Under consideration for publication in Theory and Practice of Logic Programming

An Interactive Semantics of Logic Programming

ROBERTO BRUNI, UGO MONTANARI

Dipartimento di Informatica, Università di Pisa,
Corso Italia 40, 56125 Pisa, Italia.
{bruni,ugo}@di.unipi.it

FRANCESCA ROSSI

Dipartimento di Matematica, Università di Padova,
Via Belzoni 7, 35131 Padova, Italia.
frossi@math.unipd.it

Abstract

We apply to *logic programming* some recently emerging ideas from the field of reduction-based communicating systems, with the aim of giving evidence of the hidden interactions and the coordination mechanisms that rule the operational machinery of such a programming paradigm. The semantic framework we have chosen for presenting our results is *tile logic*, which has the advantage of allowing a uniform treatment of goals and observations and of applying abstract categorical tools for proving the results. As main contributions, we mention the finitary presentation of abstract unification, and a concurrent and coordinated abstract semantics consistent with the most common semantics of logic programming. Moreover, the compositionality of the tile semantics is guaranteed by standard results, as it reduces to check that the tile systems associated to logic programs enjoy the *tile decomposition property*.

Introduction

Logic programming (Lloyd, 1987) is a foundational research field that has been extensively investigated throughout the last 25 years. It can be said that, in logic programming, theory and practice meet together since its very beginning, as each innovation on one side contributes many insights to the other side thanks to the basic principle of logic programming, which is 'writing programs by expressing their properties.'

This symbiosis has also facilitated the study and the prototyping of interdisciplinary applications that either extend the 'kernel' of the framework with additional features or transfer helpful techniques from a large variety of paradigms. A typical example is the embedding of constraints within logic programming (Marriott & Stuckey, 1998), which retains the declarative and clean semantics of logic programming, as well as its typical problem solving features, while extending its applicability to many practical domains.

More interestingly, very often these flows of ideas have been profitably bidirectional and continuous, thus allowing one to establish strong connections between different areas (bringing useful analogies) and also to bridge gaps between different formalisms.

Interaction via contextualization and instantiation

In this paper, inspired by recent progress in the fields of communicating systems and calculi for concurrency, we want to focus on an *interactive* view of logic programming. The idea is to understand logical predicates as (possibly open) interacting agents whose local evolutions are coordinated by the unification engine. In fact, the amount of interaction arises from the unification mechanism of resolution, as subgoals can share variables and therefore 'local' progress of a component can influence other components by further instantiating such shared variables. One central aspect of this view is to understand what kind of information we should observe to characterize interaction and how far the approach can be extended to deal with different semantic interpretations of logic programs. For example, one interesting issue is *compositionality*. Having a compositional semantic framework is indeed very convenient for formal reasoning on program properties and can facilitate the development of modular programs (Bossi *et al.*, 1994; Brogi *et al.*, 1992; Gaifman & Shapiro, 1989; Mancarella & Pedreschi, 1987).

We sketch here the main ideas concerning the role played by 'contexts' in reduction systems, but for a more precise overview we invite the interested reader to join us in the little detour, from the logic programming world to the process description calculi area, inserted in the last part of this introductory section (with links to related literature).

Generally speaking, the issue we focus on is that of equipping a reduction system with an interactive semantics. In fact, although reduction semantics are often very convenient because of a friendly presentation, they are not compositional 'in principle.' The problem is that they are designed having in mind a progressive reduction of the initial state to a suitable normal form, i.e., one focuses on a completely specified system that can be studied in isolation from all the rest. In logic programming, this would correspond to study the refutation of ground goals only and to develop an *ad-hoc* system to this aim. Then, if one wants to study the semantics of partially specified components the framework is no longer adequate and some extensions become necessary. For example, in process description calculi, a partially specified component can be a process term (called *open process* or *context*) that contains suitable process variables representing generic subprocesses. However, also a closed term (i.e., without free process variables) can be considered an open system when it evolves as part of a broader system, by interacting with the environment. In logic programming we can distinguish two main kinds of openness and interaction. A first kind is due to goals with variables (rather than ground) that can obviously be regarded as partially specified systems. A second kind consists of regarding an atomic goal as part of a larger conjoined goal with which it must interact.

The obvious way to deal with partially specified components is to transform the problem into the reduction case, which we know how to solve. This means that (1) the variables of open processes will be instantiated in all possible ways to obtain closed systems that can be studied; (2) in order to study the semantics of closed subprocesses we will insert them in all possible contexts and then study their reductions. Moreover, the operations of contextual and instantiation closure can be rendered dynamically, provided that one defines a *labeled transition system* (LTS) whose labels record the information on the performed closure, and this has originated the idea of observing contexts and instantiations (sometimes called *external* and *internal* contexts, respectively).

