each of the above predicates with an extra argument which can contain a structure reference. We illustrate using `call` and `assert`:

`call(` +*Atom*`)`
calls +*Atom* in the current structure.

`call(` +*Atom*, +*Ref*`)`
calls +*Atom* within the structure +*Ref*.

`assert(` + *Clause*`)`
asserts +*Clause* into the current structure.

`assert(` + *Clause*, +*Ref*`)`
asserts +*Clause* into the structure +*Ref*.

So, for instance, the sequence of goals

```
structure(Ref,a/b), call(pred(fun),Ref)
```

is "equivalent" to a goal of the form

```
call(a/b/pred(a/b/fun))
```

in the current structure. Predicates of the second form are more powerful, since the path to the structure referenced may not be expressible as a structure name from the current structure.

Run time errors arise if the atom (or clause) is not expressible within the language declared in the structure referenced.

The PROLOG database can therefore be viewed as a hierarchy of databases, communication being facilitated by the assertion and retraction primitives. Database names can be manipulated explicitly by programs, and specific predicates that are defined in these databases manipulated by `call`ing (`assert`ing, etc.) them in the usual way.

4.3. Errors: Further Declaration Forms

In the above we have specified that errors occur if an attempt is made to construct constants or manipulate program clauses within a structure that are outside the language of that structure. This is of course a very conservative position. It is taken in the interests of maintaining the well-formedness of the program throughout execution.

We can relax these restrictions in many interesting ways. In this section we shall mention a few that come to mind. The most appropriate methods may become more obvious through experience.

4.3.1. Predicates with No Source Code Definition. To declare predicates without associating a definition with them in the source code, one can provide an explicit declaration form similar to the declaration form for functions. This is in the spirit

of cross-reference systems that allow users to declare that certain constants are being treated specially by the program (`asserted` dynamically perhaps).

4.3.2. Assertions about Structures. Alternative declaration forms can be provided for structures to assert that any constant within a particular range (e.g., alphanumeric, any arity) should be considered part of the language of the structure. This is useful if the program is interacting with a user typing data. In this manner a few "anarchic" program modules need not effect the utility of the module system for structuring the rest of the program.

Another form of useful assertion about a structure is to assert that it exists and has a given signature. Such a declaration can be used to provide virtual structures in order to run a partially formed program. The behavior on attempting to prove a goal expressed in the virtual language can be user-controlled as in existing PROLOG implementations.

5. RELATED WORK

Other authors have investigated modularity in the setting of logic programming in general, and PROLOG in particular.

The functional approach to modularity on which the above module system is based is similar to that of O'Keefe [10]. His bricks correspond to our structures. O'Keefe's mechanisms for (nonrecursive) abstraction correspond to the result of a functor declaration and application. O'Keefe's algebra is "untyped", however, in the sense that no restrictions are placed on the arguments to such functors. Here of course signatures play the role of "types". In addition, O'Keefe concentrates on predicate constants, whereas we extend the scope of the module language to function constants as well. This is not strictly necessary, but prevents certain types of programming errors as well as providing a form of structural data abstraction.

Miller [9] presents a theory of modularity for a logic programming language based on nested implication. Again only predicate constants are managed by the theory; function constants are considered global. A module consists of a named set of clauses similar to our notion of structure. A limited form of parametrization on predicate constants is possible, but this is not captured naturally within his system. Dependence between modules (the substructure relation) is represented in his system by nested implication. Miller's aim is to give a logical semantics to various programming notions. A semantics for the language presented above could be given along the lines suggested by Miller by viewing a structure as a theory. This should be clear from the translation semantics presented. We have chosen the latter form to illuminate possible implementation methods within existing PROLOG environments.

Goguen and Meseguer [4] also present a theory of modularity based on the Clear specification language for a (sorted) logic programming language of Horn clauses with equality. Their motivation is similar in many ways to ours, and their approach, like ours, comes equipped with a theory of formal program development. A technical difference which may be methodologically important is that their parametrization mechanism, based on the notion of "pushout" from category theory, does not permit functors in which the parameter structure is not inherited

by the structure which results from functor application. Another difference is that their approach is not formulated for the *standard* PROLOG language, and hence requires significant extensions to the interpreter.

Quintus PROLOG (Release 2.0) [12] is a practical PROLOG implementation equipped with a notion of module. The Quintus system can be seen as a simplified version of the system proposed here. The simplifications are as follows:

1. The Quintus system provides no parametrization constructs (i.e., no functors);
2. it manages the predicate structure of a program only (i.e., function constants are global);
3. it is nonhierarchical (i.e., no substructures).

The Quintus module system is seen mainly as a means of avoiding name clashes in large programs. The size of a typical module is assumed to be quite large. We take the view that a module system should actively support the construction of programs by managing the interactions between its components and ensuring a high degree of internal coherence.

