A Parameterised Module System for Constructing Typed Logic Programs

P.M. Hill*
Division of Artificial Intelligence, School of Computer Studies
University of Leeds, Leeds, LS2 9JT, UK
hill@scs.leeds.ac.uk

Abstract

The paper is concerned with the design of a module system for logic programming so as to satisfy many of the requirements of software engineering. The design is based on the language Godel which is a logic programming language which already has a simple type and module system. The module system described here extends the Godel module system so as to include parameterised modules. In particular, this extended system allows general purpose predicates that depend on facts and rules for specific applications to be defined in modules that are independent of their applications.

1 Introduction

Logic programming has been used extensively for representing and reasoning about knowledge bases. For large knowledge bases we require a means of segmenting the program so that small component parts of the knowledge base can be developed. These can then be used to build larger components, and so on, until the program is completed. These components are called *modules.*

Modules can be researched from a number of points of view including software engineering, object-oriented programming, and theory construction. We concentrate here on the software engineering use of modules and, in particular, the use of modules in program construction. There are a number of requirements for such a module system.

1. There must be a means of combining modules. This is normally achieved by allowing one module to *import* another.
2. Part of a module should be protected from unintended use by other modules. This is called *encapsulation.* Usually a module is divided into two parts. One part defines a language that can be used by an importing module. The other part extends this language with symbols only required locally.
3. It should be possible to develop a module independently of other modules that it does not import. Thus the import relation is normally restricted to defining a partial order on the modules in a program. The order of compilation of the modules must then respect this ordering.
4. A module should be usable in as many contexts as possible. A module providing an abstract data type such as a stack or a definition of an abstract relation such as transitivity needs to be *re-usable* and not tied to a specific application.

The reasons for having types in logic programming languages are well known. The structure of the knowledge domain can be represented directly by means of type declarations. These declarations also define the intended use of the symbols and therefore protect the program from syntactic errors caused by misuse of the symbols. The language on which we have based our ideas is the logic programming language Godel [Hill and Lloyd, 1992]. Godel has a parameterised type system that supports generic but not inclusion polymorphism. Moreover, Godel has a simple module system that supports importation, encapsulation, and separate compilation as well as allowing for modules defining abstract data types. The Godel module system does not support re-usable modules defining abstract relations such as transitivity. The parameterised module system described here extends the module system in Godel so as to provide better facilities for defining abstract relations in re-usable modules.

Other authors have investigated modules for logic programming. [Miller, 1986] extends Prolog to provide a theory of modules over Horn clauses. The modules are defined by nested implication with the "semantics based on intuitionistic logic. In this module system, the modules are dynamic in the sense that they are created and deleted at run time. Thus a modification

•Supported by SERC grant GR/H/79S62

of SLDNF-resolution is required for the procedural semantics.

The module system given in [O'Keefe, 1985] only deals with untyped programs where the predicates are considered to be local to the modules but the functions are assumed to be global. Thus abstract data types cannot be defined. A number of Prolog implementations supporting module systems similar to that described by O'Keefe have been marketed.

[Sannella and Wallen, 1992] describe a Prolog module system based on the theory of modularity underlying the Standard ML module system. This module system (extended to include types) provides all the facilities provided by Godel. It also allows for a form of parameterised modules although the parameterisation is with respect to the module names instead of the symbols. However, the main difference is that a predicate must be defined in a single module, whereas, in our system, predicates that are parameterised can be defined in more than one module. The system is less flexible than the one described here but safer in that the predicates are better protected from unintended use.

[Goguen and Meseguer, 1984] present EqLog which combines functional and logic programming. The language provides a parameterised module system which appears to be similar in function to the module system described in [Sannella and Wallen, 1992]. However, it is described in the framework of EqLog rather than Prolog so that it is not immediately applicable to logic programming languages.

[Antoniou and Sperschneider, 1992] divide a module into four parts; import, export, body, and parameter. The import and export parts use Horn Clauses to specify the imported and exported predicates. These are only used for the combination of the modules. Exported predicates are also defined in the body part of the module. It is these definitions that are used in the execution of the program. The parameter part specifies generic predicates using full first order logic. Importing modules supply the implementation for each of these predicates. This must be correct with respect to its specification. Since each module has to denote a complete theory, every predicate in the module must be completely defined within the module. This disallows the more flexible system described in the current paper.

