An Approach to Modularity and Separate Compilation in Logic Programming

Steven Holte
Department of Computer Science
Brown University
sholte@gmail.com

Gopalan Nadathur
Computer Science and Engineering
University of Minnesota
gopalan@cs.umn.edu

Abstract

The ability to compose code in a modular fashion is important to the construction of large programs. In the logic programming setting, it is desirable that such capabilities be realized through logic-based devices. We present here an approach to doing this that is a supported in the *Teyjus* implementation of the λProlog language. Within this scheme, a module corresponds to a block of code whose external view is mediated by a signature. Thus, signatures impose a form of hiding that is explained logically via existential quantifications over predicate, function and constant names. Modules interact through the mechanism of *accumulation* that translates into conjoining the clauses in them while respecting the scopes of existential quantifiers introduced by signatures. We show that this simple device for statically structuring name spaces suffices for realizing features related to code scoping for which the dynamic control of predicate definitions was earlier considered necessary. While a compile-time inlining of accumulated modules that respects quantifier scoping can be used to realize the module capabilities we present in a transparently correct way, such an approach has the drawback of not supporting separate compilation. We show how this approach can be refined into a scheme that allows each distinct module to be compiled separately, with the effect of inlining being realized by a subsequent linking process.

Categories and Subject Descriptors D.1.6 [*Programming Techniques*]: Logic Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic

General Terms logic programming, modularity constructs, separate compilation

Keywords λProlog, modules, signatures, existential quantification, proof search, Warren Abstract Machine, compilation, linking

1. Introduction

We are concerned in this paper with a treatment of modularity in logic programming. Support for this feature is important to significant applications of the paradigm. The ability to develop a system through the composition of small, well-defined units of code is central to managing complexity and also facilitates the reuse of programming effort. Moreover, modular development installs boundaries in programs that can be important to the practical use of static analysis techniques and that are fundamental to the notion of separate compilation and testing. In light of these facts, it is not surprising that modularity aspects have received significant attention with respect to most existing programming paradigms. As they pertain specifically to logic programming, these notions have received theoretical treatment [9, 12, 17, 21], have been included in practical systems [2, 23, 24] and have been the topic of standardization deliberations concerning Prolog [5].

When incorporating modularity notions into logic programming systems, a common trend has been to go outside the logical base and to introduce metalinguistic mechanisms for composing separately constructed program fragments. This approach is a little unfortunate: a strength of logic programming is its basis in logic that, with proper choices, can also be used to reason about interactions between units of code [13]. However, there is also a danger in focussing too heavily on just the logical aspects. Early logic-based approaches to controlling code availability have, for instance, utilized the idea of implications or contexts in goals [12, 17]. These mechanisms call for a dynamic management of predicate definitions and, as such, their implementation is costly. Moreover, the particular way in which context is handled leads sometimes to program behaviour that is counter to the practical understanding of modularity.

We describe a treatment of modularity in this paper that balances logical and pragmatic considerations. This treatment draws on experience with a logic based approach [13] employed in the *Teyjus* implementation [19] of the language λProlog. However, our ideas are quite general and can be used in any logic programming setting that correctly implements sequences of alternating existential and universal quantifiers in goals.[1] The devices we use are, in fact, surprisingly simple at a logical level. To support information hiding, we utilize existential quantification over names. Pragmatically, the hiding of names is effected by associating a signature with each module of code; all the names used in the module and not appearing in the signature are treated as being implicitly existentially quantified. The composition of units of code, accomplished via a mechanism known as *accumulation*, translates into the conjoining of formulas. This leads to a *statically determined* code space but one in which the availability of predicate definitions can be controlled by appropriately scoped existential quantifiers. We show that these simple devices suffice for realizing features such as scoping of predicate definitions, data abstraction and module parameter-

[1] Such a capability can be added to common logic programming languages by changing the unification computation as indicated later.

ization for which more complicated dynamic code structuring capabilities were previously thought to be necessary [12, 17]. A noteworthy point is that the often problematic aspect of higher-order programming coexists *completely naturally* with this approach to modularity. From a implementation perspective, accumulation can be treated through a compile-time inlining of code [20]. However, true modularity requires support for separate compilation. Our second contribution consists of showing that this can be provided. In particular, we describe a scheme that permits each module to be compiled separately with the inlining function being relegated to a later, link-time process. This two phase process, which has been implemented in a version of the *Teyjus* system released in April 2008 [6], cumulatively expends effort similar to the compile-time inlining method, involves an extra linking time that is clearly acceptable and produces an identical executable image.

The rest of the paper is structured as follows. In the next section we introduce a logical language that includes all the features needed to capture our treatment of modularity. In Section 3 we describe the main components of the modules language and indicate their translation into the logical core. The following section describes embellishments in the form of annotations or compiler directives to the basic modules language; these directives do not affect the translation into logic but can be used to control acceptability of programs in pragmatically meaningful ways and may also lead to more efficient implementations. In Section 5 we consider the issue of separate compilation. Section 6 concludes the paper by contrasting its contents with related work.

2. The Underlying Logical Language

Our approach to information hiding involves the use of existential quantification. Since we desire the ability to hide the names of predicates and functions in addition to (first-order) constants, the right context for our ideas is that of a higher-order logic. We shall assume also that our language is typed although our main ideas apply equally in an untyped setting.

2.1 The Logical Syntax

We work with a set of types that initially contains *int*, *real*, *string* and *o*, those that can be formed using the unary type constructor *list* and, finally, all the function types, written as $\alpha \rightarrow \beta$, that can be formed from these types. The type o is that of propositions. Type variables, denoted by tokens starting with uppercase letters, are introduced as a shorthand for an infinite collection of instance types. Terms in the language are constructed from collections of typed constant and variable symbols using the operations of application and abstraction. We assume an initial set of constants that are partitioned into the *logical* ones, which are accorded a special interpretation in the language, and the *nonlogical* ones whose introduction is motivated by programming convenience. The logical constants consist of the symbols *true* that denotes the always true proposition, $\supset$, $\vee$ and $\wedge$ that denote infix forms of implication, disjunction and conjunction and two 'schema' constants *sigma* and *pi* of type $(A \rightarrow o) \rightarrow o$. The last two (family of) constants represent generalized existential and universal quantifiers. In particular, $\exists x\, P(x)$ and $\forall x\, P(x)$ are rendered in this logic as $(sigma\ \lambda x P(x))$ and $(pi\ \lambda x P(x))$ respectively; we write expressions of the form $P(x)$ here and elsewhere to denote terms that possibly have free occurrences of the variable x in them. The set of nonlogical constants that are available at the outset—referred to as the *pervasive* constants—is implementation dependent, but we shall assume this collection to include the usual constants denoting integers, reals and strings and the schema constants *nil* of type *list A* and the infix $::$ of type $A \rightarrow (list\ A) \rightarrow (list\ A)$ that provide for a builtin notion of lists. Application is assumed to be left associative. This leads to a curried notation for terms. If p is an n-ary relation symbol, the expression $(p\ t_1\ \ldots\ t_n)$ denotes this relation between the terms $t_1, \ldots, t_n$. Such an expression constitutes an atomic formula if its 'head' p is either *true* or is not a logical constant.

In any real application it is necessary to introduce new sorts and type constructors as well as new nonlogical constants. We describe mechanisms in the next section for declaring such symbols. Types must also be associated with variables. Such type information can be filled in by an inference process and the language syntax may also allow it to be explicitly provided.