Fitting [2] presents a semantic basis for logic programming modules based on recursion-theoretic enumeration operators. Each operator is a function that takes relations as input and returns relations as output. An operator represents a module. A small collection of basic operators and a set of operations (composition, product, etc.) are provided under which the class of enumeration operators is closed. Fitting's operators can be seen as a semantic counterpart to our functors. Functor application may then be interpreted via Fitting's composition operation. Indeed the semantics we have given to this operation (see Appendix A) closely follows Fitting's definition except that he uses an explicit "linking" clause, whereas we make use of an environment and rename constants. A simplified version of the calculus presented here can, we believe, be construed as a "proof theory" for Fitting's semantics in the concrete case of PROLOG. The semantics would need to be extended in order to provide a suitable interpretation for the full language presented here (e.g., to allow local function constants etc.). Our calculus is therefore quite complementary to Fitting's (and O'Keefe's) semantic prescriptions.

6. CONCLUSIONS

Most standard PROLOG systems lack a notion of module and mechanisms for data abstraction. This brings into question the utility of PROLOG for large, multiprogrammer projects. We have presented a module system for PROLOG based on a functional theory of modularity. The system supports the construction of large PROLOG programs from parametrized components and provides facilities for data abstraction. The system induces a notion of program well-formedness which correctly excludes many common PROLOG programming errors. The proposed interface to the extralogical facilities of PROLOG (Section 4) supports a hierarchical view of the PROLOG database. MacQueen's version of the module system for Standard ML is developed for a strongly typed language. The theory presented here could be extended trivially to apply to a typed PROLOG in the sense of [11].

Crucially though, with all these advantages, the module system requires minimal alteration of the existing PROLOG language. The system is a metalanguage, not a new programming language. We have provided a formal semantics that indicates how to translate a program written in modular PROLOG into its unstructured equivalent.

A word must be said about implementation. The semantics provided leads directly to an implementation on top of existing PROLOG systems via specialized `consult` and `reconsult` operations. In such an implementation multiple applications of a functor result in multiple copies of its body in the resulting program. This is not intolerable, since without the module system either the code would have to be present the same number of times (but with different predicate names) or predicate parameters and explicit `calls` would have to be employed. The latter solution is not widely employed in our experience (except perhaps for sorting), and is conceptually inelegant. The number of applications of a given functor is unlikely to be large. The advantage of functors is that they may be reused in different programs. A partial implementation along these lines has been constructed by Andrew Bowles, who has also investigated the integration of environmental tools (such as debugging aids) with the module language [1].

Such implementation problems can be overcome by appropriate separate compilation facilities, another tremendously important feature that existing PROLOG systems lack. Functors, which give a sound and elegant interpretation for the operation of relinking, can provide a basis for such facilities for PROLOG as they have done for Standard ML [6].

We have employed the same theory of modularity that underlies the Standard ML module system [7]. Consequently, the basis of our design is not original, and very little of the underlying theory is PROLOG-specific. This was intentional, and is seen as an advantage, since standard theories of program development employing this notion of module then apply directly to modular PROLOG [14]. These theories are deliberately couched in terms that do not proscribe the underlying programming language and logic [3, 13]. Our contribution has been to show how to instantiate the general theory to the PROLOG language in an interesting way so as to provide it with state of the art modules and data abstraction.

APPENDIX A. SEMANTICS

A.1. Values

Convention. We use the term "constant" to refer to both predicate constants and function constants when the distinction is unimportant.

We assume that there is an infinite supply of names available which are not accessible to the user (e.g. because they are not acceptable to the lexical analyser). These will be used as "internal" names of structures and constants.

A signature expression is denoted by a triple $sig = \langle substrs, preds, funs \rangle$ (called a *signature*), where:

substrs: substructure names → "internal" structure names,

preds: predicate constants → "internal" predicate names,

funs: function constants → "internal" function names.

All of these functions are finite maps (association lists). A predicate/function constant is a $\langle name, arity \rangle$ pair.

A *structure* (the denotation of a structure expression) is a pair $str = \langle timestamp, sig \rangle$ where:

timestamp is the "internal" name of this structure,

sig is the signature.

The structure *pervasives* contains the pervasive constants (such as the function constants `.:2` and `[]:0` and the predicate constants `=:2` and `true:0`). These are automatically a part of every signature and structure.

The structure component *timestamp* and the signature component *substrs* are needed to deal with structure sharing and the generative aspect of structure declaration. Since each elaboration of an encapsulated structure declaration or functor application creates a distinct structure, each structure must carry a distinct timestamp to distinguish it from structures which are identical but created separately. This also provides a means by which structures which are really the same can be identified. The component *substrs* gives the correspondence between substructure names and their timestamps; if a structure `A` in the structure environment has a timestamp t, then

```
struct
  ...
  structure B = A
  ...
end
```

creates a structure (with its own timestamp) where the substructure name `B` is associated with the timestamp t. Thus in a structure, *substrs* indicates the extent to which substructures share with each other and with external structures and substructures. In a signature, *substrs* only reflects internal substructure sharing, since sharing with external structures/substructures is not possible.