The paper[1] is organised as follows. In the next section, Godel's module system is explained. We show how this can be extended to include parameterised modules. In section 3, we give a number of definitions associated with a module-free typed logic program which are needed later. Then, in section 4, we provide a formal definition of a modular program with parameterised modules. Finally, section 5 outlines the intended semantics for such a program.

2 The Godel Language

The module system in Godel is best explained by means of an example[2].

```
EXPORT    The JonesFamily.
BASE        Person.
COISTAIT  Eve,Pat.Bob,Tim,Mary: Person.
PREDICATE Mother,Father: Person*Person.

LOCAL       TheJonesFaaily.
Father(Bob,Pat).
Mother(Pat,Mary)•

EXPORT      TheJonesRels.
IMPORT      TheJonesFamily.
PREDICATE Anc: Person * Person.

LOCAL       TheJonesRels♦
PREDICATE Par: Person * Person.
Par(x,y) <- Mother(x,y) V Father(x,y).
Anc(x.y) <- Par(x,z) * Anc(z,y).
Anc(x.y) <- Par(x,y).
```

In both of the above modules there are a number of *declarations* and *statements.* The statements are formulas in the language defined by the declarations. There are two kinds of declaration: language and module. The language declarations begin with a key word that indicates the category[8] of symbol being declared. In The JonesFaaily module, Person is declared to be a base type; Eve, Pat, etc., are declared to be constants of type Person; Mother and Father are declared to be predicates with arguments of type Person. In Godel, a symbol name (for a given arity and category) must have at most one declaration in a module.

Each module is in two parts called *export* and *local* A part begins with a module declaration stating whether it is the export or local part and the name of the module. The export part contains language declarations for symbols that can be used in this module and also in other modules that import it. Thus the type Person can be used in TheJonesRels as well as in either part of The JonesFamily. Symbols declared in the local part of a module are only available for use within this part. Hence, since TheJonesRels declares Par in the local part, Par cannot be used outside this module. Statements are only allowed in the local part of a

[l]A version of this paper with several longer examples showing the use of parameterised modules is available as a technical report. [Hill, 1993]

[3]Par, Rela, Anc are short for *Parent, Relations, Ancestor.*

[3]The categories are: type constructor, function, or predicate. A base, constant, or proposition is regarded as a constructor, function, or predicate, respectively, of arity 0.

module. These define the predicates declared in either part of the module. The JonesRels also has a module declaration that begins with the key word IMPORT. This makes all the symbols declared in the export part of The JonesFamily available for use in The JonesRels.

The example illustrates the many-sorted types in Godel. However, in Godel, we can also define generic functions and predicates. A common example of such a data structure is a list, which is defined to be a list of terms of a certain type, but the particular type to be used is not specified. For example, Godel provides a system module Lists. This module exports the type constructor List, constant Mil, function Cons, and predicate Member with language declarations:

```
COISTRUCTOR  List/1.
COISTANT     Mil: List(a).
FUNCTION     Cons: a*List(a)->List(a).
PREDICATE    Meaber: a*List(a).
```

If the language included the base type Int, then we have the types List(a), List(List(a)), List(Int), List(List(Int)), etc. The tuples of types in a declaration is called a declared type. Other types for Wil, Cons, and Member can be obtained from their declared types by means of type substitutions. Thus Wil also has types List(Int) and List(List(Int)), Cons has types Int * List(Int) -> List(Int) and List(Int) * List(List(Int)) -> List(List(Int)), and Member has types Int List(Int) and List(Int) * List(List(Int)).

A *definition* of a constructor *C* is a set of function declarations with range type of the form C(t1............,tn) (or *C,* if the arity n is 0). A *definition* of a predicate *P* is a set of statements with *P* in the head.

A *Godel program* for a module *rn* (called the *main module)* is the smallest set of modules that includes m and is closed wrt the modules named in the import declarations. The program must satisfy the following three conditions.

MI The module names can be partially ordered so that if m^1 occurs in an import declaration in a module named m then m' < m.