The logic that we shall use to expose our ideas is a slightly expanded version of the higher-order theory of Horn clauses. The main expressions that are treated by this logic are the G and E formulas identified by the following syntax rules:

$$\begin{array}{lcl} G & ::= & true \mid A \mid G \vee G \mid G \wedge G \mid \exists x\, G \mid E \supset G \\ D & ::= & A_r \mid G \supset A_r \mid \forall x\, D \\ E & ::= & D \mid E \wedge E \mid \exists x\, E \end{array}$$

A denotes atomic formulas here and A_r represents atomic formulas whose heads are either constants distinct from *true* or are captured by an existential quantifier in an E formula that forms an enclosing context. Existential quantifiers in E formulas are treated as constants in the computation model described below. From this it becomes clear that the head of an A_r formula is always a nonlogical constant, albeit of smaller or larger scope. A D formula of the form A_r or $G \supset A_r$ is said to have A_r as its head and its body is either empty or G depending on the case in question. Such a formula is, as usual, part of the definition of a predicate whose name is the head of the A_r formula.

2.2 The Notion of Computation

Computation in the logical language is oriented around trying to solve a *goal* given by a G formula relative to a *program* that is determined by a set of D formulas. An important facet of the language is that it embodies a careful accounting of the set of constants that are available for forming terms; this set is also known as a signature.

Formally, (the main notion of) a state in a computation is represented by a sequent of the form $\Sigma; \mathcal{P} \longrightarrow G$ in which Σ is a signature and $\mathcal{P}$ and G are, respectively, a set of closed D formulas and a closed G formula that use only constants from Σ. We shall refer to $\mathcal{P}$ in such a sequent as the *program*. At the outset, Σ consists of the logical constants, the pervasive constants and a user-defined collection of global constants and $\mathcal{P}$ corresponds to builtin definitions for some of the pervasive predicates. The transition rules in Figure 1 define the process by which computations are carried out in this setting. The *instan* and $\exists G$ rules here have the proviso that t is a closed term that is formed using constants from Σ different from $\supset$ and pi and the $\exists E$ rule has the proviso that c is a constant that is new to Σ. Expressions of the form $F[t/x]$ that are used in these rules represent the substitution of the term t for the free occurrences of x in the formula F.

The transition rules can be understood to be of one of three kinds. The main category of rules are the ones having lower sequents of the form $\Sigma; \mathcal{P} \longrightarrow G$. These rules encode the process of solving a complex goal by simplifying it based on its structure. Thus, a conjunctive goal results in an attempt to solve each conjunct, a disjunctive goal becomes an attempt to solve one of the disjuncts and an existential goal is instantiated by a suitable closed term constructed using the current signature. Goals that have the form $E \supset G$ lead to the addition of the D formulas constituting E to the program. However, prior to doing this, the existential quantifiers in E must be instantiated by new constants that are also added to the signature. This process is initiated by generating a sequent of the form $\Sigma; \mathcal{P} - \langle \Delta \rangle \rightarrow\ G$ in which Δ represents a multiset of E formulas. The second category of rules that pertains to treating

$$\frac{\Sigma; \mathcal{P} \longrightarrow G}{\Sigma; \mathcal{P} - \langle\,\rangle \rightarrow\ G}\ \textit{finish} \qquad \frac{\Sigma; \mathcal{P}, D - \langle \Delta \rangle \rightarrow\ G}{\Sigma; \mathcal{P} - \langle D, \Delta \rangle \rightarrow\ G}\ \textit{augment}$$

$$\frac{\Sigma; \mathcal{P} - \langle E_1, E_2, \Delta \rangle \rightarrow\ G}{\Sigma; \mathcal{P} - \langle E_1 \wedge E_2, \Delta \rangle \rightarrow\ G}\ \wedge E \qquad \frac{\Sigma, c; \mathcal{P} - \langle E[c/x], \Delta \rangle \rightarrow\ G}{\Sigma; \mathcal{P} - \langle \exists x\, E, \Delta \rangle \rightarrow\ G}\ \exists E$$

$$\frac{}{\Sigma; \mathcal{P} \xrightarrow{A} A}\ \textit{initial} \qquad \frac{\Sigma; \mathcal{P} \longrightarrow G}{\Sigma; \mathcal{P} \xrightarrow{G \supset A} A}\ \textit{backchain} \qquad \frac{\Sigma; \mathcal{P} \xrightarrow{D[t/x]} A}{\Sigma; \mathcal{P} \xrightarrow{\forall x\, D} A}\ \textit{instan}$$

$$\frac{}{\Sigma; \mathcal{P} \longrightarrow true}\ \top G \qquad \frac{\Sigma; \mathcal{P} \xrightarrow{D} A}{\Sigma; \mathcal{P} \longrightarrow A}\ \textit{atomic}$$

$$\frac{\Sigma; \mathcal{P} \longrightarrow G_1 \qquad \Sigma; \mathcal{P} \longrightarrow G_2}{\Sigma; \mathcal{P} \longrightarrow G_1 \wedge G_2}\ \wedge G$$

$$\frac{\Sigma; \mathcal{P} \longrightarrow G_1}{\Sigma; \mathcal{P} \longrightarrow G_1 \vee G_2}\ \vee G_1 \qquad \frac{\Sigma; \mathcal{P} \longrightarrow G_2}{\Sigma; \mathcal{P} \longrightarrow G_1 \vee G_2}\ \vee G_2$$

$$\frac{\Sigma; \mathcal{P} - \langle E \rangle \rightarrow\ G}{\Sigma; \mathcal{P} \longrightarrow E \supset G}\ \supset G \qquad \frac{\Sigma; \mathcal{P} \longrightarrow G[t/x]}{\Sigma; \mathcal{P} \longrightarrow \exists x B}\ \exists G$$

Figure 1. Transition Rules Defining the Operational Semantics of the Logical Language

sequents of this form then implements the rest of the process of simplifying the E formula into a form in which it can be added to the program. The process of goal simplification terminates with the production of atomic goals. When this goal is different from $true$, a D formula from the program must be used to solve it. The *atomic* rule begins this process by picking a particular formula for this purpose; this rule has the proviso that D must be a member of $\mathcal{P}$. The rules for deriving sequents of the form $\Sigma; \mathcal{P} \xrightarrow{D} A$ encode the rest of the behaviour that is commonly referred to as backchaining.

The framework for computation described above possesses an interesting ability for giving scopes to names. To understand this, observe first that variables appearing in D formulas that are governed by existential quantifiers in an enclosing E formula correspond, in fact, to constants or names. The fact that we can explicitly quantify over such names then means that we can indicate a distinct scope for each of them. Thus, consider the E formula $\exists x\, D_1(x) \wedge \exists x\, D_2(x)$. While the same name x appears in both $D_1(x)$ and $D_2(x)$, the fact that these are governed by different quantifiers ensures that they are treated as distinct constants by the computation model. This situation should be contrasted with the one where the E formula in question is $\exists x\, (D_1(x) \wedge D_2(x))$ instead. Existential quantification also has an impact on the visibility of names in an outside context. Thus, if x represents a global constant, then it is allowed to appear in a term that is used to instantiate the quantifier over y in the course of trying to solve the goal $\exists y\, (D(x) \supset G(y))$. However, this constant *may not* be so used if the goal is $\exists y\, ((\exists x\, D_1(x)) \supset G(y))$ instead, *i.e.*, if the scope of the constant is explicitly narrowed via an existential quantifier.