The signature components *preds* and *funs* are needed to deal with sharing of predicates constants and function constants. Each constant is mapped to an internal name which uniquely identifies it. This is the name which will be used in the code which the semantics produces as part of the denotation of a program. If two constants have the same internal name, then they are the same (they share). A single predicate constant will have multiple names when it belongs simultaneously to multiple substructures of a given structure; then it might have names `A/n` and `B/C/n`, say. The same may happen with function constants; in addition, the construct

`fun` *atid*: *nat* = *id*.

can be used to give a new name to a function constant.

A signature includes constants defined at "top level" within the signature/structure as well as constants belonging to substructures. Constants belonging to a substructure `A` have names of the form `A` / n; furthermore, every type or value having a name of this form is regarded as a part of `A`.

This is not the only possible way of representing signature/structure values, although any alternative representation must take proper account of the complications mentioned above (generative structure declarations, structure sharing, and multiple names for a single constant).

A *functor* is a 6-tuple $fun = \langle params, sig, strexpr, \rho, \psi, \pi \rangle$ where:

params is the *atid* list (the formal parameter names),

sig is the signature (combined formal parameters with sharing taken into account),

strexpr is the structure expression (the body of the functor),

ρ, ψ, π are the structure, signature, and functor environments at the point of declaration.

The structure, signature, and functor environments are finite maps $\rho : atid \rightarrow$ structure, $\psi : atid \rightarrow$ signature, and $\pi : atid \rightarrow$ functor. Functors are treated as macros which are expanded in the declaration time environment.

The structure environment includes bindings of structures occurring earlier than the construct currently being elaborated, as well as (if the current construct is a structure) bindings of its substructures. The latter is necessary because in a nested context a substructure of the current structure is just like a previously defined structure.

A.2. Semantic Operations

A.2.1. Fitting a Structure to a Signature. $Fit(str, sig)$ checks if the candidate structure *str* matches the target signature *sig*; if it does, then the structure which results from restricting *str* to *sig* is returned. The third and fourth error checks may be relatively expensive in a naive implementation, since they involve examining every pair of predicate constants and function constants in *sig*. However, if two structures A and B share, then every pair of constants $A/n, B/n$ shares. This means that if the second error check succeeds, then some of the pairs of constants in *sig* need not be checked:

$fit : structure \times signature \rightarrow structure$

$fit(\langle tag, \langle substrs, preds, funs \rangle \rangle, \langle substrs', preds', funs' \rangle) =$
  let tag' be an unused internal structure name in
    $\langle tag', \langle substrs \dagger dom(substrs'), preds \dagger dom(preds'), funs \dagger dom(funs') \rangle \rangle$
  error if $dom(substrs') \not\subseteq dom(substrs)$, $dom(preds') \not\subseteq dom(preds)$,
      or $dom(funs') \not\subseteq dom(funs)$
or if $\exists n, m \in dom(substrs').substrs'(n) = substrs'(m)$ and $substrs(n) \neq substrs(m)$
or if $\exists n, m \in dom(preds').preds'(n) = preds'(m)$ and $preds(n) \neq preds(m)$
or if $\exists n, m \in dom(funs').funs'(n) = funs'(m)$ and $funs(n) \neq funs(m)$

A.2.2. Generating New Internal Names for Constants. $Tag(sig)$ is the signature which results from changing the internal names of nonpervasive constants and substructures in *sign* to make them distinct from all other internal names. This is necessary to ensure that undesired sharing does not arise:

$tag : signature \to signature$

$tag(\langle substrs, preds, funs\rangle) =$
  let $\langle psubstrs, ppreds, pfuns\rangle = pervasives$ in
  let $subtag = \{tag \mapsto tag' \mid tag \in range(substrs) - range(psubstrs)$
    and tag' is a (different) unused internal structure name for each $tag\}$
    $\cup \{tag \mapsto tag \mid tag \in range(psubstrs)\}$ in
  let $predtag = \{tag \mapsto tag' \mid tag \in range(preds) - range(ppreds)$
    and tag' is a (different) unused internal predicate name for each $tag\}$
    $\cup \{tag \mapsto tag \mid tag \in range(ppreds)\}$ in
  let $funtag = \{tag \mapsto tag' \mid tag \in range(funs) - range(pfuns)$
    and tag' is a (different) unused internal function name for each $tag\}$
    $\cup \{tag \mapsto tag \mid tag \in range(pfuns)\}$ in
  $\langle substrs \cdot subtag, preds \cdot predtag, funs \cdot funtag\rangle$

A.2.3. Identifying Substructures in a Signature. $Identify(id, id', sig)$ is the signature which results from identifying the internal names of the substructures named *id* and *id'*. A new internal name is chosen for the substructure in the result. All of the corresponding constants in these substructures are also identified. $Identify(\{\langle id_1, id_1'\rangle, \ldots, \langle id_n, id_n'\rangle\}, sig)$ is just

$$identify(id_1, id_1', identify(id_2, id_2', \ldots, identify(id_n, id_n', sig)\ldots));$$

note that the order here is immaterial.