M2 Every symbol appearing in (the export part of) a module, must be declared in or imported into (the export part of) the module*

M3 Each constructor or predicate with a non-empty definition in a module must be declared in that module.

These conditions enable independent compilation and protect procedures defined in one module from being modified by another. The set of modules {TheJonesFamily, The JonesRels} form a Godel program.

In the example, a module containing general rules about family relations was forced, by the module system, to contain a declaration importing a module defining a specific family thereby preventing its reuse with other families. Thus we propose to modify the above language to allow parameterised modules. In the next example, a Rels module is parameterised wrt the base Person and predicates Mother and Father. Note that, here it is the TheJones module that imports the Rels module, whereas in the previous example, the importation was in the opposite direction.

```
EXPORT     TheJones.
IMPORT     Rels(Jones,Ma,Pa).
BASE       Jones.
PREDICATE  Ma,Pa: Jones * Jones.
COISTAIT   Eve,Pat,Bob,Tin,Mary:  Jones.

LOCAL      TheJones.
Pa(Bob,Pat).
Ma(Pat,Mary).

EXPORT     Rels(Person,Mother,Father).
BASE       Person.
PREDICATE  Mother,Father: Person*Person.
PREDICATE  Anc: Person * Person.

LOCAL      Rels(Person,Mother.Father).
IMPORT     Trans(Person, Par).
PREDICATE  Par: Person * Person.
Par(x,y)  <- Mother(x,y)\/Father(x,y).
Anc(x,y)  <-  Tr(x,y).

EXPORT     Trans(Point,Connect).
BASE       Point.
PREDICATE  Connect: Point * Point.
PREDICATE  Tr: Point * Point.

LOCAL      Trans(Point, Connect).
Tr(x,y)  <- Connect(x,y).
Tr(x,y)  <- Connect(x,z) ft Tr(z,y).
```

The module name that follows the key words EXPORT and LOCAL consists of an *identifier* with 0 or more symbols as arguments. The set of declarations for these symbols (which must be in the export part of the module) is called the *signature* of the module. For example, Rels (Person, Mother, Father) is a module name with identifier Rels and signature

```
BASE       Person.
PREDICATE  Mother,Father: Person*Person.
```

Symbols that are declared in a module but are not in the signature are said to be *completely specified* by the module. For example, the base Jones is completely specified in TheJones.

The written module is the *initial* module, /n-*stances* of these modules can be obtained by substituting new symbols for symbols occurring in the module name. The substituted symbols must be distinct from symbols completely specified by the initial module. Thus the following module is imported into Rels(Person, Mother, Father).

```
EXPORT     Trans(Person,Par).
BASE       Person.
PREDICATE Par: Person * Person.
PREDICATE Tr: Person * Person.

LOCAL      Trans(Person,Par).
Tr(x,y) <- Par(x_f y).
Tr(x,y) <- Par(x,z) & Tr(z.y).
```

Note that if a symbol is completely specified in a module it is completely specified in every module that is an instance of this module. Thus the predicate Tr is completely specified by Trans(Point,Connect) and Trans(Person,Par). (Trans(Person,Tr) could not occur in an import declaration in a module since Tr is completely specified in Trans (Person, Connect).)

We define a *modular program* in a similar way to the definition of a Godel program above. Informally, it is a set of initial modules with a main module, closed wrt the identifiers in the import declarations and satisfying similar module conditions to those given above.

M I* The identifiers in the module names can be partially ordered so that if I' is an identifier in an import declaration in a module with identifier I then $J^7 < /$.

M2* Every symbol name appearing in (the export part of) a module m, must either be declared in (the export part of) *m* or be completely specified by the export part of a module that is imported into (the export part of) m.

M3* Each constructor or predicate name declared in or imported into a module and completely specified by an imported module n may only have a non-empty definition in n or in imported modules that are also imported into n.

The set of modules
{TheJones,
Rels (Person, Mother, Father),
Trans (Point, Connect) },
together form a modular program for the TheJones.