Even if these views can look semantically adequate, it can be noted, as their main drawback, that they are not applicable in practice, because all considered closures are infinitary. The situation can be improved if one is able to identify a small finite set of contexts and/or instances that contains all useful information, since this can make the approach operationally satisfactory. While a general methodology for accomplishing this task in communicating and mobile calculi is difficult to find (see e.g., (Leifer & Milner, 2000)), we think that logic programming represents the perfect example where to develop this approach.

When dealing with the interactive view of logic programs, the idea is that unification is the basic action taking place during computation, and therefore the observed information must rely on such an action. We have seen that two kinds of closure can be distinguished that are dual to each other, namely *contextualization* and *instantiation*. The former can be used to embed components in a larger environment, while the latter serves to specialize an open system to some particular instance.

We shall concentrate our efforts on *pure* logic programming (i.e., classical Horn clauses, without any additional 'gadgets'). Hence contextualization corresponds to put the goal in conjunction with other goals,[1] i.e., given a goal G we should put it in the context $\_ \wedge G'$ for all possible G'. With respect to instantiation, our proposal is to regard the computed substitutions for the variables in the (sub)goals as observable internal contexts, which further instantiate the system components. Thus, given a goal G, we can apply the substitution σ to the free variables of G and study the consequent changes in the semantics.

The analogy and distinction between internal and external contexts become clear if we look at the term algebra over a signature Σ from a categorical perspective: the objects of the category are underlined natural numbers, an n-tuple of terms over m variables corresponds to an arrow from $\underline{m}$ to $\underline{n}$, and composition of arrows $t_1\colon \underline{m} \to \underline{k}$ and $t_2\colon \underline{k} \to \underline{n}$ is given by substituting the k variables in t_2 by the corresponding terms in the tuple t_1. Then, composing to the right means inserting in a context, while composing to the left means providing an internal context (e.g., t_2 above is external to t_1, while t_1 is internal to t_2).

Tile logic as a semantic framework

For pursuing this research programme, we have chosen to rely on *tile logic* (Gadducci & Montanari, 1996; Gadducci & Montanari, 2000) that can provide a convenient abstract computational model for logic programming, where many of the discussed aspects can be suitably represented and managed.

The tile framework takes inspiration from and bears many analogies with various SOS formats (Plotkin, 1981; De Simone, 1985; Bloom *et al.*, 1995; Groote & Vaandrager, 1992; Bernstein, 1998), with Larsen and Xinxin's *context systems* (Larsen & Xinxin, 1990), with *structured transition systems* defined by Corradini in joint work with the second author (Corradini & Montanari, 1992), and with Meseguer's *rewriting logic* (Meseguer, 1992). It allows to define models that are compositional both in 'space' (i.e., according to the

[1] In pure logic programming, contextualization does not provide any additional information on the possible reductions, as the head of each clause consists of only one predicate. The situation would be different if generalized multi-head Horn clauses were considered, a topic that will be discussed in the conclusions, or if second order logic were considered (other predicate contexts should be considered beside conjunction).

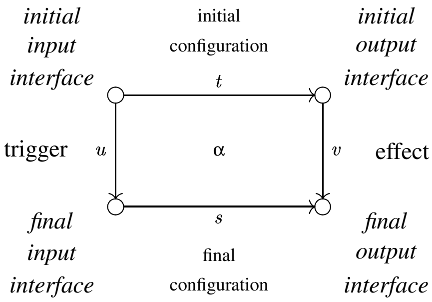

Figure 1. A tile.

structure of the system) and in 'time' (i.e., according to the computation flow). In particular, tile logic extends rewriting logic with a built-in mechanism, based on observable effects, for coordinating local rewrites. The effects are in some sense the counterparts of labels in LTS operational semantics (e.g., those arising from SOS specifications). However, since tiles are designed for dealing with open states (as opposed to the ordinary 'ground' view of LTS systems), they seem more apt for many applications. The idea is to employ a set of rules (called *tiles*) to define the behavior of partially specified components (i.e., components that can contain variables), called *configurations*, only in terms of the possible interactions with the internal/external environment, i.e., in terms of the actions related to the input and output *interfaces* of the configuration. In this way, the behavior of a system must be described as a coordinated evolution of its local subconfigurations. The name 'tile' is due to the graphical appearance of such rules, which have the form in Figure 1, also written $\alpha : t \xrightarrow[v]{u} s$, stating that the *initial configuration* t evolves to the *final configuration* s via the tile α, producing the *effect* v, which can be observed by the rest of the system, but such a step is allowed only if the subcomponents of t (i.e., the arguments to which t is connected via its input interface) evolve to the subcomponents of s, producing the effect u, which acts as the *trigger* for the application of α. Triggers and effects are called *observations* and tile vertices are called *interfaces*. The arrows t, u, v and s form the *border* of α.

