A Generic Module System for Web Rule Languages: Divide and Rule

Uwe Aßmann[1], Sacha Berger[2], François Bry[2], Tim Furche[2], Jakob Henriksson[1], and
Paula-Lavinia Pătrânjan[2]

[1] Technische Universität Dresden,
Nöthnitzer Str. 46, D-01187 Dresden, Germany
`http://www-st.inf.tu-dresden.de/`
[2] Institute for Informatics, University of Munich,
Oettingenstr. 67, D-80538 München, Germany
`http://www.pms.ifi.lmu.de/`

Abstract. An essential feature in practically usable programming languages is the ability to encapsulate functionality in reusable modules. Modules make large scale projects tractable by humans. For Web and Semantic Web programming, many rule-based languages, e.g. XSLT, CSS, Xcerpt, SWRL, SPARQL, and RIF Core, have evolved or are currently evolving. Rules are easy to comprehend and specify, even for non-technical users, e.g. business managers, hence easing the contributions to the Web. Unfortunately, those contributions are arguably doomed to exist in isolation as most rule languages are conceived without modularity, hence without an easy mechanism for integration and reuse. In this paper a generic module system applicable to many rule languages is presented. We demonstrate and apply our generic module system to a Datalog-like rule language, close in spirit to RIF Core. The language is gently introduced along the EU-Rent use case. Using the Reuseware Composition Framework, the module system for a concrete language can be achieved almost for free, if it adheres to the formal notions introduced in this paper.

1 Introduction

Modules are software units that group together parts of different programs and knowledge (or data structures) and that usually serve a specific purpose. For any practically applied programming language modules are an essential feature, as they make large scale projects tractable by *humans*. Modules not only facilitate the integration of existing applications and software into complex projects, they also offer a flexible solution to application development where part of the application logic is subject to change.

The work presented in this article advocates for the need and advantages of modularizing rule-based languages. The rule-based paradigm offers elegant and flexible means of application programming at a high-level of abstraction. During the last couple of years, a considerable number of initiatives have focused on using rules on the Web (e.g. RuleML, R2ML, XSLT, RIF Core etc.).

However, unless these rule languages are conceived with proper support for encoding knowledge using modular designs, their contributions to the Web are arguably

doomed to exist in isolation. Hence, with no easy mechanism for integration of valuable information. Many of the existing rule languages on the Web do not provide support for such modular designs. The rational behind this is the focus on the initially more crucial concerns relating to the development of the languages.

The main difference between the module system presented here and previous module system work is, that our approach focuses on genericity. That is, the approach is applicable to many rule languages with only small adaption to syntactic and semantic properties of the language to be supported with constructs for modularization. The concept is based on rewriting modular programs into semantically equivalent non-modular programs, hence not forcing the evaluation of the hosting language to be adapted to some operational module semantics. This approach not only enables reuseability for current languages, it is arguably also applicable to forthcoming rule languages. The presented module system is hence arguably the embodiment of reuseability—it not only supports users of languages to design programs in a modular fashion, it also encourages tool and language architects to augment their rule languages by reusing the abstract module system. Last but not least: the module system is *easy* in several aspects. First, it is very easy for language developers to apply due to the employment of reduction semantics of a given modularized rule language to its un-modular predecessor. Second, the reduction semantics is kept simple. There is just one operator, yet it is sufficient to model a quite rich modular landscape including visibility, scoped module use, parametric modules etc. Third, the implementation of the abstract module system can be achieved using existing language composition tools, for example, Reuseware.[3] A concrete modularized language is achieved by mere instantiation of the abstract implementation, making the implementation of the abstract module system fast and easy.

The main contribution of our work is a conceptual framework for modules in rule languages. We identify requirements defining a class of rule languages that can be extended with the proposed module system. The conceptual framework provides abstract language constructs for a module system, which are to extend the syntax of the hosting rule language. The principles of modularization and the main language constructs are gently introduced using Datalog as an example rule language.

The paper is structured as follows. Section 2 introduces the rule language Datalog and demonstrates the usability of modularization of its rules via a use-case. Section 3 describes and motivates the main requirements of any rule language in order for it to be applicable to the underlying module framework. Section 4 describes the underlying module algebra for the framework. Section 5 compares our work to other module approach and Section 6 concludes the paper with some final remarks.

2   Module Extension by Example

Rules in a particular language are usually specified in a *rule-base*, a finite set of (possibly) related rules. Recall from Section 1 that we focus our work on so-called deductive rules. A *deductive rule* consists of a *head* and a *body* and has the following form:

$$head \text{ :- } body$$

---

[3] `http://reuseware.org`

where *body* and *head* are sets of atomic formula of the rule language. The formula of each set are composed using (language specific) logical connectives. The implication operator :- binds the two rule parts together. As such, rules are read "if the body holds, then the head also holds". Usually, rule parts share variables: the variable substitutions obtained by evaluating the *body* are then used in constructing new data in the *head*.

In the following we extend a concrete rule language with modules. By following our proposed approach, which is formalized in Section 4, one can afford to abstract away from a particular data model or specific capabilities supported by a rule language and - most important - not to change the semantics of the language.