The previous example shows the way the module system works where each module imports no more than one module and each of the constructors and predicates is completely defined within a module. However, the parameterised module system allows for multiple inheritance and also for a predicate or constructor definition to be split between several modules. To illustrate this, we define another family named Hill. Due to marriage between the families, we create a new family with name HilUones.

```
EXPORT    TheHillJonss(HillJones,Ma,Pa).
BASE      HillJones.
PREDICATE Ma, Pa: HilUones * Hill Jones.
IMPORT    TheJones(HillJones,Na,Pa),
          TheHills (HilJones ,Ma,Pa).
COISTAIT Pas: HilUones.

LOCAL TheHill Jones (HilUones, Ma, Pa).
Ma(Mary,Pam).
Pa(Tom,Paa).

EXPORT     TheHills(Hill,Ma,Pa).
BASE       Hill.
PREDICATE Ma,Pa: Hill * Hill.
IMPORT     Rsls(Hill,Ma,Pa).
COMSTAIT   Robin,Jill,Tom: Hill.

LOCAL       TheHills(Hill,Ma,Pa).
Pa(Robin,Ton).
```

The set of modules
{TheHillJones(HillJones,Ma,Pa),
TheHills(Hills,Ma,Pa),
TheJones(Jones,Ma,Pa),
Rels(Person,Mother,Father),
Trans(Point,Connect)}
forms the modular program with main module TheHillJones. A goal for the program is

<- Anc(Bob,x) & Anc(Robin,x).

3 Typed Logic Programs

The Gödel language is based on typed first order logic, where each non-logical symbol has a language declaration. This associates the symbol with a tuple of symbols in another language, called the *type language*.

In a type language for a parametric type system as used in Gödel, the types are structured expressions defined over disjoint sets of constructors and parameters. Each constructor has an arity associated with it. Thus the *constructor declaration* C/n ($n \geq 0$) assigns the arity n to a constructor C. A type in the type language $\mathcal{T}$ defined over a set of constructor declarations $\mathcal{C}$ is defined recursively, to be either a parameter or of the form $C(t_1, \ldots, t_n)$, where $C/n \in \mathcal{C}$ and $t_1, \ldots, t_n$ are types in $\mathcal{T}$.

Since, in this paper, we need to define type languages in a modular program, we must have a means of constructing a new type language from a set of existing type languages. Let $\mathcal{T}, \mathcal{T}_1, \ldots, \mathcal{T}_k$ be type languages defined over the sets of constructor declarations $\mathcal{C}, \mathcal{C}_1, \ldots, \mathcal{C}_k$, respectively, using the same set of parameters. Then we say that $\mathcal{T}$ is the *join* of the set $\{\mathcal{T}_1, \ldots, \mathcal{T}_k\}$ if $\mathcal{C} = \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_k$. If $\mathcal{C}_0$ is a

set of constructors and $\mathcal{T}'$ a type language defined on the set $\mathcal{C} \cup \mathcal{C}_0$, then $\mathcal{T}'$ is the *extension* of $\mathcal{T}$ using $\mathcal{C}_0$.

A *first order typed language* $\mathcal{L}$ based on a type language $\mathcal{T}$ consists of a set of formulas, where the formulas are defined wrt a set of function declarations $\mathcal{F}$ over $\mathcal{T}$, a set of predicate declarations $\mathcal{P}$ over $\mathcal{T}$, and a set of variables V.

The *function declaration* $F : \tau_1 * \cdots * \tau_n \rightarrow \tau$ assigns a nonempty tuple of types $\tau_1, \ldots, \tau_n, \tau$ called the *declared type* to a function F of *arity* n. It is assumed that every parameter in the tuple occurs in the last type. The set of functions in $\mathcal{F}$ must be disjoint from the set of variables V.

The *predicate declaration* $P : \tau_1 * \cdots * \tau_n$ assigns a tuple of types $(\tau_1, \ldots, \tau_n)$ called the *declared type* to a predicate P of *arity* n.

If Θ is a type substitution, the function F also has the *type* $(\tau_1\Theta, \ldots, \tau_n\Theta, \tau\Theta)$, *range type* $\tau\Theta$, and *domain type* $(\tau_1\Theta, \ldots, \tau_n\Theta)$. Similarly, the predicate P has *type* $(\tau_1\Theta, \ldots, \tau_n\Theta)$.

