The use of renaming in composing general programs*

Antonio Brogi, Simone Contiero, Franco Turini
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy

Keywords: *Composition, reuse, transformation.*

1 Introduction and motivations

Most modern computing systems consist of large numbers of software components that interact with each other. Correspondingly, the process of software development is changing from writing large pieces of software in some programming language to writing small pieces of software in some (meta-)language to suitably "glue" existing components and put them at work together.

Our work aims at contributing to setting firm foundations for developing program composition methodologies. We investigate the issues of program composition in a setting where both the language of components and the meta-language for composing them have a formal mathematical semantics. More precisely, we choose logic-based programming for specifying the components and a set of meta-level composition operations on logic programs as the gluing (part of the) meta-language. In previous work [4, 5] we have shown the use of a basic set of meta-level operations on logic programs for mastering the complexity of designing complex systems. More precisely, we have considered a modular approach to software development, where a complex system is partitioned into a number of small pieces that are designed to be combined together via composition operations.

In this paper, we analyse the role of *renaming* as a key ingredient of component-based programming. The capability of renaming symbols is indeed of primary importance in a program composition setting. On the one hand, independently developed components may employ different vocabularies of symbols. Different components may use different names for the same procedure, or may instead denote different procedures by the same name. These problems typically arise when building complex systems by combining large numbers of pre-existing components. In these situations, the capability of renaming symbols to eliminate vocabulary differences is a powerful support for the reuse of existing software components. On the other hand, renaming can be also employed to implement forms of information hiding and import/export between modules. Intuitively speaking, (part of) the code of a component may be made hidden to other components by renaming its symbols into fresh symbols not occurring in any other component.

To analyse the importance of renaming for component-based programming, we will consider renaming as a basic meta-level operation on components, and formally define its meaning as a program transformation operation. More precisely, we will introduce the renaming operator in the context of the family of composition operations presented in [4], and we shall focus our discussion on some examples to illustrate the expressive power of renaming.

The computational interpretation of renaming is defined via an equivalence preserving syntactic transformation of general program expressions into standard general programs, which can be then executed by any state-of-the-art interpreter for logic programs.

*Extended Abstracts of LOPSTR'98, Eighth International Workshop on Logic-based Program Synthesis and Transformation, 15-19 June 1998, Manchester, UK. Technical Report Series, Department of Computer Science, University of Manchester, ISSN 1361-6161. Report number UMCS-98-6-1. http://www.cs.man.ac.uk/cstechrep/titles98.html

2 General program expressions

We consider four basic meta-level operations for composing general[1] programs: *union* ($\cup$), *intersection* ($\cap$), *restriction* ($\prec$), and *renaming* (RN). Starting from a collection of named general programs, these operations define the following class of *general program expressions*:

$$E \ ::= \ P \ | \ E \cup E \ | \ E \cap E \ | \ E \prec E \ | \ RN(\rho, E)$$

where P is (the name of) a general program. Syntactically, both operands of $\cup$, $\cap$ and $\prec$ are program expressions. In a renamed expression $RN(\rho, E)$ the first operand is instead a *renaming substitution.*

A renaming substitution is defined as a set of pairs $\{p_1'/p_1, ..., p_h'/p_h\}$ where $\{p_1, ..., p_h\}$ and $\{p_1', ..., p_h'\}$ are disjoint sets of predicate symbols. Moreover: $\forall i, j \in [1, h] : (i \neq j \implies p_i \neq p_j)$. Informally speaking, the effect of renaming a program expression E via a renaming substitution $\rho = \{p_1'/p_1, ..., p_h'/p_h\}$ is that every occurrence of a predicate symbol p_i in E is replaced by p_i' in the renamed program expression $RN(\rho, E)$. We will denote by $A\rho$ the application of a renaming substitution ρ to an atom A. Similarly, we will denote by $I\rho$ and $P\rho$ the application of ρ to a set of atoms I and to a program P. Moreover, we will sometimes abuse notation and denote a renaming substitution ρ simply by $\{\pi'/\pi\}$, where π', π are sets of predicate symbols.
Finally, a renaming substitution $\rho = \{\pi'/\pi\}$ is applicable to a program expression E if the set of predicate symbols π' is disjoint from the set of predicate symbols occurring in E[2].