2.1 Datalog as an Example Rule Language

In the introductory part we mentioned Datalog as an example rule language based on deductive rules. Datalog is a well-known database query language often used for demonstrating different kinds of research results (e.g. query optimization). Datalog-like languages have been successfully employed, e.g. for Web data extraction in Lixto[4]. The strong similarities between Datalog and RIF Core suggest that the ideas followed for modularizing Datalog could be also applied to RIF Core.

In Datalog, the *head* is usually considered to be a singleton set and the *body* a set of Datalog *atoms* connected via conjunction ($\wedge$). A Datalog atom is an *n*-ary predicate of the form $p(a_1, \ldots, a_n)$, where $a_i (1 \leq i \leq n)$ are constant symbols or variables. As such, a Datalog rule may take the following form:

```
─────── A Simple Datalog Rule ───────
gold-customer(Cust) :- rentals(Cust,Nr,2006), Nr > 10.
```

to be understood as defining that *Cust* is a gold customer if the number *Nr* of car rentals *Cust* made in 2006 is greater than 10.

Rules are associated a semantics, which is specific to each rule language and cannot be described in general terms. For the case of Datalog, semantics is given by definition of a least Herbrand model of a rule-set. We don't go into details regarding the semantics, since our work on modularizing Datalog preserves the language's semantics.

Datalog, and more general deductive rules, infer new knowledge (called intensional knowledge) from existing, explicitly given knowledge. As already recognized [12], there is a need to 'limit' the amount of data used in performing inference on the Web – a big and open source of knowledge. Thus, the notion of *scoped inference* has emerged. The idea is to perform inference within a scope of explicitly given knowledge sources. One elegant solution for implementing scoped inference is to use *modules* for separating the knowledge. In such a case, the inference is performed within a module. Since inference is essential on the Web and, thus, modules for rule languages such as Datalog, let's see how we could modularize Datalog!

2.2 Module Extension for Datalog

This section gives a light introduction to modularizing a rule language such as Datalog that should ease the understanding of the formal operators proposed in Section 3. We

---

[4] Data extraction with Lixto, `http://www.lixto.com/li/liview/action/display/frmLiID/12/`

consider as framework for our examples the EU-Rent[5] case study, a specification of business requirements for a fictitious car rental company. Initially developed by Model Systems Ltd., EU-Rent is promoted by the business rules community as a scenario for demonstrating capabilities of business rules products.

The concrete scenario we use for showing advantages of introducing modularization in rule languages is similar to the use case for rule interchange 'Managing Inter-Organizational Business Policies and Practices', published by the W3C Rule Interchange Format Working Group in the 2nd W3C Working Draft of 'RIF Use Cases and Requirements'.[6] The car rental company EU-Rent operates in different EU countries. Thus, the company needs to comply with existing EU regulations and each of its branches needs also to comply with the regulations of the country it operates in.

The EU-Rent company heavily uses rule-based technologies for conducting its business. This was a straightforward choice of technology from the IT landscape, since rule languages are more than suitable for implementing company's policies. Moreover, EU regulations are also given as (business) rules.
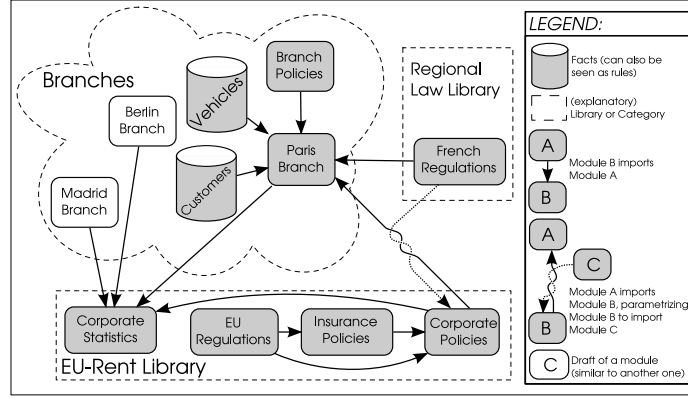
Different sets of rules come into play for most of the company's rental services, such as advance reservations for car rentals or maintenance of cars at an EU-Rent branch. A set of rules implement, as touched on above, the company's policies (e.g. that *the lowest price offered for a rental must be honored*). These rules are used by each of the EU-Rent branches. Another set of rules implements the policies used at a particular EU-Rent branch, as they are free to adapt their business to the requirements of the market they address (of course, as long as they remain in conformance with the EU-Rent company-level rules). As is the case for EU regulations, EU-Rent branches might need to comply with existing national regulations – these represent actually an extra set of rules to be considered.

We have illustrated so far a typical scenario for data integration. The sets of rules our EU-Rent company needs to integrate for its services may be stored locally at each branch or in a distributed manner (e.g. company level rules are stored only at EU-Rent Head Quarter and EU and national rule stores exist on different servers for the corresponding regulations). Rules might change at the company level and regulations might also change both at EU and at national level. So as to avoid the propagation of such changes every time they occur, a distributed and modularized approach to the EU-Rent implementation would be a suitable solution.