Using sets of function and predicate declarations $\mathcal{F}$ and $\mathcal{P}$ and a set of assignments of types to variables, called a *variable typing*, we define a term with its associated type and an atom.

1. A variable x is a term whose type is assigned to x by the variable typing.
2. If a function F has type $(\tau_1, \ldots, \tau_n, \tau)$ and $A_1, \ldots, A_n$ are terms of types $\tau_1, \ldots, \tau_n$, then $F(A_1, \ldots, A_n)$ is a term of type τ.
3. If a predicate P has type $(\tau_1, \ldots, \tau_n)$ and $A_1, \ldots, A_n$ are terms of types $\tau_1, \ldots, \tau_n$, then $P(A_1, \ldots, A_n)$ is an atom.

It is now straightforward to define a formula in $\mathcal{L}$ (see [Hill and Topor, 1992]).

Language declarations with the same category and arity in $\mathcal{L}$ are *similar*. If each symbol has a unique declared type, then $\mathcal{L}$ is *universal*. In this paper we assume that all the languages are universal. Since the category and arity can be used to distinguish symbols with the same name, we require similar declarations that are not identical to have distinct symbol names.

For each $i \in \{1, \ldots, k\}$, let $\mathcal{L}_i$ be a first order typed language defined wrt the sets of function and predicate declarations $\mathcal{F}_i$ and $\mathcal{P}_i$. Suppose also that $\mathcal{L}$ is a first order language defined wrt the sets of function and predicate declarations $\mathcal{F}$ and $\mathcal{P}$. Then $\mathcal{L}$ is the *join* of $\{\mathcal{L}_1, \ldots, \mathcal{L}_k\}$ if $\mathcal{F} = \mathcal{F}_1 \cup \cdots \cup \mathcal{F}_k$ and $\mathcal{P} = \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_k$. If $\mathcal{F}_0$ and $\mathcal{P}_0$ are sets of function and predicate declarations, and $\mathcal{L}'$ a first order language defined wrt $\mathcal{F}_0 \cup \mathcal{F}$ and $\mathcal{P}_0 \cup \mathcal{P}$, then $\mathcal{L}'$ is the *extension* of $\mathcal{L}$ using $\mathcal{F}_0$ and $\mathcal{P}_0$.

Let $\mathcal{L}$ be a universal first order language defined wrt the sets $\mathcal{F}$ and $\mathcal{P}$ of function and predicate declarations and $P : \tau_1 * \cdots * \tau_n$ a predicate declaration in $\mathcal{P}$. A *statement* for P is a formula in $\mathcal{L}$ of the form $P(A_1, \ldots, A_n)$ or $P(A_1, \ldots, A_n) \leftarrow W$ with some assignment V of types to the variables in the statement, where W is a formula and $A_1, \ldots, A_n$ are terms of type $\tau_1, \ldots, \tau_n$ using $\mathcal{F}$ and V. $P(A_1, \ldots, A_n)$ is called the *head* of the statement.

A *module-free program* Π consists of a set of language declarations defining a type language $\mathcal{T}$ (called the *type language of* Π), a typed first order language $\mathcal{L}$ based on $\mathcal{T}$ (called the *language of* Π), and a set of statements in $\mathcal{L}$. A *goal* G for Π is either the empty clause or of the form $\leftarrow W$ where W is a formula in the language of Π.

The declarative semantics of a module-free (typed) logic program is based on a typed form of the Clark completion of logic programs using an equality predicate that has declared type (α, α) where α is a parameter. Details can be found in [Hill, 1992] and [Hill and Topor, 1992].

4 A Modular Program: Definition

We now define a *module* $\mu(m)$[4] and a *modular program* Π_m with *main module* m[4].