In the rest of this paper we shall limit the programmer to writing only G formulas without implications and D formulas. Restricted to the first-order setting, these formulas are essentially a typed version of the queries and program clauses that underlie Prolog. We shall assume a syntax for their presentation that is motivated by this correspondence except that we shall use a curried notation for predicates. The E formulas and implication forms for G formulas will play what amounts to a meta-theoretic role, as a device for understanding the semantics of the modularity features that we discuss next. We note, however, that the full collection of formulas presented here can also be used for programming and are, in fact, a subset of the logical language defining λProlog. A substantive fact about this larger logic is that the procedure we have outlined for solving a G formula from a set of D formulas exactly captures the provability of the corresponding sequent in intuitionistic logic [14].

3. The Modules Language

We present in this section a simple modules language that builds on a logic programming core and we explain its static and dynamic semantics. In presenting the language, we focus only on the new components and avoid a description of those aspects of syntax that should be obvious from a familiarity with a language like Prolog. We do this assuming that there is no resulting loss in precision: a reader interested in all the details of program syntax spelled out relative to λProlog is referred to the documentation accompanying Version 2 of the *Teyjus* system for a BNF specification [6].

3.1 Modules and Signatures

In developing real programs, it is necessary, first of all, to identify a vocabulary of types and term constants. New type constructors are defined in our language through *kind* declarations that have the form

```
kind  tyc1, ..., tycn  type -> ... -> type.
```

where the arity of the constructors `tyc1`,. . .,`tycn` is one less than the number of occurrences of `type` in the declaration. Term constants are identified through *type* declarations of the form

```
type  c1,...,cn  <type expression>.
```

where the type expression is constructed using the available type constructors.

The notion of scope for kind and type declarations is important in constructing type and predicate definitions. Modules impart a structure to this space of names. Formally, a module begins with a declaration of the form

```
module <name>.
```

and continues with the kind, type and predicate definitions to be associated with the indicated name. We adopt a file oriented view of modules here: all the code defining a module named `foo` is to be found in a file with the name `foo.mod`. Now, the boundaries of a module determine a textual notion of scope in that the kind and type declarations that appear within them are interpreted as ranging over all the other declarations contained in the module. Consistent with this viewpoint, these boundaries also provide a delimiting region for analyses that a compiler might perform in the course of translating a source language program.

The following definition of a module called `store` illustrates module syntax:

```
module  store.

kind  store  type -> type.

type  emp  (store A).
type  stk  A -> (store A) -> (store A).
type  init  (store A) -> o.
type  add, remove
        A -> (store A) -> (store A) -> o.

init emp.
add X S (stk X S).
remove X (stk X S) S.
```

This module identifies a representation for stores with three associated operations, one for initializing a store and two others for adding an element to and removing an element from an existing store. Towards providing a definition that is parametric in the type of the elements stored, this module declares a unary type constructor for stores that is then used in the types of constants that implement store representations. The particular realization of a store embedded in this code is based on the idea of a stack. The last three lines in the module are D formulas that define the desired operations based on this interpretation. The tokens that begin with uppercase letters stand, as usual, for variables that are implicitly universally quantified at the head of the formula.

Not all the declarations in a module are typically intended to be externally visible. With the module `store`, for example, it is sensible to hide the actual representation of stores, requiring these to be manipulated opaquely through the predicates `init`, `add` and `remove`. In our language, the contents of a module that are to be visible to the outside must be explicitly identified through a signature that shares its name with the module. Formally, a signature begins with a declaration of the form

```
sig <name>.
```

and continues with the kind and type declarations to be associated with the specified name. Thus, the following declarations constitute a signature that imposes the kind of view desired on the module `store`:

```
sig  store.

kind  store  type -> type.

type  init  (store A) -> o.
type  add, remove
        A -> (store A) -> (store A) -> o.
```

There is an obvious consistency requirement with signatures: they must identify all the type constructors that are needed to sensibly interpret the types that appear in them. A syntactically well-formed signature actually functions as an interface definition for a module. From the perspective of an external use, a compiler must treat this signature as a complete description of all the type and kind declarations contained in the module that are available for global use in any context into which the module has been imported. From an internal perspective, the compiler must correspondingly check that the declarations in the signature agree with what is actually present in the module. We describe formally the process of checking these various requirements in the next subsection. In support of this kind of type checking role, we shall adopt a file oriented view of signatures similar to that for modules: the definition of the signature named `foo` is to be found in a file called `foo.sig`.

An important aspect of a module system is the mechanisms it provides for realizing interactions between units of code. In our system, this ability is obtained from a single operation referred to as *module accumulation*. The combination of modules through this operation is achieved by placing a declaration of the form

```
accumulate  M1, ..., Mk.
```

in a new module being constructed, assuming `M1`, ..., `Mk` are module names. The intended lexical effect of this declaration is to provide access to the signatures of the modules `M1`, ..., `Mk` within the new module. From a computational perspective, the effect of this declaration is to make available, in a logically controlled way, the clauses contained in the accumulated modules at the point of occurrence. This dynamic semantics will be made precise shortly via a translation into the underlying logical language.

Just as with modules, it is also possible to accumulate signatures. This effect is realized by using a declaration of the form

```
accum_sig  S1, ..., Sp.
```

in a module or signature being constructed; we assume here that `S1`, ..., `Sp` are signature names. This declaration is intended to have only a lexical effect that is similar to the one accompanying module accumulation.

3.2 Checking Signatures and Modules

A key part of formalizing the modules language is making precise the intended relationship between signatures and modules. There are two parts to this relationship: one part concerns the checking of wellformedness and the other part relates to the use of the module in computation. We treat the first aspect here, leaving a discussion of the logical and search related aspects to the next subsection.

Before the matching of signatures and modules can be discussed, it is necessary to understand how the full content of a given signature is to be extracted. This is done by a process called *signature elaboration* that also simultaneously determines if a signature is well-formed. The process is easily defined when the signature does not accumulate any other signatures: it simply collects all the kind and type declarations appearing in the signature. As for wellformedness, one set of conditions is obvious: all the type constructors used in the signature should either be defined in it or should be drawn from the pervasive set and each of these symbols should be used in a way that respects its arity. The second set of conditions stems from the fact that kind and type associations must be functional in nature. For kind declarations, this amounts to requiring that all such associations with any given token in the signature be identical. For type declarations the requirement takes into account the presence of type variables: all the types associated with a token must be identical up to (type) variable renaming. If this requirement is fulfilled, any one of the alphabetic variants is treated as *the* type associated with the token by the extracted signature.[2]

Signature elaboration for a signature that accumulates other signatures requires the notion of *signature merging*. A collection of elaborated signatures are *mergeable* if the following properties hold:

[2] It may seem unnecessary to allow for more than one kind or type declaration for the same symbol. However, this becomes more natural when we consider that such declarations might come from different signatures that are accumulated into the same context, an aspect that we treat next.

- If a symbol has a kind declaration in more than one signature in the collection, then all the declarations pertaining to it are identical, and
- If a symbol has a type declaration in more than one signature in the collection, then all the types associated with it through such declarations are identical up to a renaming of type variables.