An architectural overview of the scenario described so far is given above. The overview sketches a possible modularization of rules employed by the EU-Rent company. Modules are represented here as boxes. An example EU-Rent branch, the Paris branch, subdivides its vehicle and customer data as well as its policies into different modules, hence separating concerns for gaining clarity and ease maintenance. The module `ParisBranch` *imports* the defined modules and, thus, uses their rules together with those defined in `ParisBranch`. Such module dependencies (i.e., *import*s) are indicated by arrows between boxes.

---

[5] EU-Rent case study, `http://www.businessrulesgroup.org/egsbrg.shtml`

[6] W3C Working Draft of 'RIF Use Cases and Requirements',`http://www.w3.org/TR/2006/WD-rif-ucr-20060710/`

Fig. 1 EU-Rent Use Case: Module Structure

Note that modules can be imported and their rules can be defined as *private* or *public*. Pivate rules are not visible, i.e. the knowledge inferred by such rules can not be accessed directly in modules importing them. A statement `private import M2` in a module `M1` makes all rules of `M2` private for the modules importing `M1`. Rules defined as public can be used directly in modules importing them. By importing a module `M` as public makes its (public) rules visible in all modules importing `M`.

In the following we specify an excerpt of the module `ParisBranch` in Datalog:

```
 ParisBranch 
import private Vehicles
import private Customers
import private BranchPolicies
import private CorporatePolicies ( regional-law = "FrenchRegulations" )
public vehicle(X) :- voiture(X).
public vehicle(X) :- bicyclette(X).
```

The `CorporatePolicies` module, imported in the module example above, can be considered as the core of the depicted architecture. It aggregates various (domain specific) modules like `EURegulations` and `InsurancePolicies`, which implements relevant European insurance regulations. Branches import the `CorporatePolicies` module and comply thus to corporate policy. By design, the company branches use the corporate policies module as a common access point also to local regulations. As the previous example shows, the Paris branch (and other EU-Rent branches too) parameterizes the module `CorporatePolicies` using as parameter the local regulations the branch needs to comply to. More concrete, the parameter `regional-law` is instantiated with the name of the module implementing the specific regulations, in this case `FrenchRegulations`. For this to work, the module `CorporatePolicies` should be defined as follows:

```
 CorporatePolicies 
declare parameter regional-law
import public InsurancePolicies
import private EURegulations
import public parameter regional-law
... // rules defining the Eu-Rent policies
```

In the following we turn our attention to another module depicted in our architecture, the corporate operational module `CorporateStatistics`. This module not only imports `CorporatePolicies`, but also all branch modules (such as `ParisBranch`, `MadridBranch`, and `BerlinBranch`), a task doomed to produce vast naming clashes of symbols defined in the imported modules. To overcome the problem, a *qualified import* is to be used: Imported modules get local names that are further used to disambiguate the symbols. Thus, one can smoothly use knowledge inferred by different, imported rules with same heads.

The following example shows how Datalog can be extended with a qualified import. To have an overview over the status of EU-Rent vehicles, the notion of an old vehicle is defined differently for the different branches. The modules `ParisBranch`, `MunichBranch`, and `MadrdidBranch` get the local names `m1`, `m2`, and `m3`, respectively.

```
━━━━━━━━━━━━━━━━━━━━ CorporateStatistics ━━━━━━━━━━━━━━━━━━━━
import private CorporatePolicies
import ParisBranch @ m1
import MunichBranch @ m2
import MadridBranch @ m3
private old-vehicle(X) :- m1.vehicle(V), manufactured(V,Y), Y < 1990.
private old-vehicle(X) :- m2.vehicle(V), manufactured(V,Y), Y < 2000.
private old-vehicle(X) :- m3.vehicle(V), manufactured(V,Y), Y < 1995.
```

The simple module-based architecture and the given module examples show a couple of advantages of such a module-based approach. We have already touched on the *separation of concerns* when describing the modules of Paris branch as having each a well-defined purpose. Modules such as `EURegulations` and `FrenchRegulations` can be *reused* by other applications too (not only by new established EU-Rent branches, but also by other companies). A module-based implementation is much more flexible and less error-prone than one without modules; this eases considerably the *extensibility* of the implementation.

3   Framework for rule language module systems

In Section 2 we saw an example of how it is possible to modularize rule-sets—in this case for Datalog—and that it is important to ensure separation (or encapsulation) of the different modules. We choose to enforce this separation statically, i.e., at compile time, due to our desire to reuse—rather than extend—existing rule engines that do not have an understanding of modules. An added advantage is a clean and simple semantics based on concepts already familiar to the users of the supported rule language.

We are interested in extending what was done for Datalog in Section 2 to a general framework for rule languages. The notion of modules is arguably important not only for Datalog, but for any rule language lacking such an important concept.

However, our reduction semantics for module operators poses some requirements to the expressiveness of a rule language: To describe these requirements, we first introduce in the following a few notions and assumptions on rule languages that give us formal means to talk about a rule language in general. Second, we establish the single requirement we ask from a rule language to be amenable to our module framework: the provision of rule chaining (or rule dependency).