Any composition of general programs can be transformed into an equivalent general program. Namely, we define a syntactic transformation τ that maps any general program expression E into a general program $\tau(E)$ which is equivalent to E.

$$\begin{array}{lll}
\tau(P) & = P & \text{if } P \text{ is a general logic program} \\
\tau(E \cup F) & = \{A \leftarrow \overline{L} \ | \ A \leftarrow \overline{L} \in \tau(E) \ \vee \ A \leftarrow \overline{L} \in \tau(F)\} & \\
\tau(E \cap F) & = \{A \leftarrow \overline{L} \ | \ \exists A', A'', \overline{L'}, \overline{L''}, \vartheta : \ A' \leftarrow \overline{L'} \in \tau(E) \ \wedge \ A'' \leftarrow \overline{L''} \in \tau(F) \ \wedge & \\
 & \qquad \vartheta = mgu(A', A'') \ \wedge \ A = A'\vartheta \ \wedge (\overline{L} = \overline{L'}\vartheta \cup \overline{L''}\vartheta)\} & \\
\tau(E \prec F) & = \{A \leftarrow \overline{L} \ | \ A \leftarrow \overline{L} \in \tau(E) \ \wedge \ name(A) \notin defs(\tau(F))\} & \\
\tau(RN(\rho, E)) & = \tau(E)\rho &
\end{array}$$

where $name(A)$ denotes the predicate symbol of atom A, and $defs(P)$ denotes the set of predicate symbols defined in a program P.

The transformation τ merely depends on the syntactic structure of the given expression, namely on the composition operations and on the syntax of the programs forming it. When the program expression is already a general program then τ is just the identity function. The *union* of two programs corresponds to the set-theoretic union of the clauses of the separate programs. The *intersection* operation combines two programs by merging pairs of clauses with unifiable heads into clauses having the (duplicate-free) conjunctions of the bodies of the original clauses as a body. The net effect is that the two programs act as sets of constraints one upon the other. Consider for instance the programs:

P:	Q:
Likes(x,y) <- ¬ Bitter(y).	Likes(Bob,y) <- Sour(y).
Hates(x,y) <- Sour(y).	

[1] We shall use the standard terminology and notation of logic programming [1]. A literal is an atom or its negation, and a (general) query is a finite conjunction of literals. A *general clause* is a clause of the form $A \leftarrow \overline{L}$ where A is an atom and $\overline{L}$ is a query. A *general program* is a finite set of general clauses. Following [2], we consider a universal language in which all programs and queries are written, and we denote by $ground(P)$ the fully instantiated version of program P.

[2] The set of predicate symbols occurring in a program expression E is defined as the set of predicate symbols occurring in the program $\tau(E)$, obtained by applying the transformation τ (see below) to E.

Then the program $\tau(P \cap Q)$ = { Likes(Bob,y) $\leftarrow \neg$ Bitter(y), Sour(y). } consists of just one clause obtained by joining the first clause of P with the clause of Q. Notice that $\tau(P \cap Q)$ does not contain a definition for predicate Hates, since the latter is not defined in Q.

The *restriction* operation filters the set of clauses of a program by eliminating all the clauses defining predicates which are defined in another program. Namely, $\tau(P \prec Q)$ contains all the clauses of P which define predicates for which no definition is present in Q. In the previous example we have that $\tau(P \prec Q)$ = { Hates(x,y) <- Sour(y). }.

As far as renaming is concerned, the transformation of an expression $RN(\rho, E)$ is simply obtained by applying the renaming substitution ρ (admissible for E) to the general program $\tau(E)$. For instance: τ(RN({Loves/Likes}, (P $\cap$ Q))) ={ Loves(Bob,y) <- $\neg$ Bitter(y), Sour(y). }

3 Reusing components

The ability of renaming symbols is of primary importance when considering the problem of combining independently developed components.