$identify : id \times id \times signature \to signature$

$identify(id, id', \langle substrs, preds, funs\rangle) =$
  let tag be an unused internal structure name in
  let $ppairs = \{\langle id/p, id'/p\rangle \mid id/p \in dom(preds)\}$ in
  let $fpairs = \{\langle id/f, id'/f\rangle \mid id/f \in dom(funs)\}$ in
  let $joinsub = \{tag' \mapsto tag' \mid tag' \in range(substrs)\}$
    $[substrs(id) \mapsto tag, substrs(id') \mapsto tag]$ in
  $identify\text{-}preds(ppairs, identify\text{-}funs(fpairs, \langle substrs \cdot joinsub, preds, funs\rangle))$
  error if $\{id/p \in dom(preds)\} \neq \{id'/p \in dom(preds)\}$
    or if $\{id/f \in dom(funs)\} \neq \{id'/f \in dom(funs)\}$

Identify-preds and *identify-funs* are defined similarly.

A.2.4. Extracting a Substructure from a Signature / Structure. $Substructure(id, sig)$ is the structure corresponding to the substructure *id* of *sig*. The

structure *substructure*(*id*, *str*) is just *substructure*(*id*, *sig*[*str*]).

substructure : *id* × *signature* → *structure*

substructure(*id*, ⟨*substrs*, *preds*, *funs*⟩) =
  let *substrs'* = {*n* ↦ *tag* | (*id*/*n* ↦ *tag*) ∈ *substrs*} in
  let *preds'* = {*n* ↦ *tag* | (*id*/*n* ↦ *tag*) ∈ *preds*} in
  let *funs'* = {*n* ↦ *tag* | (*id*/*n* ↦ *tag*) ∈ *funs*} in
  ⟨*substrs*(*id*), ⟨*substrs'*, *preds'*, *funs'*⟩⟩

A.2.5. Adding New Substructures to a Structure / Signature. *Addsubstrs*({⟨$atid_1$, str_1⟩, ..., ⟨$atid_n$, str_n⟩}, *sig*) is the signature which results from adding the structure $str_1, \ldots, str_n$ to *sig* as substructures named $atid_1, \ldots, atid_n$ respectively. *Addsubstrs*(*S*, *str*) is the structure ⟨*tag*, *addsubstrs*(*S*, *sig*[*str*])⟩, where *tag* is an unused internal structure name. Finally, *addsubstrs*({⟨$atid_1$, sig_1⟩, ..., ⟨$atid_n$, sig_n⟩}, *sig*) is just

addsubstrs({⟨$atid_1$, ⟨tag_1, sig_1⟩⟩, ..., ⟨$atid_n$, ⟨tag_n, sig_n⟩⟩}, *sig*)

where $tag_1, \ldots, tag_n$ are unused internal structure names.

addsubstrs : (*atid* × *structure*)-*set* × *signature* → *signature*

addsubstrs({⟨$atid_1$, ⟨tag_1, ⟨$substrs_1$, $preds_1$, $funs_1$⟩⟩⟩, ...,
  ⟨$atid_n$, ⟨tag_n, ⟨$substrs_n$, $preds_n$, $funs_n$⟩⟩⟩},
  ⟨*substrs*, *preds*, *funs*⟩) =
let *substrs'* = {$atid_1 \mapsto tag_1, \ldots, atid_n \mapsto tag_n$}
  ∪ ∪$_{i \le n}$ {$atid_i$/*id* ↦ *tag* | (*id* ↦ *tag*) ∈ $substrs_i$} in
let *preds'* = ∪$_{i \le n}$ {$atid_i$/*id* ↦ *tag* | (*id* ↦ *tag*) ∈ $preds_i$} in
let *funs'* = ∪$_{i \le n}$ {$atid_i$/*id* ↦ *tag* | (*id* ↦ *tag*) ∈ $funs_i$} in
⟨*substrs* ∪ *substrs'*, *preds* ∪ *preds'*, *funs* ∪ *funs'*⟩
  error if $atid_i \in dom(substrs)$ for some *i*

A.3. Semantic Functions

Prog : *prog*
  → *structure-environment* → *signature-environment* → *functor-environment*
  → (*structure* × *code*)
Sigb : *sigb*
  → *signature-environment*
  → (*atid* × *signature*)
Funb : *funb*
  → *structure-environment* → *signature-environment* → *functor-environment*
  → (*atid* × *functor*)
Plist : *plist*
  → *signature-environment*
  → (*atid-list* × *signature*)
Patheq : *patheq*
  → *signature*
  → (*id* × *id*)-*set*