In the next section, we than use these notions to describe the single operator needed to formally define our approach to modules for rule languages. We show that, if the rule language supports (database) views, that single operator suffices to obtain a powerful, yet simple to understand and realize module semantics. Even in absence of (database) views, we can obtain the same result (from the perspective of the module system's semantics) by adding two additional operators.

Rule languages. For the purpose of this work, we can take a very abstract view of a rule language: A rule language is any language where

1. *programs* are finite sequences of rules of the language;
2. *rules* consist of one head and one body, where the body and the head are finite sequences of rule parts;
3. *rule parts*[7] are, for the purpose of this work, atomic. Body rule parts can be understood as a kind of condition and head rule parts as a kind of result, action, or other consequence of the rule.

Rule dependency. Surprisingly, we care very little about the actual shape of rule parts, let alone their semantics. The only critical requirement needed by our framework is that the rule language has a concept of *rule chaining* or *rule dependency*. That is, one rule may depend on another for proper processing.

Definition 1 (Rule dependency). *With a program P, a relation $\Delta \subset \mathbb{N}^2 \times \mathbb{N}^2$ can be associated such that $(r_1, b, r_2, h) \in \Delta$ iff the condition expressed by the b-th body part of rule r_1 is dependent on results of the h-th head part of the r_2-th rule in P (such as derived data, actions taken, state changes, or triggered events), i.e., it may be affected be that head part's evaluation. Intuitively, Δ partitions the space of rule parts into allowed and forbidden pairs.*

Controlling rule (or rule part) dependency can take different forms in different rule languages: in Datalog, predicate symbol provide one (easy) means to partition the dependency space; in XSLT, modes can serve a similar purpose; in Xcerpt, the labels of root terms; …However, the realisation of the dependency relation is left to the rule language. We simply assume its existence and that it can be manipulated arbitrarily, though we will never introduce cycles in the dependency relation if they do not already exist. All the above realisations of the dependency relation are, up to an isomorphisms on the symbols of the domain (which, for generic (in the sense of [1]) query and rule languages, has no affect on the program's semantics), equivalence preserving.

The module extension framework requires the ability to express an arbitrary (acyclic) dependency relation, however poses no restrictions on the shape of Δ for a module-free program. Indeed, for any module-free rule program P $\mathbb{N}^4$ forms a perfectly acceptable Δ relation on P.

We only require the ability to express *acyclic* dependency relations as the discussed module algebra does not allow cycles in the module composition for simplicity's sake.

---

[7] We refrain from calling rule parts literals, as they may be, e.g., entire formulas or other constructs such as actions that are not usually considered logical literals.

Most rule languages that allow some form of rule chaining, e.g., datalog, SWRL, SQL, Xcerpt, R_2G_2, easily fulfill this requirement. However, it precludes rule languages such as CSS where all rules operate on the same input and no rule chaining is possible. Interestingly, though CSS already provides its own module concept, that module concept provides no information hiding, the central aim of our approach: Rules from all imported modules are merged into one sequence of rules and all applied to the input data, only precedence but not applicability may be affected by the structuring in modules.

Reduction semantics. Why do we impose this requirement on a rule language to be amenable to our module framework? The reason is that we aim for a reduction semantics where all the additions introduced by the module framework are reduced to expressions of the original language. However, to achieve this we need a certain expressiveness and that is exactly what the dependency requirement ensures.

Consider, for instance, the Module "CorporateStatistics" in Section 2. Let's focus only on old-vehicle and the three vehicle predicates from the three local branches. Using the semantics defined in the following section, we obtain a program containing also all rules from the included modules plus a dependency relation that enforces that only certain body parts of old-vehicle depend on the vehicle definition from each of the local branches. This dependency relation can be *realized* in datalog, e.g., by properly rewriting the predicate symbols through prefixing predicates from each of the qualified modules with a unique prefix.

4   Module system algebra

Remember, that the main aim of this work is to allow a rule program to be divided into *conceptually independent collections* (modules) of rules with well-defined interfaces between these collections.

For this purpose, we introduce in Section 4.1 a formal notion of a collection of rules, called "module", and its public interface, i.e., that subset of rules that constitutes the (public) interface of the module. Building on this definition, we introduce an algebra (comprised of a single operator) for composing modules in Section 4.2 together with a reduction semantics, i.e., a means of reducing programs containing such operators to module-free programs.

Operators by example. Before we turn to the formal definitions, let's again consider the EU-Rent use case from Section 2, focusing on the three modules "CorporateStatistics", "CorporatePolicies", and "ParisBranch".

We can define all the import parts of these modules using the module algebra introduced in the following. We use $A \times B$ for indicating that a module B is imported into module A and A inherits the public interface of B (cf. `import public`), $A \bowtie B$ to indicate private import (cf. `import private`), and $A \rtimes_S B$ for scoped import (cf. `import ... @`) where S are pointers to all rule parts addressing a specific module.