Now, suppose that S_0 is a signature that accumulates the signatures $S_1, \ldots, S_n$. A prerequisite for the wellformedness of S_0 is that there be no accumulation cycles going through it, *i.e.*, that there is no sequence of accumulations starting at any one of $S_1, \ldots, S_n$ that includes S_0. If this condition holds, then the next requirement is that each of $S_1, \ldots, S_n$ should be well-formed. Let this property also hold and let $S'_1, \ldots, S'_n$ be the elaborations of these signatures. Then the third requirement for S to be well-formed is that $S'_1, \ldots, S'_n$ be mergeable. Suppose this also to be true and let S' be the signature obtained by replacing the accumulation declarations in S by the elaborations $S'_1, \ldots, S'_n$. Then, S' is the elaboration of S and the latter signature is well-formed only if the former (accumulation-free) one is.

The second step in the syntactic checking of a module consists of verifying that it is well-formed and simultaneously identifying an implicit signature for it. This step, once again, has an easy explanation in the case of a module that does not accumulate any other modules. The implicit signature here is determined by signature elaboration applied to the result of dropping all the program clauses from the module. The module is then well-formed if this signature is well-formed, if every constant used in the program clauses is defined in the signature or in the collection of pervasive constants, if each such constant is used at an instance of the type associated with it and, finally, each program clause is well-typed.

To treat the general case, suppose that a module M accumulates, in this order, the modules $M_1, \ldots, M_n$ that have specified signatures $\Sigma_1, \ldots, \Sigma_n$. There are then three requirements for M to be well-formed:

- M must not be part of a module accumulation cycle, *i.e.*, no sequence of module accumulations starting at one of $M_1, \ldots, M_n$ should include M.
- Each of the modules $M_1, \ldots, M_n$ must be well-formed and must match its specified signature.
- The module M' that is obtained from M by replacing the accumulation of modules $M_1, \ldots, M_n$ with an accumulation instead of the signatures $\Sigma_1, \ldots, \Sigma_n$ must be wellformed by virtue of the criteria already described for modules that do not accumulate other modules.

Suppose that all these conditions are met. The implicit signature for M is then identical to that associated with the module M' described above.

Let M be a well-formed module with the implicit signature S. Then we say that M matches a signature S' just in case S and the (signature) elaboration of S' are mergeable.

3.3 The Logical Interpretation of Modules

The *logical* or *dynamic* semantics of a module, as opposed to its *lexical* or *static* semantics that was treated in the previous subsection, is explained by a translation into an E formula and a subsequent use of the computation model for the underlying logical language.

The translation for a module that does not accumulate any other modules is actually quite simple. We first determine all the local constants for the module: these are the constants that appear in implicit signature of the module but not in the explicit signature associated with it. We then construct the desired E formula by conjoining all the D formulas (program clauses) contained in the module and existentially quantifying all the local constants over this conjunction. As an illustration of this idea, the logical essence of the module `store` considered earlier in this section is reduced to the formula

$$\exists Emp \exists Stk((init\ Emp) \wedge \forall X \forall S\,(add\ X\ S\ (Stk\ X\ S)) \wedge \forall X \forall S\,(remove\ X\ (Stk\ X\ S)\ S)).$$

under this approach.

The translation in the situation where the module accumulates other modules is only slightly more complicated. We proceed as before to identify the local constants for the module; as a pragmatic issue, notice that calculating this set requires the implicit signature of the module to be constructed which, in turn, requires that we look also at the (explicit) signatures of the accumulated modules. We then extract an E formula corresponding to each of the accumulated modules. This step involves a recursion that is well-defined because of the absence of accumulation cycles. Next, we construct an E formula by conjoining the E formulas corresponding to the accumulated modules with the program clauses contained in the module. The formula to be associated with the given module is now obtained by existentially quantifying the local constants over this E formula.

The formula that is constructed for a module that accumulates other modules may contain within it (essential) existential quantifiers that scope only over subparts of the formula. It is possible to move these quantifiers so that they have global scope. In carrying out such a transformation we may utilize the fact that an E formula of the form $(\exists x\, D_1(x)) \wedge D_2)$ is equivalent, in a provability sense and in the context of the calculus described in Section 2, to the formula $\exists x\, (D_1(x) \wedge D_2)$ provided that the variable x does not occur free in D_2. Notice that the kind of transformation that we are describing here may involve the renaming of local constants; for example, the formula $(\exists x D_1(x)) \wedge (\exists x D_2(x))$ would translate under it to something of the form $\exists y \exists z (D_1(y) \wedge D_2(z))$, with x in D_1 and D_2 being renamed to the distinct and fresh variables y and z. At a programming level, this kind of transformation is tantamount to treating accumulation as the inlining of code in the accumulated module, taking care, however, to preserve the locality of names within it. The implementation scheme that we describe in Section 5 will exploit the possibility of carrying out this kind of static simplification of E formulas (or, alternatively, of the code available from a module) extensively in getting improved run-time behaviour.

3.4 Interpreting Queries Against Modules

The attempt to solve a goal at the top level is always made in the context of a chosen module. From a lexical perspective, the significance of such a relativization is that all the kind and type declarations in the signature of the module become available in analyzing the syntax of the goal. On the other hand, the relativization from the perspective of computation is realized (at least conceptually) by constructing an implicational goal. In particular, suppose that we are interested in solving the goal $G(y_1, \ldots y_n)$ given the code in the module M; the sequence $y_1, \ldots, y_n$ depicts the free variables of the goal here. Assuming that M translates into the formula E, this desire is then understood precisely as that of wanting to solve the goal $\exists y_1 \ldots \exists y_n\, (E \supset G(y_1, \ldots, y_n))$.

The interpretation of modules and goals that we have just described actually embodies a fairly strong form of hiding. In the schematic example that we have considered, the lexical interpretation obviously prevents $G(y_1, \ldots, y_n)$ from referring directly to any constant local to M. A little less obvious is the fact that these local constants cannot also percolate out of M in the form of computation results. In particular, the results computed for this query would be instantiations for the variables $y_1, \ldots, y_n$ that lead to a

successful solution of the goal $\exists y_1 \ldots \exists y_n \, (E \supset G(y_1, \ldots, y_n))$. The operational semantics that we have described in the previous section now clearly precludes the appearance of the local constants of M in instantiations for y.

3.5 Some Remaining Odds and Ends

The discussion of the restrictions on bindings for variables appearing in goals reveals an interesting dynamic aspect to the ability to limit scopes of constants that existential quantifiers in E formulas provide. To understand this, consider again the goal $\exists y_1 \ldots \exists y_n \, (E \supset G(y_1, \ldots, y_n))$. A practical treatment of this goal would need to defer the particular choice of instantiations for the variables $y_1, \ldots, y_n$ till such time that there is information available for making them insightfully. Typically, such delaying is realized by using placeholders or logic variables $Y_1, \ldots, Y_n$ for the instantiation initially; these logic variables have the characteristic that they can be substituted for via a unification process later in the computation. Subsequent to this instantiation, suitable constants would have to be determined for the existentially quantified variables appearing in E prior to adding the clauses in E to the program context. We note at this point that, in order to be correct with respect to the logical semantics, the variables $Y_1, \ldots, Y_n$ *must not* be allowed to be instantiated with terms containing these newly introduced constants.