We first define a *simple* module m which is a module with no import declarations and has *import depth* 0. It consists of a set $\{m, Exp, Loc\}$ where m is the name of the module, The name m consists of an identifier followed by 0 or more symbols. These symbols must have their constructor $\mathcal{C}_m$, function $\mathcal{F}_m$, and predicate $\mathcal{P}_m$ declarations in Exp. The set $\mathcal{C}_m \cup \mathcal{F}_m \cup \mathcal{P}_m$ is called the *signature* of m. $Exp = \{\emptyset, \mathcal{C}_{Exp}, \mathcal{F}_{Exp}, \mathcal{P}_{Exp}\}$, and $Loc = \{\emptyset, \mathcal{C}_{Loc}, \mathcal{F}_{Loc}, \mathcal{P}_{Loc}, S\}$. Exp is the *export part* and Loc is the *local part* for m. $\mathcal{C}_{Exp}$ and $\mathcal{C}_{Loc}$ are disjoint sets of constructor declarations. $\mathcal{C}_{Exp} - \mathcal{C}_m$ defines the *export type language* for m and $\mathcal{C}_{Exp} \cup \mathcal{C}_{Loc}$ (resp., $\mathcal{C}_{Exp}$) defines the *type language* for m (resp., export part of m). $\mathcal{F}_{Exp}$ and $\mathcal{P}_{Exp}$ are sets of function and predicate declarations in the type language for the export part of m. $\mathcal{F}_{Loc}$ and $\mathcal{P}_{Loc}$ are sets of function and predicate declarations in the type language for m. $\mathcal{F}_{Exp} - \mathcal{F}_m$ and $\mathcal{P}_{Exp} - \mathcal{P}_m$ define the *export language*. $\mathcal{F}_{Exp} \cup \mathcal{F}_{Loc}$ and $\mathcal{P}_{Exp} \cup \mathcal{P}_{Loc}$ define the *language* for m (which is assumed to be universal). S is a set of program statements in the language for m. A symbol s with a declaration in m is *completely specified by* m if s does not occur in the name m.

The module, as written is called an *initial* simple module. *Instances* of initial modules can be obtained by means of module substitutions. A module substitution θ for a module m is a substitution whose domain is a subset of the

[4]We use the module name m instead of $\mu(m)$ if it is clear from the context which module the name m is referring to.

symbols in m and whose range is any set of names not completely specified by m. This set must be such that $\mu(m)\theta$ is a (simple) module. In particular, the language for $\mu(m)\theta$ must be universal. Thus, if symbols s_1 and s_2 have similar declarations d_1 and d_2, respectively, in m and $s_1\theta = s_2\theta$, then $d_1\theta$ and $d_2\theta$ must be the same.

The modular program Π_m for m is the singleton set $\{\mu(m)\}$.

Suppose we have a set Δ of initial modules with distinct module identifiers such that, for each module $i \in \Delta$, a modular program Π_i for i is in Δ and the import depth of i is $< d$. A module m is a set $\{m, Exp, Loc\}$ where $Exp = \{I_{Exp}, C_{Exp}, \mathcal{F}_{Exp}, \mathcal{P}_{Exp}\}$, and $Loc = \{I_{Loc}, C_{Loc}, \mathcal{F}_{Loc}, \mathcal{P}_{Loc}, S\}$. The name m consists of an identifier not in Δ followed by 0 or more symbols. These symbols must have their constructor $\mathcal{C}_m$, function $\mathcal{F}_m$, and predicate $\mathcal{P}_m$ declarations in Exp. The set $\mathcal{C}_m \cup \mathcal{F}_m \cup \mathcal{P}_m$ is called the *signature* of m. Exp is the *export part* and Loc is the *local part* for m. I_{Exp} and I_{Loc} are sets of module names. Every symbol in I_{Exp} (resp., I_{Loc}) that is not completely specified by an imported module should have a language declaration in Exp (resp., $Exp \cup Loc$). For each $i \in I = I_{Exp} \cup I_{Loc}$, there must be a (unique) module $\mu(i')$ in Δ and a module substitution θ_i such that $i = i'\theta_i$ and $\mu(i')\theta_i$ is a module (denoted by $\mu(i)$). The *import depth* of m is one more than the maximum of the import depths of modules $\mu(i)$ where $i \in I$. We say that a module i is *available to* m (resp., the export part of m) if either $i = m$ or there is a module $j \in I$ (resp., $j \in I_{Exp}$) and i is available to the export part of j.