Using these operators we can build formal module composition expressions corresponding to the surface syntax from Section 2 as follows:

$$\text{CorporatePolicies}' = (\text{CorporatePolicies} \times \text{InsurancePolicies}) \bowtie \text{EURegulations}$$

$$\text{CorporatePolicies}'_{french} = \text{CorporatePolicies}' \times \text{FrenchRegulations}$$

$$\text{ParisBranch}' = (((\text{ParisBranch} \bowtie \text{Vehicles}) \bowtie \text{Customers}) \bowtie \text{BranchPolicies})$$
$$\bowtie \text{CorporatePolicies}'_{french}$$

$$\text{CorporateStatistics}' = (((\text{CorporateStatistics} \bowtie \text{CorporatePolicies}') \rtimes_{(1,1)} \text{ParisBranch}')$$
$$\rtimes_{(2,1)} \text{MunichBranch}') \rtimes_{(3,1)} \text{MadridBranch}'$$

$$\text{MunichBranch}' = ...$$

Thus, given a set of basic modules, each import statement is translated into a module algebra expression that creates a new module, viz. the semantics of the import statement. Unsurprisingly, parameterized modules lead to multiple "versions" of similar module composition expressions that only differ in instantiations of the parameters.

4.1 Defining Modules

We use *module identifiers* as means to refer to modules, e.g., when importing modules. Some means of resolving module identifiers to modules (stored, e.g., in files or in a database) is assumed, but not further detailed here.

Definition 2 (Module). *A module M is a triple* $(R_{\text{PRIV}}, R_{\text{PUB}}, \Delta) \subset \mathscr{R} \times \mathscr{R} \times \mathbb{N}^4 \times \Sigma$ *where $\mathscr{R}$ is the set of all finite sequences over the set of permissible rules for a given rule language. We call R_{PRIV} the private, R_{PUB} the public rules of M, and Δ the dependency relation for M. For the purpose of numbering rules, we consider $R = R_{\text{PRIV}} \diamond R_{\text{PUB}}$[8] the sequence of all rules in M.*

We call a module's public rules its "interface": When importing this module, only these rules become accessible to the importing module. Though the module composition discussed in this section does not rely on any further information about a module, a module should be accompanied by additional information for its *users*: documentation about the purpose of the module, what to expect from the modules interface, what other modules it relies on, etc.

Note, that a standard program (without modules) is considered just a special case of a module where R_{PUB} is empty.

Figure 2 shows an exemplary configuration of a program together with two modules A and B. Where the program consists in a single sequence of (private) rules, the rules of each of the modules are partitioned into private and public rules. The allowed dependency relation Δ is represented in the following way: All body parts in each of the areas ▨, ▨, and ▨ are depending on all head parts in the same area and no other head parts. No access or import of modules takes place, thus no inter-module dependencies exists in Δ between rule parts from one of the modules with each other or with the (main) program.

---

[8] $\diamond$ denotes *sequence concatenation*, i.e., $s_1 \diamond s_2$ returns the sequence starting with the elements of s_1 followed by the elements of s_2, preserving the order of elements from s_1 and s_2 respectively.

Fig. 2 Program and two defined modules without imports
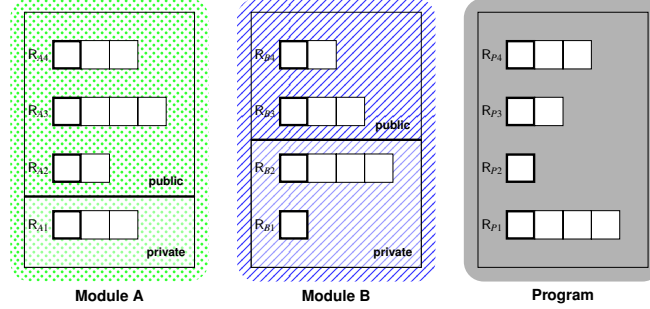
Notice, that for the dependency within a module the partitioning in private and public plays *no role whatsoever*. Body parts in private rules may access head parts from public rules and vice versa.

4.2   Module Composition

Module composition operators allow the (principled, i.e., via their public interface) definition of inter-module dependencies. Our module algebra (in contrast to previous approaches) needs only a single fundamental module composition operator. Further operators can then be constructed from a combination of that fundamental operator and (standard database) views. However, we also discuss an immediate definition of such further operators for languages where the view based approach is not applicable or desirable.

Scoped import. The fundamental module composition operator is called the *scoped import operator*. It name stems from the fact that it allows to specify not only which module to import but also which of the rules of the importing module or program may be affected by that import. In other words, it allows us to precisely control the scope of a module's import.

Informally, the scoped import uses two modules *A* and *B* and a set *S* of body parts from *A* that form the scope of the module import. It combines the rules from *B* with the rules from *A* and extends the dependency relation from all body parts in *S* to all public rules from *B*. No other dependencies between rules from *A* and *B* are established.

To illustrate this consider again the configuration from Figure 2. Assume that we import (1) module A into B with a scope limited to body part 3 of rule R_{B2} and that (2) we import the result into the main program limiting the scope to body parts 2 and 3 of rule R_{P1}. Third, we import module A also directly into the main program with scope body part 1 of R_{P1}.