The way we have described the computation above might make it seem prohibitively expensive and, hence, unacceptable as the basis for a practical notion of information hiding related to modules: local constants that appear in modules should after all be treated largely as constants and should not lead to costly run-time manipulations. It turns out, however that an implementation can be provided that essentially respects these pragmatic considerations. We have described (and proved correct) elsewhere a benign extension to the usual unification computation that realizes explicitly provided scopes for names by assigning numeric labels to logic variables and constants and then using these these to ensure adherence to the occurrence constraints [18]. We do not discuss this matter in any more detail here, but we assume the availability of such an implementation in ensuring the overall practicality of our modules language.

An important requirement that we have imposed on the accumulation of modules and signatures is that such accumulation should eventually not be cyclic. Note that, in a separate compilation model, cyclicity in module accumulation chains is a property that can only be checked at linking time. A common worry is that the prohibition of accumulation cycles might be tantamount to preventing a mutual dependence between modules. Fortunately, this worry turns out not to be true. For example, suppose that the two modules M_1 and M_2 each contain code that depends on predicate definitions that appear in the other module. This kind of interaction can be supported without running afoul of the requirement that there be no accumulation cycles by accumulating any one of these modules into the other or, perhaps better still, by accumulating both modules into a common context.

We have at this point described the logical structure of the modules language in its entirety: A module corresponds syntactically to a possibly large E formula obtained by combining its clauses, its signature and the formulas corresponding to its accumulated modules in the manner just described. The dynamic semantics of the module is then explained completely and precisely through this formula and the operational semantics for the underlying logical language. The next section adds further syntactic sugar and some compiler-oriented annotations to this language without modifying the logical core.

4. The Practical Use of Modules

Although the modules language described in the previous section is simple, it is quite versatile at a programming level. We attempt to bring this facet out in this section by considering some paradigms for its use.

4.1 Hiding and Abstract Data Types

A module in our language allows code that supports a desired functionality to be collected into a named unit and an associated signature provides a window into this code. Now, the capabilities implemented by a module may be needed in the context of another module. Module accumulation supports the realization of such an interaction.

As an illustration of the above idea, suppose that we wish to implement a heuristic-based graph search procedure. This procedure would initialize a collection of states and then expand this set based on the rules for generating new states and an underlying strategy for selecting the next state for expansion. To realize what is required of it, this code may need the implementation of a store. This can be obtained by accumulating the module `store` presented earlier. Figure 2 displays part of the definition of a graph search module to illustrate this idea. The accumulation of `store` gives this module a type for stores that is used in the type declarations of `init_open` and `expand_graph`. This accumulation also allows the procedures `init`, `add` and `remove` to be used in the code in `graph_search` module. An aspect worthy of note is that while the (universally quantified) variables in the program clauses in this module can be instantiated with store representations, these representations are still abstract: the implicit existential quantification over local constants imposes visibility restrictions that ensure that their inner structure can only be accessed by recourse to operations in the module `store`. Note also that by excluding declarations for `init`, `add` and `remove` from the signature for `graph_search`, the predicate definitions in `store` can be made entirely private to the graph search module. Thus, in contrast to the `import` construct of [12], we are able to achieve code scoping by simply structuring name spaces in a *completely statically determined* collection of code.

4.2 Module Parameterization and Sharing of Code

The example in the previous subsection shows how a *private* copy of code can be acquired by a module. While this may be the desired behaviour in some situations, the accumulated module may in many other cases represent a common capability that is to be shared between different modules in a large system. As a specific example of this, consider the need for the modules implementing different functionalities in a system like a compiler to interact through a shared, efficient representation of a collection of symbols extracted from a user program. In such a setting, none of the modules in question need to know the innards of the representation of the "symbol table," but they do need to know that the representation is one that is shared between them and that it can be manipulated through a common set of procedures for inserting, deleting and looking up items in the table. Moreover, it is desirable to use *one* copy of the code implementing the symbol table in the entire system rather than replicating the code at each place it is needed.

A solution to this problem is to think of modules that use such library capabilities as being *parameterized* by them. Our modules language supports this kind of parameterization in a natural way. For example, suppose that we wish to think of the module `graph_search` as one that depends on an externally provided implementation of stores rather than one that it accumulates. This dependency can be manifest by including appropriate declarations in its signature. In particular, this signature would identify the type constructor `store` and the constants `init`, `add` and `remove`. We

```
module graph_search.
accumulate store.
kind state,action type.
type graph_search list action -> o.
type init_open store state -> o.
type expand_graph store state -> list state -> list action -> o.
...
graph_search Soln :- init_open Open, expand_graph Open nil Soln.
init_open Open :- start_state State, init Op, add State Op Open.
expand_graph Open Closed Soln :-
        remove State Open Rest, final_state State, soln State Soln.
expand_graph Open Closed Soln :-
        remove State Open ROpen, expand_node State NStates,
        add_states NStates ROpen (State::Closed) NOpen,
        expand_graph NOpen (State::Closed) Soln.
...
```

Figure 2. A Module Implementing Graph Search

would, of course, have to provide the module with the functionality it needs eventually. This can be done by accumulating the module `store` at the relevant place. As a specific illustration, suppose we wish to test our implementation of graph search. A harness suitable for this purpose can be expressed via the following module:

```
module test_graph_search.
accumulate graph_search, store.
```

By endowing this module with a signature that makes the needed types, data representations and predicates defined within the module `graph_search` externally visible, we can pose queries against it that exercise the capabilities that are to be tested. This discussion also shows how different modules in a composite system can share "library" capabilities: they can all be parameterized in the same way that the `graph_search` module was in this example and the the parameterization can eventually be discharged by accumulating the library module into a common context.

4.3 Renaming of Imported Constants

The `accumulate` and `accum_sig` constructs as we have described them up to this point import the global names from the relevant module or signature without change into the context of accumulation. This can sometimes be problematic: the same name may have been used for different purposes in independent components of code and an inability to distinguish between them would preclude benefitting from the functionalities they represent in a common context. To circumvent this problem, we allow for the accumulation declaration to be optionally parameterized by a function that renames some of the incoming global constants. In particular, a declaration of the form

```
accumulate  M1 {
          kind tyc1 --> tyc2
          c1 --> c2
          }
```

signals that the global type constructor `tyc1` and the global constant `c1` in the module `M1` are first to be renamed to `tyc2` and `c2` respectively and the resulting module is then to be accumulated into the relevant context; this renaming process will use an identity mapping on the names of the global constants and type constructors that are not mentioned explicitly in the `accumulate` declaration and also requires that the eventual function used has a one-to-one character on the type constructor and constant spaces, respectively. As a specific example, the accumulate declaration

```
accumulate  store {
          kind store --> mystore
          add --> insert
          remove --> delete
          }
```

results in the accumulation of the module `store` with the type constructor it provides renamed to `mystore` and the predicates for adding and removing from a store renamed to `insert` and `delete`, respectively. A similar syntax is used to realize renaming with respect to accumulated signatures.