Strb : *strb*
→ *signature*
→ *structure-environment* → *signature-environment* → *functor-environment*
→ (*atid* × *structure* × *code*)
Sigexpr : *sigexpr*
→ *signature-environment*
→ *signature*
Spec : *spec*
→ *signature*
→ *signature-environment*
→ *signature*
Specstrb : *specstrb*
→ *signature*
→ *signature-environment*
→ (*atid* × *signature*)
Strexpr : *strexpr*
→ *structure-environment* → *signature-environment* → *functor-environment*
→ (*structure* × *code*)
Dec : *dec*
→ *signature*
→ *structure-environment* → *signature-environment* → *functor-environment*
→ (*signature* × *structure-environment* × *signature-environment*
× *functor-environment* × *code*)
Fun : *id*
→ *nat*
→ *signature*
→ *structure-environment*
→ *internal function name*

A.4. Semantic Equations

The result of *Prog* is a structure containing all the (top-level) bindings introduced by the program, together with the code (a sequence of Horn clauses) produced by structure declarations and top-level clauses in the program. To compile a program *prog* in the initial environment $\langle \rho_0, \psi_0, \pi_0 \rangle$ and then evaluate a goal:

compute $Prog[\![prog]\!]\rho_0\psi_0\pi_0$, obtaining a structure *str* and some PROLOG code;

compile the code in PROLOG;

translate the goal by replacing each constant with its internal name according to *preds*[*str*] and *funs*[*str*]; and

evaluate the resulting goal using PROLOG.

The semantics below gives an error if a constant is used without ever being defined. (Note that there is no requirement that constants be defined *before* use.)

In practice it might be desirable to relax this rule so that using top-level predicate constants which are not defined results in failure at execution time (in case that predicate constant is encountered during an attempt to satisfy a goal) rather than at compile time, which is what happens in PROLOG at present. It is not so clear whether such a relaxed rule should apply to function constants as well.

$$Prog[\![dec]\!]\rho\psi\pi = Strexpr[\![\texttt{struct}\ dec\ \texttt{end}]\!]\rho\psi\pi$$

$$Sigb[\![atid = sigexpr]\!]\psi = \langle atid, Sigexpr[\![sigexpr]\!]\psi\rangle$$

Functors are treated as marcros in this semantics, in the sense that the body of a functor is kept as a syntactic object rather than as some sort of parametrized structure. However, the parameter declaration is processed at definition time, and the functor body is checked to ensure that it is well formed and that any application will produce a valid structure. The environment at declaration time must be saved for use at application time so that identifiers in the functor body can be interpreted.

A functor with several parameters is treated as a functor with a single parameter having a substructure of the appropriate name for each of the several parameters. In checking whether applications of the functor will produce valid structures, the functor body is elaborated in a structure environment augmented by binding the formal parameter names to the parameter signatures. Note that the signature of the final result of this process differs from the declared signature (if any) in that it shares constants with the parameter signature in a way which reflects the references which the functor body makes to the formal parameters.

$$Funb[\![atid(plist) = strexpr]\!]\rho\psi\pi =$$
$$\quad \text{let } \langle atid_1 \ldots atid_n, sig\rangle = Plist[\![plist]\!]\psi \text{ in}$$
$$\quad \langle atid, \langle atid_1 \ldots atid_n, sig, strexpr, \rho, \psi, \pi\rangle\rangle$$
$$\quad error \text{ if } Strexpr[\![strexpr]\!]\rho'\psi\pi \text{ fails}$$
$$\quad \text{where } \rho' = \rho[atid_1 \mapsto substructure(atid_1, sig), \ldots, atid_n \mapsto substructure(atid_n, sig)]$$

$$Plist[\![atid_1 : sig_1, \ldots, atid_n : sig_n\ [\texttt{sharing}\ patheq_1\ \texttt{and}\ \ldots\ \texttt{and}\ patheq_m]]\!]\psi =$$
$$\quad \text{let } sig = Spec[\![\texttt{structure}\ atid_1 : sig_1\ \texttt{and}\ \ldots\ \texttt{and}\ atid_n : sig_n$$
$$\qquad [\texttt{sharing}\ patheq_1\ \texttt{and}\ \ldots\ \texttt{and}\ patheq_m]]\!]sig[pervasives]\psi \text{ in}$$
$$\quad \langle atid_1 \ldots atid_n, sig\rangle$$

$$Patheq[\![id_1 = \cdots = id_n]\!]sig = \{\langle id_1, id_j\rangle \mid 2 \le j \le n\}$$
$$\quad error \text{ if } id_i \notin substrs[sig] \text{ for some } i$$

A structure binding

$$atid = strexpr$$

has the effect of adding a substructure called *atid* to the current signature (by adding bindings of all the constants in *strexpr*, with their names prefixed by *atid*) as well as to the structure environment. The result of *Strb* is the identifier *atid*, the structure to which it is to be bound and the code generated while elaborating *strexpr*. There must not be a constant in the current signature with a name of the

form *atid / n*, since this would cause it to be regarded as a part of the new substructure.