This can be compactly expressed by the following module composition expression:

$$\left(P \bowtie_{(1,2),(1,3)} \left(B \bowtie_{(2,3)} A \right) \right) \bowtie_{(1,1)} A$$

As usual, such expressions are best read inside out: We import the result of importing A into B with scope $\{(2,3)\}$ into *P* with scope $\{(1,2),(1,3)\}$ and then also import

A with scope $\{(1,1)\}$. The result of this expression (in particular the dependency) is shown in Figure 3.

Fig. 3 Scoped import of (1) module A into body part 3 of rule R_{B2} and into body part 1 of rule R_{P1} and (2) of (the expanded) module B into body part 2 and 3 of rule R_{P1}. into the main program

---
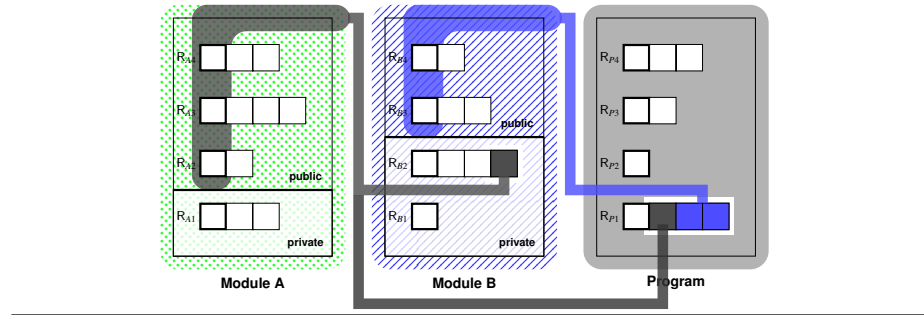
Formally, we first introduce the concept of (module) scope.

Definition 3 (Scope). *Let $M = (R_{\mathrm{PRIV}}, R_{\mathrm{PUB}}, \Delta)$ be a module (or program if R_{PUB} is the empty sequence. Then a set of body parts from M is called a scope in M. More precisely, a scope S in M is a set of pairs from $\mathbb{N}^2$ such that each pair identifies one rules and one body part within that rules.*

E.g., the scope $\{(1,2),(1,3),(4,2)\}$ comprises for program P from Figure 2 the body parts 2 and 3 of rule 1 and the body part 2 of rule 4.

Second, we need a notation for adjusting a given dependency relation when adding rules. It turns out, a single operation (slide) suffices for our purposes:

Definition 4 (Dependency slide). *Given a dependency relation Δ, slide computes a new dependency relation by sliding all rules in the slide window $W = [start + 1, start + length + 1]$ in such a way that the slide window afterwards starts at $start_{new} + 1$:*

$$slide(\Delta, start, length, start_{new}) = \{(r_1', b, r_2', h) : (r_1, b, r_2, h) \in \Delta$$

$$\wedge r_1' = \begin{cases} start_{new} + 1 + (r_1 - start) & \text{if } r_1 \in W \\ r_1 & \text{otherwise} \end{cases}$$

$$\wedge r_2' = \begin{cases} start_{new} + 1 + (r_2 - start) & \text{if } r_2 \in W \\ r_1 & \text{otherwise} \end{cases}$$

With this definition, a scoped import becomes a straightforward module composition:

Definition 5 (Scoped import $\bowtie$). *Let $M' = (R'_{\mathrm{PRIV}}, R'_{\mathrm{PUB}}, \Delta')$ and $M'' = (R''_{\mathrm{PRIV}}, R''_{\mathrm{PUB}}, \Delta'')$ be two modules and S a scope in M'. Then*

$M' \bowtie_S M'' := (R_{\mathrm{PRIV}} = R'_{\mathrm{PRIV}} \diamond R''_{\mathrm{PRIV}} \diamond R''_{\mathrm{PUB}}, R_{\mathrm{PUB}} = R'_{\mathrm{PUB}}, \Delta'_{slided} \cup \Delta''_{slided} \cup \Delta_{inter}),$ *where*