4.4 Annotations for Fixing Predicate Definitions

Predicate definitions in the logic programming context can be expanded by adding further clauses. The definitions emanating from an accumulated module have the potential of being extended in this way in the accumulating module. It is sometimes desirable to curtail this possibility. Referring to the module `store`, for instance, we may want to freeze the definition of the operations `init`, `add` and `remove` that it provides. This possibility is supported by permitting an "export" annotation in signatures. Specifically, by replacing

```
type  add  A -> (store A) -> (store A) -> o.
```

with the annotated declaration

```
exportdef  add
           A -> (store A) -> (store A) -> o.
```

in the signature of the module `store`, we may signal that the definition of `add` may not be altered by the accumulating context. Paying attention to the model for pairing functionality that we have just sketched, our modules language also provides a complementary "useonly" annotation. Thus, by using the declaration

```
useonly  add
         A -> (store A) -> (store A) -> o.
```

instead of the type declaration for `add` in the `graph_search` module, we indicate that the definition of this predicate can be used but cannot be altered in this module. We note that at a logical level both the `exportdef` and `useonly` declarations are *identical* to type declarations. They differ from type declarations only at a pragmatic level by imposing special wellformedness restrictions—that must be checked and can be made use of by a compiler—on module composition. The restrictions can, however, be quite useful in practice: they impart a completeness property to definitions that can

```
sig comblibrary.
type  call  o -> o.

sig test.
type  test  list int -> o.
```

```
module comblibrary.
type  call  o -> o.
type  p  list int -> o.
p (1 :: nil).
call Q :- Q.

module test.
accumulate comblibrary.
type  test  list int -> o.
type  p  list int -> o.
p (2 :: nil).
test X :- call (p X).
```

Figure 3. Interpreting predicate names in higher-order programs.

help in reasoning about program properties and also in generating better object code especially in a separate compilation setting.

The kind of fixed predicate definitions discussed above might be expected to be provided typically by library modules. A simple way to benefit from their functionality would be to accumulate their (external) signatures but with a twist: we would want all the `exportdef` declarations in such a signature to be transformed into the counterpart `useonly` declarations at the point of accumulation. We provide the declaration

```
use_sig S1, ..., Sp.
```

in which `S1`, ..., `Sp` are expected to be signature names for this purpose. Like the `accum_sig` declaration, this declaration also causes the mentioned signatures to be included in place but only after each `exportdef` declaration is converted to a `useonly` one.

4.5 Code Extension and Modular Composition

While predicate definitions can made to be self-contained within specific modules either because of annotations of the kind described in the previous subsection or because they pertain to local constants, it may also sometimes be desirable to distribute them across two or more interacting modules. This kind of distribution is in keeping with the logic programming style that builds a predicate definition by combining the effect of several clauses. Pragmatically, the spreading of a definition seems to have some uses as witnessed by multifile declarations in Prolog. As another practically pertinent example, consider the task of implementing proof relations in different logics. A common part to all these logics may be the treatment of propositional rules. This treatment may be isolated in a particular module named, say, `prop_logic`. A realization of first-order logic may then accumulate `prop_logic` and extend the predicates defined therein. This kind of construction of predicate definitions by incremental composition is naturally supported by our modules language. Such a composition also raises special problems for separate compilation. A treatment of these problems is considered in the next section.

4.6 Higher-Order Programming and Predicate Visibility

Our language allows for the writing of predicates that are parameterized by other predicates. Such predicate definitions have general applicability and might therefore be usefully collected into a library module. The invocation of such predicates will, of course, supply them with specific predicate names as arguments. A question that has evoked interest relative to the discussion of modularity and the treatment of the `call/1` predicate in Prolog is how these names should be interpreted.

Figure 3 presents module definitions designed to bring out the relevant issue. An attempt to solve the goal `test X` relative to the module `test` will lead to the invocation of the goal `call (p X)` that, in turn, will cause the goal `p X` to be called. The question then is what the definition of the name `p` should be when this goal is called. Two competing possibilities have been suggested [7]: its denotation may be determined by whatever is visible in the context where the predicate `call` is defined or by the environment in which the name is explicitly used. In this instance, the top-level query will succeed either way, but with different results depending on which answer one takes: `X` will be bound to `1 :: nil` in the first case and to `2 :: nil` in the second.

The second interpretation is the resolution that is now commonly accepted. This interpretation has the advantage that the denotations of names are determined statically, an important requirement for any good notion of modularity. It is interesting to note that this interpretation is also a natural consequence of the semantics that we have presented for our modules language. The possibility of two different interpretations arises from a separation between the calling and the called context. This separation plays no role in our semantics. The module `test` is, in fact, treated as *one* collection of declarations in which existential quantifiers with limited scope control the visibility and, hence, the identity of names. The relevant occurrence of the name `p` therefore has an unambiguous interpretation and the only answer substitution to the query shown above is the binding `2 :: nil` for `X`.

5. A Separate Compilation Scheme

A naive implementation of the language we have described can be obtained by a compile-time inlining of accumulated modules. We show in this section that this approach can be refined into one where each module is individually compiled and the inlining is carried out by a later linking phase. Towards exposing the issues that have to be addressed, we first outline the naive approach below. We then use this context to develop the improved separate compilation scheme. In this discussion we ignore the effects of the `exportdef` and `useonly` annotations and also the capability of explicitly renaming constants and type constructors that we discussed in Section 4 in association with the `accumulate` and `accum_sig` declarations. An actual compiler must build in a treatment of these features but it is easy to see how to do this once we have understood how the basic form of signature and module accumulation can be effectively handled.

5.1 A Naive Implementation of Module Accumulation

Figure 4 presents a typical example of module interactions that an implementation must be capable of handling. In this example, rather than explicitly displaying signatures, we have marked constants as either global or local directly in the code of modules. For simplicity, we have also elided type declarations.

```
module m1.                  module m2.
global r,w.                 global r.
[clauses in m1]             [clauses in m2]

module m3.                  module m4.                  module m5.
accumulate m1.              accumulate m2.              accumulate m3,m4.
local r.                    local r, w.                 local q.
global w.                                               global w.
[clauses in m3]             [clauses in m4]             [clauses in m5]
```

Figure 4. An example of nested accumulation

A naive implementation of our language can be obtained through a process of inlining accumulated modules.[3] However, this process has to be careful about distinguishing constants that come from different accumulated modules and must map them to appropriately scoped ones in the larger module it constructs. A schematic depiction of what such an inlining compiler must accomplish appears in Figure 5. We have used here the names `r[1]` and `r[2]` to distinguish the two constants with name `r` that come from the modules `m3` and `m4` and we have similarly employed the names `w[1]` and `w[2]` to differentiate between the global constant in module `m5` and the local constant in module `m4` that share the name `w`. Further, we have exploded the renaming process into a cascade of steps following the accumulation chain. For example, the constant `r` appearing in the clauses of module `m1` is to be renamed to the first local in the enclosing context (module `m3`) which is itself renamed to the second local in the outermost context. In reality, an inlining compiler can collapse this nesting by actually carrying out the sequence of renamings, yielding a module with one set of global and local constants and a collection of clauses from all the modules with the constant in them appropriately identified. It can then proceed to compile the clauses with complete knowledge of all the relevant predicate definitions.

5.2 A Separate Compilation Based Treatment

The previous model indicates the structure of the code that needs to be produced prior to execution. We are interested, however, in generating this code from compiled versions of each of the modules `m5`, `m4`, `m3`, `m2` and `m1` that have been generated without knowledge of where they are going to be used and information at most of the signatures of modules that they accumulate. In this situation

1. the compiler will not have specific knowledge when compiling a potentially accumulated module of what the global and local constants are going to be mapped to in the enclosing context,
2. for predicates whose names are global and whose definitions are extendible, the compiler will have to produce code assuming that the clauses in the module form an incomplete set and must be fitted into a larger context, and
3. the code that the compiler produces may have calls to predicates whose entry points cannot be determined at compilation time but must wait till the relevant assembly of modules is put together.