C_{Exp} and C_{Loc} are disjoint sets of constructor declarations. Let $\mathcal{T}^I$ (resp., $\mathcal{T}^I_{Exp}$) be the join of the export type languages for i, $i \in I$ (resp., $i \in I_{Exp}$). The *export type language* for m is the extension of $\mathcal{T}^I_{Exp}$ using $C_{Exp} - \mathcal{C}_m$. The *type language* for m (resp., the export part of m) is the extension of $\mathcal{T}^I$ using $C_{Exp} \cup C_{Loc}$ (resp., C_{Exp}). $\mathcal{F}_{Exp}$ and $\mathcal{P}_{Exp}$ are sets of function and predicate declarations in the type language for the export part of m. $\mathcal{F}_{Loc}$ and $\mathcal{P}_{Loc}$ are sets of function and predicate declarations in the type language for m. Let $\mathcal{L}^I$ (resp., $\mathcal{L}^I_{Exp}$) be the join of the export languages for i, $i \in I$ (resp., $i \in I_{Exp}$). The *export language* for m is the extension of $\mathcal{L}^I_{Exp}$ using $\mathcal{F}_{Exp} - \mathcal{F}_m$ and $\mathcal{P}_{Exp} - \mathcal{P}_m$ and the *language* for m is the extension of $\mathcal{L}^I$ using $\mathcal{F}_{Exp} \cup \mathcal{F}_{Loc}$, $\mathcal{P}_{Exp} \cup \mathcal{P}_{Loc}$. The language for m must be universal. Thus, if module i is available to m and a symbol s has similar declarations d in m and d_i in i, then d and d_i must be the same. Similarly, if modules i and j are available to a module and a symbol s has similar declarations d_i in i and d_j in j, then d_i and d_j must be the same. Finally, the set S is a set of statements in the language for m such that every predicate in the head has a declaration in $\mathcal{P}_{Exp} \cup \mathcal{P}_{Loc}$.

We say that a symbol s is *completely specified by* m if s does not occur in m and there is a language declaration for s in $\mu(m)$.

The module as written is called the *initial* module. *Instances* of initial modules can be obtained, as for simple modules, by means of module substitutions.

The program Π_m is the union of $\{\mu(m)\}$ with the union of the programs $\Pi_{i'}$, for each initial module i' in Δ such that, for some module substitution θ, $i = i'\theta$ and $i \in I$. The *language* of Π_m is that of m.

5 A Modular Program: Semantics

In this section we define a mapping called *a demodularisation* from a modular program to a module-free program. The semantics of a modular program is then defined to be the semantics of its demodularisation.

A symbol, declared in or imported into an initial module m, is assumed to be distinct from any other symbol that is completely specified in an imported module. If the programming language (such as Gödel) allows overloading, then the overloaded names of distinct symbols declared or imported into the initial module must be standardised apart before the demodularisation mapping is applied.

The two components of a demodularisation are the generating of imported modules from initial modules and the standardising apart of names in the local parts of the imported modules. The demodularisation of a module is defined by induction on its import depth.

We first define an *associated simple module* $\sigma(m)$ for a modular program Π_m for an initial module $\mu(m)$ by induction on its import depth. It is assumed that the character '!' does not occur within the declared symbols.

1. If $\mu(m)$ is an simple module, then $\mu(m)$ is the associated simple module for Π_m.
2. Suppose $\mu(m)$ has import depth > 0. For each initial module $\mu(i')$ such that, for some module substitution θ, $i = i'\theta$ occurs in an import declaration in $\mu(m)$, obtain the associated simple module $\sigma(i')$.
3. For each module name i in an import declaration in (the export part of) $\mu(m)$ construct an *imported simple module* for (the export part of) $\mu(m)$ as follows. If i' is an initial module name and θ is a module substitution such that $i = i'\theta$, let $\sigma(i)$ denote $\sigma(i')\theta$. Then, for each symbol s in $\sigma(i)$,

if a module j is available to $\mu(i)$ but not available to the export part of $\mu(i)$ and s is completely specified by j, replace each occurrence of s in $\sigma(i)$ by $j!s$.

4. Let $\sigma(m)$ be the module with name and signature of the module $\mu(m)$ and containing all the language declarations and statements in $\mu(m)$ together with all the language declarations and statements in the imported simple modules for m. The export part of $\sigma(m)$ contains all the declarations in the export part of $\mu(m)$ and the export parts of the imported simple modules for the export part of $\mu(m)$.