Fig. 4 Private import of A into B and B into the main program
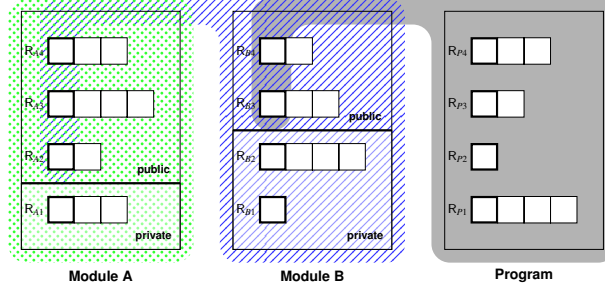
- $\Delta'_{slided} = \text{slide}(\Delta', |R'_{\text{PRIV}}|, |R'_{\text{PUB}}|, |R'_{\text{PRIV}}|)$ *is the dependency relation of the* importing *module* M' *with the* public *rules slided to the very end of the rule sequence of the new module (i.e., after the rules from* M''*),*
- $\Delta''_{slided} = \text{slide}(\Delta'', 1, |R''_{\text{PRIV}}| + |R''_{\text{PUB}}|, |R'_{\text{PRIV}}|)$ *is the dependency relation of the* imported *module* M' *with all its rules slided between the private and the public rules of the importing module (they have to "between" because they are part of the private rules of the new module),*
- $\Delta_{inter} = \{(r_1, b, r_2, h) : (r_1, b) \in S \land \exists$ *a rule in* R_{PRIV} *with index* r_1 *and head part* h : $r_1 > |R'_{\text{PRIV}}| + |R''_{\text{PRIV}}|\}$ *the inter-dependencies between rules from the importing and rules from the imported module. We simply allow each body part in the scope to depend on all the public rules of the imported module. This suffices for our purpose, but we could also choose a more precise dependency relation (e.g., by testing whether a body part can at all match with a head part).*

Note, that the main difficulty in this definition is the slightly tedious management of the dependency relation when the sequence of rules changes. We, nevertheless, chose an explicit sequence notation for rule programs to emphasize that this approach is entirely capable of handling languages where the order of rules affects the semantics or evaluation and thus should be preserved.

Public and private import. Two more import operators suffice to make our module algebra sufficiently expressive to formalize the module system discussed in Section 2 as well as module systems for languages such as XQuery or Xcerpt.

In fact, if the language provides a (database) view concept, the single scoped import operator suffices as the two remaining ones can be defined using views on top of scoped imports. Before introducing these rewritings, we briefly introduce the public and private import operator. Formal definitions are omitted for conciseness reasons, but can be fairly straightforwardly derived from the definition for the scoped import.

All three operators hide information resulting from private rules in a module, however the public information is made accessible in different ways by each of the operators: The scoped import operator makes the information from the public interface of *B* accessible only to explicitly marked rules. The private and public import operators, in contrast, make all information from the public interface of *B* accessible to all rules of *A*. They differ only w.r.t. cascading module import, i.e., a module *A* that imports another

modules B is itself imported. In that case the public import operator ($\times$) makes the public interface of B part of the public interface of A, whereas the private import operator ($\bowtie$) keeps the import of B hidden.

Figure 4 shows the effect of the private import operator on the configuration from Figure 2 using the module composition expression $\mathsf{P} \bowtie (\mathsf{B} \bowtie \mathsf{A})$: Module B imports module A privately and the main program imports module B privately. In both cases, the immediate effect is the same: The body parts of B get access to the head parts in A's public rules and the body parts of the main program get access to the head parts in B's public rules. The import of A into B is hidden entirely from the main program. This contrasts to the case of the public import in the expression $\mathsf{P} \times (\mathsf{B} \times \mathsf{A})$. There the main program's body parts also gain access to the head parts in A's (and not only B's) public rules.

Operator Rewriting. As stated above, we can express both public and private import using additional views (i.e., deductive rule) plus a scoped import, if the rule language provides view.

Corollary 6 (Rewriting $\times$). *Let $M' = (R'_{\text{PRIV}}, R'_{\text{PUB}}, \Delta'), M'' = (R''_{\text{PRIV}}, R''_{\text{PUB}}, \Delta'')$ be modules and $M = (R_{\text{PRIV}}, R_{\text{PUB}}, \Delta) = M' \times M''$. Then, $M^* = (R'_{\text{PRIV}}, R'_{\text{PUB}} \diamond R, \Delta') \rtimes_S M''$ is (up to the helper predicate R) equivalent to M if*

$$R = [h \colon\!\text{-}\ h : h \text{ is a head part in } R''_{\text{PUB}}]$$
$$S = \{(i,1) : |R'_{\text{PRIV}}| + |R'_{\text{PUB}}| < i \leq |private'| + |R'_{\text{PUB}}| + |R|\}.$$

The gist is the introduction of "bridging" rules that are dependent on no rule in the existing module but whose body parts are the scope of the import of M''.

Note, that we use one rule for each public head part of the imported module. If the language provides for body parts that match any head part (e.g., by allowing higher-order variables or treating predicate symbols like constant symbols like in RDF), this can be reduced to a single additional rule. In both cases, however, the rewriting is linear in the size of the original modules.

For private import, an analogous corollary holds (where the "bridge" rules are placed into the private rules of the module and the dependency relation properly adapted).

5   Related Work

Despite the apparent lack of modules in many Web rule languages, module extensions for logic programming and other rule languages have been considered for a long time in research. We believe, that one of the reasons that they are still not part of the "standard repertoire" of a rule language lies in the complexity of many previous approaches.

For space reasons, we can only highlight a select few approaches. First, in logic programming module extensions for Prolog and similar languages have fairly early been considered, cf., e.g., [4,14,6]. Miller [13] proposes a module extension for Prolog that includes paramterized modules similar in style to those as discussed in Section 2 and is the first to place a clear emphasize on strict (i.e., not to overcome) information

hiding. In contrast to our approach, the proposed semantics is an extension to standard logic programming (with implication in goals and rule bodies).