The tasks of the inlining compiler will, under these circumstances, have to be divided between a compiler that produces code for each module separately but includes in such code suitable annotations that allow it to be fitted into a larger context and a linker that uses the "glue" information with each module to build a complete bytecode image of the system. We sketch the structure of these components below that in combination achieve the desired result. For concreteness in presentation, we will assume as a target low-level code that can be run on an architecture closely related to the Warren Abstract Machine (WAM) [25]. We assume familiarity with this machine structure below.

[3] We have chosen to explain the semantics of our modules constructs here by translation into the core logical language and a subsequent use of the operational semantics of that language. An alternative approach, that might not pay as close attention to the logical underpinnings, would be to develop this inlining or *module elaboration* presentation more formally.

5.2.1 The Outcome of Compilation

Given a module, the compiler that we envisage will produce a file for the linker that has the following items of information:

1. A listing of the global constants that includes their names and other information such as their types that will be needed during execution.
2. A listing of the local constants similar to that for the global ones but, this time, the names are not needed.
3. A list of names for each accumulated module paired with a mapping from its global names (obtained from its signature) to indices into either the local or global constant list for this module.
4. A listing of the (indices of) externally redefinable predicates.
5. WAM-like code obtained by compiling the clauses presented in the module. Constant indices in this code will be indices into the lists in the header, to be patched up eventually by the linker. Calls to externally redefinable predicates also use indices into the listing of these predicates and will have to be filled in after the entry point for these has been finally determined.
6. A map from predicate names (represented by their indices) to their entry points in the code space.

In the WAM setting, the code that is produced for individual clauses defining a predicate is surrounded by instructions for sequencing through choices and also indexing into them. The typical structure for this is illustrated below:

```
        try_me_else L1
        switch_on_term V1,C1,Lst1,S1
C1:     switch_on_constant CHT1
S1:     switch_on_structure SHT1
V1:     try_me_else L12
        [code for one clause]
L12:    retry_me_else L13
        ...
LLn:    trust_me
        [code for last clause]
L1:     retry_me_else L2
        [code for another block]
```

```
module m5.
global w[1].
local q, r[1], w[2], r[2].
{accumulate m3 [r -> second local: r[1], w -> first global: w[1]]
 global w.
 local r.
 [clauses from m3] with constant references suitably resolved
 {accumulate m1 [r -> first local: r, w -> first global: w]
  [clauses from m1] with constant references suitably resolved}}
{accumulate m4 [r -> fourth local: r[2], w -> third local: w[2]]
 local r, w.
 [clauses from m4] with constant references suitably resolved
 {accumulate m2 [r -> first local: r]
  [clauses from m2] with constant references suitably resolved}}
[clauses from m5] with constant references suitably resolved
```

Figure 5. The definition of module m5 after inlining accumulates

```
      ...
Ln:   trust_me
      [code for last block]
```

At the outermost level, this code captures a possible sequencing through different chunks of clauses. Each chunk corresponds to a subsequence over which indexing may be useful. One component of this "indexed" subsequence pertains to the variable case that must support a simple sequencing through the entire collection. The other possibility, corresponding to lists or hashing on constant name or structure names, is that only some of the clauses in the sublist are relevant. In this case, auxiliary sequencing code using the instructions `try`, `retry` and `trust` would be generated. These possibilities are not specifically illustrated above but are nonetheless relevant to the discussions that follow.

5.2.2 The Linking Process

Linking begins by creating a frame for a flattened image of the top-level module to be filled out by functions that map global and local constants to runtime indices, recursively load the accumulated modules and, finally, add the code to the frame.

The mapping of constants to runtime indices follows the cascading structure of the inlining compiler, except that this is done at linking time. In some detail, each local constant in the chain of accumulates translates into a unique index. The global constants of the top-level module also are assigned unique indices. Finally, the assignments for the local and global constants of the parent module and the associated renaming functions determine the indices of global constants of accumulated modules.

An issue that is important in preparing the code for addition to the frame is that of combining predicate definitions: different modules may provide pieces of the definition of a predicate and these need to be assembled together. To begin with, there must be a fixed order governing the assembly—in λProlog, for instance, clauses from the accumulated modules appear first in the order of accumulation followed by those in the parent module—and this is adhered to by the linker. Now, the code that is generated for a predicate in each module has the structure of a list and a natural first step towards integration is appending separate lists together. Ignoring for the moment the existence of indexing in the WAM code, it is easy to see how this might be done. For instance, suppose that compilation of one module has produced the following code sequence for a particular predicate:

```
L1: try_me_else L2
    [code for a block]
L2: retry_me_else L3
    [code for a block]
L3: trust_me
    [code for a block]
```

Further, suppose that the compilation of another module has resulted in the following sequence for the same predicate:

```
L4: try_me_else L5
    [code for a block]
L5: retry_me_else L6
    [code for a block]
L6: trust_me
    [code for a block]
```

This combination of these two blocks of code can be realized by changing the `trust_me` instruction that precedes the last block of the first sequence into a `retry_me_else` instruction pointing to the start of the next definition and by changing the first instruction of that collection into a `retry_me_else` to yield the following:

```
L1: try_me_else L2
    [code for a block]
L2: retry_me_else L3
    [code for a block]
L3: retry_me_else L4
    [code for a block]
L4  retry_me_else L5
    [code for a block]
L5: retry_me_else L6
    [code for a block]
L6: trust_me
    [code for a block]
```

The indexing optimization in the WAM complicates matters a little because some elements of the top-level sequence may be indexed blocks. If we were to simply append the top-level sequences as suggested, the new sequence may have two adjacent blocks of this kind. This is undesirable: it may mean, for instance, that we end up keeping a choice point on the stack when one is not really needed. Fortunately, it is possible to avoid this. We can determine if this will occur by examining the last element of the first sequence

and the first element of the second sequence. If they are both indexed blocks, then we proceed to merge them.

One problem to be addressed in generating a single indexed block is, once again, that of merging two segments of code that represent sequencing through clauses. For the "main" sequences corresponding to the variable case, this can be effected as for top-level sequences. To treat the case when these may be sequences realized through `try`, `retry` and `trust` instructions, we augment the instruction set with two new instructions called `try_else` and `retry_else`. These instructions behave like `try` and `retry` except that that they take an additional argument that provides the address of the code to try upon on backtracking. Suppose now that the two blocks of sequencing code that we need to merge are the following:

```
S1:   try L1          S2:   try L4
      retry L2              retry L5
      trust L3              trust L6
```

This merging can be realized by changing the code to the following:

```
S1:   try L1                S2:   retry L4
      retry L2                      retry L5
      retry_else L3,S2              trust L6
```

The `try_else` instruction is needed in implementing this idea in the case where the first block corresponds to a unique clause choice.

The other problem that needs to be dealt with in combining indexing blocks is that of merging hash tables corresponding to constant and structure names. This is easy to do. The compiler can actually emit the separate tables simply as lists of pairs of constant names and corresponding entry points to code. The linker can determine from this which lists have to be merged and then emit the merged lists, which are used by the emulator to generate the hash tables.

The last aspect that the linker must resolve is the (relative) code location for predicates that could not be finalized at compile time. After the combining of all the clause code has been completed, a map is available from each predicate name in our universal namespace to the location of its definition. These addresses can now be patched in at the appropriate places.