$Strb[\![atid = strexpr]\!] sig\ \rho\psi\pi =$
  let $\langle str, code\rangle = Strexpr[\![strexpr]\!]\rho\psi\pi$ in
  $\langle atid, str, code\rangle$
  error if $atid \in dom(\rho)$

$Sigexpr[\![atid]\!]\psi = \psi(atid)$
  error if $atid \notin dom(\psi)$

$Sigexpr[\![$ `sig` *spec* `end` $]\!]\psi = Spec[\![spec]\!]sig[pervasives]\psi$

$Spec[\![$ `pred` *atid* : *nat* $.]\!]\langle substrs, preds, funs\rangle\psi =$
  let *tag* be an unused internal predicate name in
  $\langle substrs, preds \cup \{\langle atid, nat\rangle \mapsto tag\}, funs\rangle$
  error if $\langle atid, nat\rangle \in dom(preds)$

$Spec[\![$ `fun` *atid* : *nat* $.]\!]\langle substrs, preds, funs\rangle\psi =$
  let *tag* be an unused internal function name in
  $\langle substrs, preds, funs \cup \{\langle atid, nat\rangle \mapsto tag\}\rangle$
  error if $\langle atid, nat\rangle \in dom(funs)$

Constants declared in a structure binding contribute to the current environment, with names prefixed by the name of the (sub)structure in which they appear.

$Spec[\![$ `structure` $specstrb_1$ `and` ... `and` $specstrb_n]\!]\ sig\ \psi =$
  let $\langle atid_1, sig_1\rangle, \ldots, \langle atid_n, sig_n\rangle =$
    $Specstrb[\![specstrb_1]\!]sig\ \psi, \ldots, Specstrb[\![specstrb_n]\!]sig\ \psi$ in
  $addsubstrs(\{\langle atid_1, sig_1\rangle, \ldots, \langle atid_n, sig_n\rangle\}, sig)$
  error if $atid_i = atid_j$ for some $i \neq j$
    or if $atid_i \in dom(substrs[sig])$ for some i

$Spec[\![$ `structure` $specstrb_1$ `and` ... `and` $specstrb_n$
    `sharing` $patheq_1$ `and` ... `and` $patheq_m]\!]sig\ \psi =$
  let $sig' = Spec[\![$ `structure` $specstrb_1$ `and` ... `and` $specstrb_n]\!]sig\ \psi$ in
  $identify(Patheq[\![patheq_1]\!]sig' \cup \cdots \cup Patheq[\![patheq_m]\!]sig', sig')$

$Spec[\![spec\ spec']\!]sig\ \psi =$
  let $sig' = Spec[\![spec]\!]sig\ \psi$ in
  $Spec[\![spec']\!]sig'\ \psi$

A structure binding in a signature context of the form

atid : *sigexpr*

adds a substructure called *atid* containing the constants in *sigexpr* to the current environment. These are forced to be distinct from all the constants already present, with the exception of pervasive constants. That this is necessary is shown by the example declaration

`structure` *A* : *sigexpr* `and` *B* : *sigexpr*

since *A / n* is not expected to share with *B / n* for a constant *n* in *sigexpr* (unless

e.g. n = `true : 0`).

$Specstrb[\![atid : sigexpr]\!]sig\ \psi = \langle atid, tag(Sigexpr[\![sigexpr]\!]\psi)\rangle$
error if $atid \in dom(substrs[sig])$

$Strexpr[\![atid]\!]\rho\psi\pi = \langle \rho(atid), \emptyset\rangle$
error if $atid \notin dom(\rho)$

$Strexpr[\![atid / id]\!]\rho\psi\pi = \langle substructure(id, \rho(atid)), \emptyset\rangle$
error if $atid \notin dom(\rho)$

$Strexpr[\![strexpr : sigexpr]\!]\rho\psi\pi =$
let $\langle str, code\rangle = Strexpr[\![strexpr]\!]\rho\psi\pi$ in
$\langle fit(str, Sigexpr[\![sigexpr]\!]\psi), code\rangle$

$Strexpr[\![$`struct` dec `end`$]\!]\rho\psi\pi =$
let $\langle sig, \rho', \psi', \pi', code\rangle = Dec[\![dec]\!]sig[pervasives]\rho\psi\pi$ in
let *tag* be an unused internal structure name in
let *code'* be the result of translating *code* by replacing each constant with its internal name (according to $preds[sig]$ and $funs[sig]$) in
$\langle\langle tag, sig\rangle, code'\rangle$
error if some previously untranslated predicate symbol in *code* is not in $preds[sig]$
or if some previously untranslated function symbol in *code* is not in $funs[sig]$

The result of applying a functor to a list of actual parameters is obtained by elaborating the body of the functor in the declaration time environment augmented by binding the parameter names to the actual parameters (after fitting them to the formal parameter signatures).