A reduction semantics, as used in this paper, is proposed in [11], though extra logical run-time support predicates are provided to allow module handling at run-time. However, the approach lacks support for module parameters and a clear semantics (most notable in the distinction between import and merge operation).

The most comprehensive treatment of modules in logic programming is presented in [5]. The proposed algebra is reminiscent of prior approaches [3] for first-order logic modules in algebraic specification formalisms (such as [15]). It shares a powerful expressiveness (far beyond our approach) and beautiful algebraic properties thanks to a full set of operators such as union, intersection, encapsulation, etc. The price, however, is that these approaches are far more complex. We believe, that a single well-designed union (or combination) operation together with a strong reliance on views as an established and well-understood mechanism in rule languages is not only easier to grasp but also easier to realize. E.g., intersection and renaming operations as proposed in [3] can be handled by our module algebra through a combination of scoped imports and views. More recently, modules have also been considered in the context of distributed evaluation [8] beyond the scope of our paper.

Modules are useful for structuring not only logic programming but also other rule languages such as rewriting languages or active rule languages [2]. The Venus Rule Language, e.g., [7], offers programmatic control over the control flow between modules by allowing modules to be explicitly called as part of an action.

6   Conclusion

This work started with the ambition to abstract from relatively similar module systems realizable for different rule languages. In accordance with this ambition we defined a framework with clearly identifiable requirements put on rule languages to be usable in the framework. An important point to make, and a great advantage with such a framework, is that not only current rule language lacking constructs for modular design can be addressed, but also other rule languages which are yet to be designed and developed.

For the framework to be viable in practice, the concrete module implementation for a specific rule language falls back to the semantics and tools of the underlying rule language. Thus, the underlying tools can be used unmodified. Other techniques exist that take a similar 'reductive' approach to program manipulation, for example the composition framework Reuseware [10,9]. Due to these similarities, it is possible to implement the re-writing mechanism assumed in the module framework using the approach and tools provided by the composition framework Reuseware. As such, the practical implementation needed for different rule languages to support modular development can rely on existing work.

We have already implemented module systems for several languages based on the ideas presented here, in particular for the logic programming-style query language Xcerpt, for a rule-based grammer language R2G2, and for a Datalog fragment as discussed here. In all cases, the presented module system proved expressive enough to

yield a useful yet easy to use language extension with minimal implementation effort. Application of the framework to further languages are under work.

Acknowledgments

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley Publishing Co., Boston, MA, USA, 1995.
2. E. Baralis, S. Ceri, and S. Paraboschi. Modularization techniques for active rules design. *ACM Transactions on Database Systems*, 21(1):1–29, 1996.
3. J. A. Bergstra, J. Heering, and P. Klint. Module algebra. *Journal of the ACM*, 37(2):335–372, 1990.
4. K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. In K. Clark and S. A. Tarnlund, editors, *Logic Programming*, Apic Studies in Data Processing. Academic Press, Inc., 1983.
5. A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Trans. Program. Lang. Syst.*, 16(4):1361–1398, 1994.
6. M. Codish, S. K. Debray, and R. Giacobazzi. Compositional analysis of modular logic programs. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 451–464, New York, NY, USA, 1993. ACM Press.
7. S. Correl and D. P. Miranker. On isolation, concurrency, and the venus rule language. In *Proc. Int'l. Conf. on Information and Knowledge Management (CIKM)*, pages 281–289, New York, NY, USA, 1995. ACM Press.
8. A. Giurca and D. Savulea. An algebra of logic programs with applications in distributed environments. In *Annales of Craiova University*, volume XXVIII of *Mathematics and Computer Science Series*, pages 147–159, 2001.
9. J. Henriksson, U. Aßmann, F. Heidenreich, J. Johannes, and S. Zschaler. How dark should a component black box be? The Reuseware Answer. *Proc. of the 12th International Workshop on Component-Oriented Programming (WCOP) co-located with 21st European Conf. on Object-Oriented Programming (ECOOP'07) (to appear)*, 2007.
10. J. Henriksson, J. Johannes, S. Zschaler, and U. Aßmann. Reuseware – adding modularity to your language of choice. *Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear)*, 2007.
11. I. Karali, E. Pelecanos, and C. Halatsis. A versatile module system for prolog mapped to flat prolog. In *Proc. ACM Symp. on Applied Computing (SAC)*, pages 578–585, New York, NY, USA, 1993. ACM Press.
12. M. Kifer, J. de Bruijn, H. Boley, and D. Fensel. A realistic architecture for the semantic web. In *RuleML*, pages 17–29, 2005.
13. D. Miller. A theory of modules for logic programming. In *Proc. IEEE Symp. on Logic Programming*, pages 106–114, 1986.
14. D. T. Sannella and L. A. Wallen. A calculus for the construction of modular prolog programs. *Journal of Logic Programming*, 12(1-2):147–177, 1992.
15. M. Wirsing. Structured algebraic specifications: A kernel language. *Theoretical Computer Science*, 42(2):123–244, 1986.