The demodularised program for the modular program Π_m is the program obtained from $\sigma(m)$ by removing the module declarations and combining the local and export parts.

It can be shown that $\mu(m)$ together with its imported simple modules is a modular program of import depth 1 and $\sigma(m)$ is a simple module whose language is the same as that of Π_m.

We illustrate the demodularisation using the modular program for TheJones in section 2. First we give the associated simple module σ(Rels(Person,Mother,Father)).

```
EXPORT      Rels(Person,Mother,Father).
BASE        Person.
PREDICATE   Mother,Father: Person*Person.
PREDICATE   Anc : Person * Person.

LOCAL       Rels(Person,Mother,Father).
PREDICATE
  Par: Person*Person;
  Trans(Person,Par)!Tr: Person*Person.
Par(x,y) <- Mother(x,y) \/ Father(x,y).
Anc(x,y) <- Trans(Person,Par)!Tr(x,y).
Trans(Person,Par)!Tr(x,y) <-
  Par(x,z) & Trans(Person,Par)!Tr(z,y).
Trans(Person,Par)!Tr(x,y) <-
  Par(x,y).
```

The imported simple module Rels(Jones, Ma, Pa) for TheJones is obtained by applying the module substitution {Person/Jones, Mother/Ma, Father/Pa} to the above module and then replacing each occurrence of Par by Rels(Jones,Ma,Pa)!Par. We conclude the paper with the demodularisation of the modular program with main module TheJones.

```
BASE        Jones.
CONSTANT    Eve,Pat,Bob,Tim,Mary: Jones.
PREDICATE   Ma,Pa: Jones * Jones.
PREDICATE   Anc: Jones * Jones.
PREDICATE
    Rels(Jones,Ma,Pa)!Par: Jones*Jones;
    Trans(Jones,Rels(Jones,Ma,Pa)!Par)!
                          Tr: Jones*Jones.
Pa(Bob,Pat).
Ma(Pat,Mary).
Rels(Jones,Ma,Pa)!Par(x,y) <-
    Ma(x,y) \/ Pa(x,y).
Anc(x,y) <-
    Trans(Jones,Rels(Jones,Ma,Pa)!Par)!
                                Tr(x,y).
Trans(Jones,Rels(Jones,Ma,Pa)!Par)!
                                Tr(x,y) <-
    Rels(Jones,Ma,Pa)!Par(x,z) &
    Trans(Jones,Rels(Jones,Ma,Pa)!Par)!
                                Tr(z,y).
Trans(Jones,Rels(Jones,Ma,Pa)!Par)!
                                Tr(x,y) <-
    Rels(Jones,Ma,Pa)!Par(x,y).
```

References

[Antoniou and Sperschneider, 1992] G. Antoniou and V. Sperschneider. Modularity for logic programs. In *Proceedings of ALPUK-92,* pages 3-14. City University, London, 1992.

[Goguen and Meseguer, 1984] J.A. Goguen and J. Meseguer. EQLOG: Equality, types and generic modules for logic programming. *Journal of Logic Programming,* 1:68-131, 1984.

[Hill and Lloyd, 1992] P.M. Hill and J.W. Lloyd. The Godel programming language. Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, UK, 1992.

[Hill and Topor, 1992] P.M. Hill and R.W. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming,* pages 1-62. MIT Press, 1992.

[Hill, 1992] P.M. Hill. Data structures and typed logic programs. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence,* Vienna, Austria, pages 109-113. John Wiley & Sons, 1992.

[Hill, 1993] P.M. Hill. A parameterised module system for constructing typed logic programs. Technical Report 93.12, School of Computer Studies, 1993.

[Miller, 1986] D.A. Miller. A theory of modules for logic programming. In *IEEE Symposium on Logic Programming,* pages 106-115,1986.

[O'Keefe, 1985] R.A. O'Keefe. Towards an algebra for constructing logic programs. In *Proceedings of the Symposium on Logic Programming,* Boston, pages 152-160. IEEE, 1985.

[Sannella and Wallen, 1992] D.T.Sannella and L.A. Wallen. A calculus for the construction of modular Prolog programs. *Journal of Logic Programmimg,* 12(1 & 2):147-177, 1992.