Consider for instance a simple program Graph for determining, via standard transitive closure, the existence of a path between two nodes in an undirected graph. Namely Graph defines a relation Path in terms of an Arc relation over the nodes of a graph. Program Graph can be then composed with a program containing the representation of an actual graph. For instance, we may wish to use Graph with an existing database Trains containing information on train connections between cities:

```
Graph:
Path(x,y) <-  Arc(x,y).
Path(x,z) <-  Arc(x,y), Path(y,z).
```

```
Trains:
Train_connection(Florence, Milan).
Train_connection(Pisa, Florence).
...
```

The renaming operator can be then exploited for combining Graph with Trains so as to find the existence of train connections between cities. The example — which is over-simplified here because of space limits — is however representative of the general situation of a program defining some property (Path in this case) over some generic relation (Arc). Such a composition can be obtained by applying renaming predicate Arc into Train_connection by means of the expression:

$$\mathsf{RN}(\{\mathsf{Train\_connection/Arc}\}, \mathsf{Graph}) \ \cup \ \mathsf{Trains}.$$

Notice that, in this example, the reuse supported by renaming can also be achieved by introducing "bridge clauses" to link separate components. Namely we may simply consider the expression Graph $\cup$ Trains $\cup$ { Arc(x,y) <- Train_connection(x,y). } . It is worth observing, however, that renaming and bridge clauses are essentially different. Intuitively speaking, this is due to the fact that renaming *changes* the meaning of a program, while bridge clauses only *extend* the meaning of a program.

Renaming can be exploited together with other composition operations to define more structured compositions. For instance, consider a database Holidays containing general rules for choosing a country for the next holidays. Predicate Good_Place(c,s) is intended to indicate that country c is a good place for holidays during season s.

```
Holidays:
Good_Place(c,s)      <-  ¬ Dangerous(c,s), Attractive(c,s).
Dangerous(c,s)       <-  Raining_Season(c,s).
Dangerous(c,_)       <-  Civil_War(c).
Attractive(c,s)      <-  Hot_Season(c,s), Nat_Beauty(c).
Attractive(c,_)      <-  Art_Country(c).
Civil_War(Algeria).
Art_Country(Italy).
Nat_Beauty(Brasil).
...
```

Imagine also to have a knowledge base Prices containing information on the cost of living in foreign countries, and a set Prefs of rules specifying additional information on possible preferences.

```
   Prices:                           Prefs:
Rich_Country(Sweden).             Medium(x,s)    <- ¬ Rich_Country(x), Med_Season(x,s).
Med_Country(Spain).               Exclusive(x,s) <- Rich_Country(x), High_Season(x,s).
Poor_Country(Brasil).             Cheap(x,s)     <- Poor_Country(x), ¬ High_Season(x,s).
...                               ...
```

To constrain the search on the basis of a small budget, we may employ the expression:

$$\mathsf{Cheap\_Holiday} = (\mathsf{RN}(\{\mathsf{Good\_Place/Cheap}\}, \mathsf{Prefs})\ \mathsf{constrains}\ \mathsf{Holidays})\ \cup\ \mathsf{Prices}$$

where A $\mathsf{constrains}$ $\mathsf{DB} = (\mathsf{A} \cap \mathsf{DB}) \cup (\mathsf{DB} \prec \mathsf{A})$. Namely the predicates renamed in Prefs are used to constrain the general rules in Holidays defined by means of predicate Good_Place. The obtained expression also keeps all other rules from Holidays as well as the rules in Prices. Notice that other choice criteria can be expressed by different compositions of the same programs, for instance by changing the renaming substitution into {Good_Place/Exclusive}.

4 Information hiding

In the setting described so far, all symbols and predicate definitions of a component are by default visible to other components. There are however situations in which we would like to hide (part of) the code of a component to the other components.

Let us first consider a simple form of hiding that can be directly implemented by means of the *renaming* operator. Consider for instance an over-simplified database of a journal containing information on papers submitted for publication:

```
   Journal_DB:
Status (paper, Accepted)      <-    Accepted_Papers(ps), Member(paper,ps).
Status(paper, UnderReview)    <-    ¬ Status(paper, Accepted), Reviewers(paper, ref1, ref2),
                                    Missing_Review(paper, ref1, ref2).
Reviewers(P23-97, Brown, Smith).
...
```

In order to allow external users to query the database on the status of a paper, the database manager would reasonably need to hide some of the information present in the database, like for instance the names of the referees that are reviewing a paper. Intuitively speaking, such a simple form of hiding can be implemented by renaming part of the symbols in the database with fresh system symbols that cannot occur in any user program. For instance, the name of the relation Reviewers can be replaced by a fresh system predicate name, thus making Reviewers not visible to other components.