$Strexpr[\![atid(strexpr_1, \ldots, strexpr_n)]\!]\rho\psi\pi =$
let $\langle atid_1 \ldots atid_n, sig, strexpr, \rho', \psi', \pi'\rangle = \pi(atid)$ in
let $\langle str_1, code_1\rangle, \ldots, \langle str_n, code_n\rangle =$
$Strexpr[\![strexpr_1]\!]\rho\psi\pi, \ldots, Strexpr[\![strexpr_n]\!]\rho\psi\pi$ in
let $str = fit(addsubstrs(\{\langle atid_1, str_1\rangle, \ldots, \langle atid_n, str_n\rangle\}, pervasives), sig)$ in
let $\langle str', code\rangle = Strexpr[\![strexpr]\!]\rho'[atid_1, \ldots, \mapsto substructure(atid_1, str), \ldots,$
$atid_n \mapsto substructure(atid_n, str)]\psi'\pi'$ in
$\langle str', code_1.\cdots.code_n.code\rangle$
error if $atid \notin dom(\pi)$
or if $n \neq m$

$Dec[\![atid(term_1, \ldots, term_n)[:\!- atom_1, \ldots, atom_m].]\!]\langle substrs, preds, funs\rangle\rho\psi\pi =$
if $\langle atid, n\rangle \in dom(preds)$
then $(\langle substrs, preds, funs\rangle, \rho, \psi, \pi,$
$\{atid(term_1, \ldots, term_n)[:\!- atom_1, \ldots, atom_m].\}\rangle$
else let *tag* be an unused inernal predicate name in
$\langle\langle substrs, preds \cup \{\langle atid, n\rangle \mapsto tag\}, funs\rangle, \rho, \psi, \pi,$
$\{atid(term_1, \ldots, term_n)[:\!- atom_1, \ldots, atom_m].\}\rangle$

$Dec[\![$`fun` $atid : nat$ `.`$]\!]\langle substrs, preds, funs\rangle\rho\psi\pi =$
let *tag* be an unused internal function name in
$\langle\langle substrs, preds, funs \cup \{\langle atid, nat\rangle \mapsto tag\}\rangle, \rho, \psi, \pi, \emptyset\rangle$
error if $\langle atid, nat\rangle \in dom(funs)$

$Dec[\![\texttt{fun}\ atid : nat = id\ .]\!]\langle substrs, preds, funs\rangle\rho\psi\pi =$
$\langle\langle substrs, preds, funs \cup \{\langle atid, nat\rangle \mapsto Fun[\![id]\!]nat\langle substrs, preds, funs\rangle\rho\}\rangle, \rho, \psi, \pi, \emptyset\rangle$
error if $\langle atid, nat\rangle \in dom(funs)$

$Dec[\![\texttt{open}\ id]\!]sig\ \rho\psi\pi =$
let $\langle\langle tag', sig'\rangle, code'\rangle = Strexpr[\![id]\!]\rho\psi\pi$ in
$\langle sig \cup sig', \rho \cup \bigcup_{atid \in dom(substrs[sig'])}\{atid \mapsto substructure(atid, sig')\}, \psi, \pi, \emptyset\rangle$
error if $dom(\rho) \cap dom(substrs[sig']) \neq \emptyset$
or if *sig* and *sig'* have substructure or (nonpervasive) constant names in common

Constants declared in a structure binding contribute to the environment of current bindings. The newly declared structure also contributes to the structure environment for the benefit of nested encapsulated structure declarations, to which it appears as a previously defined structure. Sharing constraints are not permitted in structure contexts; sharing in a structure arises by construction rather than by declaration.

$Dec[\![\texttt{structure}\ strb_1\ \texttt{and}\ \ldots\ \texttt{and}\ strb_n]\!]sig\ \rho\psi\pi =$
let $\langle atid_1, str_1, code_1\rangle, \ldots, \langle atid_n, str_n, code_n\rangle = Strb[\![strb_1]\!]sig\ \rho\psi\pi, \ldots, Strb[\![strb_n]\!]\ sig\ \rho\psi\pi$ in
$\langle addsubstrs(\{\langle atid_1, str_1\rangle, \ldots, \langle atid_n, str_n\rangle\}, sig),$
$\rho \cup \{atid_1 \mapsto str_1, \ldots, atid_n \mapsto str_n\}, \psi, \pi, code_1.\cdots.code_n\rangle$
error if $atid_i = atid_j$ for some $i \neq j$
or if $atid_i \in dom(\rho)$ for some i

$Dec[\![\texttt{signature}\ sigb_1\ \texttt{and}\ \ldots\ \texttt{and}\ sigb_n]\!]sig\ \rho\psi\pi =$
let $\langle atid_1, sig_1\rangle, \ldots, \langle atid_n, sig_n\rangle = Sigb[\![sigb_1]\!]\psi, \ldots, Sigb[\![sigb_n]\!]\psi$ in
$\langle sig, \rho, \psi \cup \{atid_1 \mapsto sig_1, \ldots, atid_n \mapsto sig_n\}, \pi, \emptyset\rangle$
error if $atid_i = atid_j$ for some $i \neq j$
or if $atid_i \in dom(\psi)$ for some i