6. Related Work and Conclusion

This paper has described a logic-based interpretation of modularity in logic programming. A natural question that arises in assessing its contributions is the relationship of the treatment it proposes to that in functional and other related styles of programming. There are obvious similarities at the pragmatic level to the ideas of signatures and structures in Standard ML [11, 15] and existential quantification in programs looks enticingly similar to existential types that underlie hiding in functional programming [16]. There is a large body of work pertaining to type checking, separate compilation and related issues in this setting (*e.g.* see [8, 10, 22]) that may have bearing on similar issues in the logic programming setting and that should be examined more closely to get a clearer understanding of the connections. We have not done this here primarily because our treatment of types has been simple and our main concern has been with the impact the modularity features have on the proof search aspect that is unique to logic programming. Much of the attention in this paper has, in fact, been on spelling out a coherent logical viewpoint for modularity in logic programming and then describing how a satisfactory computational treatment can be provided to realize the effect this has on the attendant proof search.

Another relevant comparison is with work that endows actual Prolog systems with modularity capabilities. A major concern within such efforts (*e.g.*, [3, 23, 24]) has been the interpretation of metalogical predicates such as *call* and the treatment of declarations relating to syntax. The focus on this view of modularity has significant practical relevance—*e.g.*, see [3] for its importance to language extensibility. However, this concern is orthogonal to our primary one here relating to name and code scoping. With regard to the handling of names, these other approaches have been somewhat *ad hoc* at a logical level, permitting the hiding of predicate names but not those of functors and constants, a critical aspect of data abstraction. The treatment in the *Mercury* language is closer to ours pragmatically but differs in that it requires either all or none of the constructors of a type to be hidden [1]. Moreover, our use of existential quantifiers can lead to richer computations that require a more sophisticated unification procedure. Unfortunately space does not permit a fuller discussion of this issue.

We should also contrast our work with those in logic programming that focus on a logic-oriented approach to realizing modularity. The proposal of Sanella and Wallen [21] that brings ideas from ML into the logic programming setting is one example of this. The notions of signatures and structures in this proposal once again correspond closely to our ideas of signatures and modules. One *difference* is that [21] does not allow for a predicate definition to be built up across different structures/modules; such a capability has potential usefulness in a logic programming setting as argued in Section 4. We also note that, in our setting, the hiding realized through signature specifications is explained in a logic-based way. Another proposal is that of Miller [13] that subsumes the constructs we have used here. Our contribution relative to this work is to demonstrate that an entirely static subpart of it suffices to realize scoping over clause definitions as well. Finally we mention the work of Harper and Pfenning [9] that adapts an ML-like approach to modularity to an LF based logic programming language but that, like [12], also allows for dynamic modifications of predicate definitions.

At an implementation level, we have had to deal with two different issues: the treatment of scope for existentially quantified variables and the combining of code for a given predicate that is obtained from compiling different modules. The second issue is pertinent also to multifile definitions in, for instance, the SICStus system. The solution adopted there is different at least on the surface: compilation is done directly to core and indexing is realized interpretively based on a data structure that is built up incrementally as each clause is compiled [4]. It is of interest, however, to see if aspects of that approach can be adapted to our separate compilation setting as well.

There are different aspects relevant to the work we have presented here such as the logical features underlying our proposal for modularity, the syntax chosen to support this notion, the treatment of name scopes in compilation and computation (or, more precisely, in unification) and the realization of separate compilation. Each of these aspects has received consideration individually in past work, raising the question of what precisely the contribution of this paper is. The main novelty here, in our estimation, is in the way we combine these different ideas to yield a logic motivated approach to modularity that is pragmatically useful and that has a simple, separate-compilation based implementation.

The modules language that we have described has been implemented within the *Teyjus* system. The inlining approach described in Section 5.1 was already present in the first version of this system [19]. Experience relative to this system with the approach to scoping that we have advocated has been positive: users have adapted easily to this method from the dynamic, import based approach that it also supported. We have recently completed and released a second implementation of this system [6]. The newer system includes an implementation of the modules constructs described in this paper that is based on the ideas presented in Section 5 and that consequently supports separate compilation.

Acknowledgments

This paper has benefitted from suggestions for improvement received from the reviewers of an earlier version. Dale Miller has also provided helpful comments. Support for this work has been provided by the National Science Foundation under Grant No. 0429572. Opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] Ralph Becket. Mercury tutorial, 2005. Available at the URL `http://www.cs.mu.oz.au/research/mercury/tutorial/book/book.pdf`.

[2] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. Lpez, and G. Puebla. *The Ciao Prolog System*, August 1997. Reference Manual, Technical Report CLIP 3/97, School of Computer Science, Technical University of Madrid.

[3] D. Cabeza and M. Hermenegildo. A new module system for Prolog. In *Computational Logic - CL 2000*, pages 131–148. Springer, 2000. LNAI Vol 1861.

[4] M. Carlsson. Private Communication, June 2007.

[5] International Organization for Standardization. Prolog. ISO/IEC 13211 — Part 2: Modules, 2000.

[6] A. Gacek, S. Holte, G. Nadathur, X. Qi, and Z. Snow. Teyjus version 2: An implementation of λProlog, April 2008. Available from `http://teyjus.cs.umn.edu`.

[7] Rémy Haemmerlé and François Fages. Modules for Prolog revisited. In S. Etalle and M. Truszczynski, editors, *ICLP: Logic Programming, 22nd International Conference*, volume 4079 of *LNCS*, pages 41–55. Springer, August 2006.

[8] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–137, New York, NY, USA, 1994. ACM.

[9] R. Harper and F. Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 8(1):5–31, 1998.

[10] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122, New York, NY, USA, 1994. ACM.

[11] D. MacQueen. Modules for Standard ML. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 198–207, New York, NY, USA, 1984. ACM.

[12] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[13] D. Miller. A proposal for modules in λProlog. In R. Dyckhoff, editor, *Proceedings of the 1993 Workshop on Extensions to Logic Programming*, pages 206–221. Springer-Verlag, 1994. Volume 798 of Lecture Notes in Computer Science.

[14] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[15] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[16] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3):470–502, 1988.

[17] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Sixth International Logic Programming Conference*, pages 284–299. MIT Press, June 1989.

[18] G. Nadathur. A proof procedure for the logic of hereditary Harrop formulas. *Journal of Automated Reasoning*, 11(1):115–145, August 1993.

[19] G. Nadathur and D.J. Mitchell. System description: Teyjus—a compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *Automated Deduction–CADE-16*, number 1632 in Lecture Notes in Artificial Intelligence, pages 287–291. Springer-Verlag, July 1999.

[20] G. Nadathur and G. Tong. Realizing modularity in λProlog. *Journal of Functional and Logic Programming*, 1999(9), April 1999.

[21] D.T. Sannella and L.A. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programming*, 12:147–178, January 1992.

[22] Zhong Shao. Transparent modules with fully syntatic signatures. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 220–232, New York, NY, USA, 1999. ACM.

[23] Swedish Institute of Computer Science. *SICStus Prolog v3 User's Manual*. The Intelligent Systems Laboratory, PO Box 1263, S-164 28 Kista, Sweden, 1991–2004.

[24] Swedish Institute of Computer Science. *Quintus Prolog v3 User's Manual*. The Intelligent Systems Laboratory, PO Box 1263, S-164 28 Kista, Sweden, 2003.

[25] D.H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, October 1983.