A more general form of information hiding is motivated by the abstract data type view of components. Consider for instance the programs:

```
   Palindrome:                        Lists:
Pal(l) <-  Reverse(l,r), l=r.      Reverse([ ],[ ].)
                                   Reverse([x|xs],ys) <-  Reverse(xs,zs), Append(zs,[x],ys).
                                   Append([ ],ys,ys).
                                   Append([x|xs],ys,[x|ws]) <-  Append(xs,ys,ws).
```

where program Palindrome defines a predicate `Pal` for determining whether a given sequence is palindrome. The definition of predicate Pal makes use of the Reverse operation on lists, which is defined in Lists. Following an abstract data type view, only the names of the operations on lists should be made visible to other modules, while their implementation should be hidden.

We now show how renaming can be exploited in a disciplined way to implement various forms of information hiding. Let us first consider an operation $Export(\pi_n, E)$, whose intended meaning is to leave only the names of predicates in π_n visible to other modules, while hiding to other modules the code of all predicates in E.

Such a form of information hiding can be implemented by means of the *renaming* and *union* operations. Namely, the expression $Export(\pi_n, E)$ can be defined as the *union* of two expressions:

(1) the expression E suitably renamed so as to replace each predicate symbol p in E with a fresh predicate symbol p' (not visible to the other components), and

(2) a set of bridge clauses of the form $p(x) \leftarrow p'(x)$ linking each predicate p in π_n with the corresponding fresh system predicate p'.

Syntactically, if π is the set of predicate symbols occurring [3] in E and π' is a set of *fresh* predicate symbols in a one-to-one correspondence with the set π, then $Export(\pi_n, E)$ can be defined as:

$$Export(\pi_n, E) \;=\; RN(\pi'/\pi, E) \;\cup\; \{p(x) \leftarrow p'(x) \mid \; p \in \pi_n \wedge (p'/p) \in \pi'/\pi\}.$$

Namely, predicate names in E are renamed with fresh system predicate names, which do not occur in any other user program[4], so that the entire code of E becomes not accessible to other components. Bridge clauses re-introduce a link to the names of predicates in π_n so that these names become again visible to other components.

For instance, the operation $Export(\pi_n, E)$ can be used to hide the entire code of an expression E while leaving all the names in E visible to other modules. For instance, in the previous example, in the expression:

$$\mathsf{Palindrome} \cup Export(\{\mathsf{Reverse}, \mathsf{Append}\}, \{\}, \mathsf{Lists})$$

only the names of predicates in Lists are visible to Palindrome, while the entire code of Lists is hidden. The operation $Export(\pi_n, E)$ can be also used to hide the code of an expression E while leaving only *some* of the procedure names in E visible to other modules. For instance:

$$\mathsf{Palindrome} \cup Export(\{\mathsf{Reverse}\}, \mathsf{Lists})$$

combines programs Palindrome and Lists so that the former can only use the name of the operation Reverse defined in the latter. On the other hand, the code of Reverse is hidden to Palindrome, while Append is fully hidden (both the name and the code) to Palindrome. It is worth noting that this usage of renaming supports the reconstruction of the form of exportation present in conventional modular programming languages.

In logic-based programming, procedure (viz. predicate) definitions consist of sets of clauses which may be distributed over different components, in contrast with the monolithical structure of procedure definitions in conventional languages. Moreover in logic-based programming the knowledge about a procedure is available at two different levels. The *intensional* definition of a procedure is the code of the predicate, i.e. the set of clauses defining it. The *extensional* definition is instead the set of atomic formulae provable for that predicate. Following this observation, the use of renaming in a logic program composition setting may hence support different forms of information hiding.

Let us now consider the more general situation in which we wish: (a) to hide the code and leave visible the name of some predicates, (b) to leave visible both the name and the code of other predicates, and finally (c) to hide both the code and the name of the remaining predicates.

[3] The predicate symbols occurring in an expression E actually are all predicate symbols occurring in the program $\tau(E)$.