$Dec[\![\texttt{functor}\ funb_1\ \texttt{and}\ \ldots\ \texttt{and}\ funb_n]\!]sig\ \rho\psi\pi =$
let $\langle atid_1, fun_1\rangle, \ldots, \langle atid_n, fun_n\rangle = Funb[\![funb_1]\!]\rho\psi\pi, \ldots, Funb[\![funb_n]\!]\rho\psi\pi$
in
$\langle sig, \rho, \psi, \pi \cup \{atid_1 \mapsto fun_1, \ldots, atid_n \mapsto fun_n\}, \emptyset\rangle$
error if $atid_i = atid_j$ for some $i \neq j$
or if $atid_i \in dom(\pi)$ for some i

$Dec[\![dec\ dec']\!]sig\ \rho\psi\pi =$
let $\langle sig', \rho'\psi', \pi', code'\rangle = Dec[\![dec]\!]sig\ \rho\psi\pi$ in
let $\langle sig'', \rho'', \psi'', \pi'', code''\rangle = Dec[\![dec']\!]sig'\ \rho'\psi'\pi'$ in
$\langle sig'', \rho'', \psi'', \pi'', code'.code''\rangle$

The function *Fun* is used to interpret a function names. It returns the internal name of the function constant referenced.

$Fun[\![atid]\!]nat\langle substrs, preds, funs\rangle\rho = funs(\langle atid, nat\rangle)$
error if $\langle atid, nat\rangle \notin dom(funs)$

$$
\begin{aligned}
&Fun[\![atid \,/\, id]\!] nat\,sig\ \rho = \\
&\quad \text{let } \langle substrs', preds', funs' \rangle = \rho(atid) \text{ in} \\
&\quad funs'(\langle id, nat \rangle) \\
&\qquad error \text{ if } atid \notin dom(\rho) \\
&\qquad\quad \text{or if } \langle id, nat \rangle \notin dom(funs')
\end{aligned}
$$

Thanks to David MacQueen for developing the module system for Standard ML on which the system we describe in this paper is based. Thanks to James Harland for helpful comments on a draft of this paper, and to Ralph Hasselgren for correcting some errors in an earlier version of the semantics.

REFERENCES

1. Bowles, A., Enhancing Prolog Programming Environments, Master's Thesis, Univ. of Edinburgh, 1987.
2. Fitting, M. C., Enumeration Operators and Modular Logic Programming, *J. Logic Programming* 4:11–21 (1987).
3. Goguen, J. A. and Burstall, R. M., Introducing Institutions, in: *Proceedings of the Logics of Programming Workshop, Carnegie-Mellon*, Lecture Notes in Comput. Sci. 164, 1984, pp. 221–256.
4. Goguen, J. A. and Meseguer, J., Eqlog: Equality, Types and Generic Modules for Logic Programming, in: DeGroot and Lindstrom (eds.), *Functional and Logic Programming*, Prentice-Hall, 1985.
5. Harper, R., Introduction to Standard ML, Report ECS-LFCS-86-14, Dept. of Computer Science, Univ. of Edinburgh, 1986.
6. Harper, R., Modules and Persistence in Standard ML, Report ECS-LFCS-86-11, Dept. of Computer Science, Univ. of Edinburgh, 1986.
7. Harper, R., MacQueen, D., and Milner, R., Standard ML, Report ECS-LFCS-86-2, Dept. of Computer Science, Univ. of Edinburgh, Mar. 1986.
8. Milner, R., Tofte, M., and Harper, R., *The Definition of Standard-ML*, MIT Press, 1990.
9. Miller, D. A., A Theory of Modules for Logic Programming, in: *IEEE Symposium on Logic Programming*, 1986.
10. O'Keefe, R. A., Towards an Algebra for Constructing Logic Programs, in: *IEEE Symposium on Logic Programming*, 1985, pp. 152–160.
11. O'Keefe, R. A. and Mycroft, A., A Polymorphic Type System for Prolog, *Artifi. Intell.* 23(3):295–307 (Aug. 1983).
12. *Quintus Prolog User's Guide*, Mountain View, Calif., version 10 ed., 1987.
13. Sannella, D. T. and Tarlecki, A., Extended ML: An Institution-Independent Framework for Formal Program Development, in: *Proceedings of the Workshop on Category Theory and Computer Programming, Guildford*, Springer Lecture Notes in Comput. Sci. 240, Springer-Verlag, 1985, pp. 364–389.
14. Sannella, D. T. and Tarlecki, A., Program Specification and Development in Standard ML, in: *12th ACM Symposium on Principles of Programming Languages*, ACM, 1985, pp. 67–77.