[4] The choice of fresh system predicates must be defined in such a way that once a symbol has been chosen it will never be chosen again. This can be easily implemented by means of a system function able to generate on demand a stream of different predicate names.

To this end, let us consider a more general definition of the *Export* operation of the form $Export^{+}(\pi_n, \pi_c, E)$, whose intended meaning is to leave the names of predicates in π_n visible while hiding their code, and to leave visible both the name and the code of predicates in π_c. All other predicates in E but not in $(\pi_n \cup \pi_c)$ are fully hidden. Syntactically, if π is the set of predicate symbols occurring in E and π' is a set of *fresh* predicate symbols in a one-to-one correspondence with the set $(\pi - \pi_c)$, then $Export^{+}(\pi_n, \pi_c, E)$ can be defined as follows:

$$Export^{+}(\pi_n, \pi_c, E) = RN(\pi'/(\pi - \pi_c), E) \ \cup \ \{p(x) \leftarrow p'(x) \mid \ p \in \pi_n \wedge (p'/p) \in \pi'/\pi\}.$$

Namely the effect of $RN(\pi'/(\pi - \pi_c), E)$ is to make the code and the names of all predicates in E except those in π_c not accessible to other components, while the bridge clauses re-introduce a link to the names of predicates in π_n so that these names become again visible to other components. Notice that $Export^{+}$ properly generalises the definition of *Export* as $Export(\pi_n, E) = Export^{+}(\pi_n, \{\}, E)$.

5 Semantics

We have tried to illustrate some significant possible uses of a renaming operator for component-based programming, in the context of a family of meta-level composition operations over logic programs. The computational interpretation of program expressions has been defined in Sect. 2 by means of the syntactic transformation τ that maps arbitrary program expressions back into general programs, which can be then executed by any state-of-art interpreter for logic programs.

It is however important to define an abstract semantics of program compositions, besides this transformational semantics, so as to characterise the intended meaning of each composition without the need of referring to the corresponding transformed program. Brogi and Turini showed [6] the need of resorting to higher-order semantics in order to define compositional semantics for definite programs with non-trivial sets of composition operations. The same consideration applies to compositions of *general* programs, where in addition the orthogonal aspects of compositionality and non-monotonicity must be reconciled.

A compositional semantics for general program expressions containing $\cup$, $\cap$ and $\prec$ operations has been defined in [4] in terms of three-valued logic. Fitting [8] proposed to use a three-valued logic [11] to formalise the meaning of general programs. Three-valued logic models the idea that a query can yield *three* possible outcomes: success, failure, and divergence. Fitting [8] also defined a three-valued immediate consequence operator Φ, which given a general program P and a three-valued interpretation[5] I yields a three-valued interpretation. Namely:

$$\begin{array}{rcll} \Phi(P)(I) & = & (T, F) \ \text{where} & T = \{A \mid \ \exists\, \overline{L}: \ (A \leftarrow \overline{L}) \in ground(P) \text{ and } \overline{L} \text{ is true in } I\ \} \\ & & \text{and} & F = \{A \mid \ \forall\, \overline{L}: \ (A \leftarrow \overline{L}) \in ground(P) \ \text{ implies } \ \overline{L} \text{ is false in } I\ \}. \end{array}$$

A compositional semantics for program expressions has been defined in [4] by inductively extending the definition of Φ so that its first argument is an arbitrary general program expression. For instance, for any pair E_1, E_2 of program expressions and for any three-valued Herbrand interpretation I, we put:

$$\begin{array}{rcl} \Phi(E_1 \cup E_2)(I) & = & (\Phi^{+}(E_1)(I) \cup \Phi^{+}(E_2)(I), \ \ \Phi^{-}(E_1)(I) \cap \Phi^{-}(E_2)(I)) \\ \Phi(E_1 \cap E_2)(I) & = & (\Phi^{+}(E_1)(I) \cap \Phi^{+}(E_2)(I), \ \ \Phi^{-}(E_1)(I) \cup \Phi^{-}(E_2)(I)) \end{array}$$

where $\Phi^{+}(E)(I)$ (resp., $\Phi^{-}(E)(I)$) denotes the positive (resp. negative) part of the three-valued interpretation $\Phi(E)(I)$. Namely, the atoms assumed to be true for $E_1 \cup E_2$ and I are the atoms assumed to be true for E_1 and I *or* for E_2 and I. In a dual manner, the atoms assumed to be

[5]A *three-valued Herbrand interpretation* is a pair $I = (I^{+}, I^{-})$ of Herbrand interpretations, where I^{+} are atoms assumed to be true, and I^{-} are atoms assumed to be false. All other atoms are assumed to be unknown.

false for the $E_1 \cup E_2$ and I are the atoms which are assumed to be false for E_1 and I *and* for E_2 and I. The duality of the *union* and *intersection* operations is reflected by the corresponding definition of the Φ operator.

It is worth observing that the above semantics extends to deal with the *renaming* operator introduced in this paper. To this aim, if $\rho = \{\pi'/\pi\}$ is a renaming substitution and I is a Herbrand interpretation, we use the notation:

$$\mathcal{R}(\rho, I) = \{p(t) \mid \; p'(t) \in I \;\wedge\; p'/p \in \rho\} \;\cup\; (I - \{p(t) \mid p \in (\pi \cup \pi')\}).$$

The notation trivially extends to three-valued (Herbrand) interpretations, that is, if $I = (I^+, I^-)$ then $\mathcal{R}(\rho, I) = (\mathcal{R}(\rho, I^+), \mathcal{R}(\rho, I^-))$. The definition of the immediate consequence operator Φ can be then extended to deal with renamed program expressions as follows. For each expression E, three-valued interpretation I, and admissible renaming substitution ρ, we put:

$$\Phi(RN(\rho, E))(I) \;=\; (J^+\rho \;,\; \{p(t) \mid p \in \pi\} \;\cup\; (J^- - \{p'(t) \mid p' \in \pi'\})\rho \;\;)$$

where $\Phi(E)(\mathcal{R}(\rho, I)) = J$. Namely the immediate consequences of a renamed program expression $RN(\rho, E)$ w.r.t. a three-valued interpretation I are obtained from the immediate consequences of the original expression E and the interpretation $\mathcal{R}(\rho, I)$. The positive consequences simply correspond to the renaming of J^+ via ρ. The negative consequences instead are obtained from the negative consequences J^- (suitably filtered of all the renamed atoms) renamed via ρ and augmented with all the atoms whose predicate symbol belongs to π.

For example, the program $P = \{a \texttt{ <- } b \;,\; \neg c.\}$ the renaming substitution $\rho = \{a'/a,\; b'/b\}$, and the interpretation $I = (\{b'\}, \{c\})$, give rise to the interpretation $\mathcal{R}(\rho, I) = (\{b\}, \{c\})$ and the consequences:

$\Phi(RN(\rho, P))(I) \;=\; ((\Phi^+(P)(\mathcal{R}(\rho, I)))\rho \;,\; \{a, b\} \cup (\Phi^-(P)(\mathcal{R}(\rho, I)) - \{a', b'\})\rho) \;=$
$(\Phi^+(P)(\{b\}, \{c\})\rho \;,\; \{a, b\} \cup (\Phi^-(P)(\{b\}, \{c\}) - \{a', b'\})\rho) \;=$
$(\{a\}\rho \;,\; \{a, b\} \cup (\{a', b', b, c\} - \{a', b'\})\rho) \;=\; (\{a'\} \;, \{a, b, b', c\}).$

It is worth observing that the transformation τ introduced in Sect. 2 preserves the three-valued semantics defined by Φ. More precisely, each program expression E is Φ-equivalent to the transformed general program $\tau(E)$, as stated by the following proposition.

Proposition [4] *Let E be a general program expression. Then for any three-valued Herbrand interpretation I:* $\quad \Phi(E)(I) \;=\; \Phi(\tau(E))(I).$

It is possible to show that the above proposition extends to the case in which general program expressions may include the *renaming* operator. By virtue of the above equivalence result, the three-valued semantics of general program expressions can be then related [4] to various operational semantics for general programs proposed in the literature.

6 Concluding remarks

The concept of renaming has been widely studied and employed in mathematics and computer science for different purposes, including program composition. Adopting an algebraic approach for defining the semantics of module and module compositions, Bergstra et al. [3] introduced a renaming operator for combining different signatures. However, the kind of renaming they proposed is "permutative", that is, if a is replaced by b, then b is replaced by a. This can be useful for avoiding name clashes, but not for purposes such as increasing or decreasing the cooperation between different parts of code, as we intend to use it.

To the best of our knowledge, only a few papers have been devoted to discuss the problems of composing general logic programs.

Lifschitz and Turner [12] investigated the idea of *splitting* a logic program into parts in the context of the answer set semantics [9]. The basic idea is that, in many cases, a program can be divided into a "bottom" part and a "top" part, such that the latter does not refer to predicates defined in the former. Lifschitz and Turner showed that computing the answer sets for a program can be simplified when the program is split into parts. They also showed that the idea of splitting can be applied for proving properties of simple program compositions, like a conservative extension property for a program P extended by rules whose heads do not occur in P. Etalle and Teusink [7] and Verbaeten et al. [13] studied the problems of defining compositional semantics for general logic programs. Etalle and Teusink [7] define a compositional semantics for general programs by means of a first-order unfolding operator. Such a semantics is applied for composing "open" general programs by considering the union of programs as the only composition operation. Verbaeten et al. [13] present several results on the compositionality of general logic programs by generalising the well-founded semantics of logic programming. More precisely, they identify some conditions under which the class of extended well-founded models of the union of two general programs coincides with the intersection of the classes of extended well-founded models of the two programs. All these approaches differ from ours since the union of programs is the only composition operation considered, and since a restrictive naming policy is imposed on the predicate names of the programs to be composed. Namely, in contrast with our naming policy, predicate definitions cannot be spread over different programs.

The use of parameterisation proposed by Hill [10] for typed logic-based languages shares some of the objectives of our work. Hill considered two kinds of parameters: "open" parameters to support code linking, and "renaming" parameters to support code reuse. As we have shown, both these mechanisms can be expressed in our setting, and the main differences between our work vs. [10] are to consider a family of composition operations vs. only one module composition operation, and the use of untyped vs. typed programs.

From an implementation viewpoint, the transformation τ for the renaming operation has been developed in `SICStus` Prolog. This actually enriches, in a practical applicability perspective, the toolkit for composing general logic programs described in [4].

References

[1] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 493–574. Elsevier, 1990. Vol. B.

[2] K. R. Apt and R. Bol. Logic Programming and Negation: A Survey. *Journal of Logic Programming*, 19-20:9–71, 1994.

[3] J. A. Bergstra, J. Heering, and P. Klint. Module Algebra. *Journal of the ACM*, 37(2):335–372, 1990.

[4] A. Brogi, S. Contiero, and F. Turini. Composing General Logic Programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of LPNMR'97*, number 1265 in LNAI, pages 273–288. Springer Verlag, 1997.

[5] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1361–1398, 1994.

[6] A. Brogi and F. Turini. Fully abstract compositional semantics for an algebra of logic programs. *Theoretical Computer Science*, 149(2):201–229, 1995.

[7] S. Etalle and F. Teusink. A Compositional Semantics for Normal Open Programs. In M. Maher, editor, *Proc. of JICSLP 96*, pages 468–482, 1996.

[8] M. Fitting. A Kriple-Kleene semantics for general logic programs. *Journal of Logic Programming*, 4:295–312, 1985.

[9] M. Gelfond and V. Lifschitz. Classical negation in logic programming and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[10] P. Hill. A Module System for Systematic Software Development: Design and Implementation. In A. Brogi and P. Hill, editors, *Proc. of LOCOS97: Workshop on Logic Based Composition of Software*, Leuven, 1997.

[11] S.C. Kleene. *Introduction to Metamathematics.* van Nostrand, New York, 1952.

[12] V. Lifschitz and H. Turner. Splitting a logic program. In Pascal Van Entenryck, editor, *Proc. 11th International Conference on Logic Programming*, pages 23–37. The MIT Press, 1994.

[13] S. Verbaeten, M. Denecker, and D. De Schreye. Compositionality of normal logic programs. In J. Maluszynski, editor, *Logic Programming, Proceedings of the 1997 International Symposium*, pages 371–395, October 1997.