Tiles can be composed horizontally, vertically, and in parallel to generate larger steps. The three compositions are illustrated in Figure 2. Horizontal composition yields rewriting synchronization (e.g., between the evolution of an argument via the tile α and the evolution of its environment via the tile β): the effect of the first tile provides the trigger for the second tile, and the resulting tile expresses the synchronized behavior of both. Vertical composition models the execution of a sequence of steps starting from an initial configuration. It corresponds to sequential composition of computations. The operation of parallel composition corresponds to building concurrent steps, where two (or more) disjoint configurations can concurrently evolve. Of course, the border of a concurrent step is the parallel composition of the borders of each component of the step.

Given a set of basic tiles, the associated *tile logic* is obtained by adding some canonical 'auxiliary' tiles and then closing by composition — horizontally, vertically, and in parallel — both auxiliary and basic tiles. As an example, auxiliary tiles may be introduced that accommodate isomorphic transformations of interfaces, yielding consistent rearrangements of configurations and observations (Bruni *et al.*, 1998; Bruni, 1999).

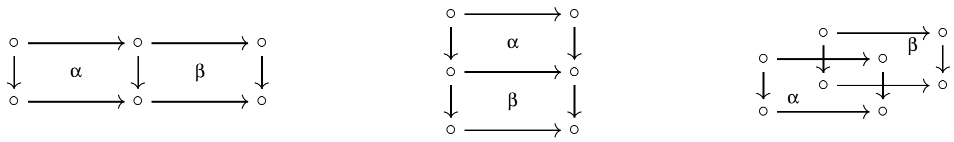

Figure 2. Horizontal, vertical and parallel tile compositions.

Tile logic deals with algebraic structures on configurations that can be different from the ordinary, tree-like presentation of terms employed in most LTS. All these structures, ranging from graphs and term graphs to partitions and relations, give rise to monoidal categories and, therefore, possess the two basic operations needed by tile configurations. This is very convenient, as the models of the logic can be formulated in terms of *monoidal double categories*. In this paper we assume the reader to be familiar with the basic concepts of category theory, though we shall not push their usage too far (employing categories in a very mild way) and shall try to informally comment on most categorical constructs presented.

Varying the algebraic structure of configurations and observations, tiles can model many different aspects of dynamic systems, ranging from synchronization of Petri net transitions (Bruni & Montanari, 2000), to causal dependencies for located calculi and finitely branching approaches for name-passing calculi (Ferrari & Montanari, 2000), to actor systems (Montanari & Talcott, 1998), names abstraction and creation and higher order structures (Bruni & Montanari, 1999). In addition, likewise Larsen and Xinxin's context systems (Larsen & Xinxin, 1990) and Rensink's *conditional transition systems* (Rensink, 2000), tile logic allows one to reason about terms with variables (Bruni *et al.*, 2000a), while SOS formats work for ground terms only. This means, e.g., that trace semantics and bisimilarity can be extended straightforwardly to open terms by taking as observation the pair $\langle \mathit{trigger}, \mathit{effect} \rangle$ (whereas ordinary LTS deal with transitions from closed terms to closed terms for which triggers are trivial identities). The compositionality of abstract semantics (either based on traces or on bisimilarity) can then be guaranteed by algebraic properties of the tile system or by suitable specification formats (Bruni *et al.*, 2000a). In particular, we shall see that the *decomposition property* (Gadducci & Montanari, 2000) yields a very simple proof of the compositionality of the tile logic associated to a logic program.

The tile approach to logic programming

A well-known fact (cf. the discussion in Section 2 and (Burstall & Rydeheard, 1985; Corradini & Montanari, 1992)) that is exploited in the construction we propose is that in categorical terms the construction of the *most general unifier* (mgu) between a subgoal and the head of a clause can be expressed as a *pullback* in the syntactic category associated to the signature under consideration. One of the contribution of this paper is in fact to provide a constructive, modular way of building the pullback construction. It is similar to the ordinary unification mechanism but formulated in a completely abstract way by means of coordination rules. This translates immediately in terms of tile logic, completing the first part of our research programme, that is, understanding the extent of interaction we shall observe, and expressing it in a formal system.

For the rest, we define a transformation from logic programs to tile systems by associating a basic tile to each Horn clause in the program. Then, the resulting tile models are shown to provide a computational and semantic framework where the program and its tile representation yield exactly the same set of computed answer substitutions for any goal.

One of the advantages of tile logic is to make evident the duality between contextualization and instantiation, still being able to deal in a uniform way with both perspectives. The same thing can be said for the uniform treatment of configurations and observations that facilitates the use of contexts as labels, providing many insights on the way the basic pieces can be put together to form a whole computation.

The tile presentation of a program allows us not only to transfer to logic programming abstract semantic equivalences based on traces and bisimilarity , but also to show that these equivalences are compositional (i.e., they are *congruences*) via an abstract proof based on the decomposition property of the underlying tile system. More concretely, denoting by $\sim$ any of the equivalences (on goals) discussed above, we have almost for free that if $G_1 \sim G_2$, then: (1) $\sigma(G_1) \sim \sigma(G_2)$ for any substitution σ, and (2) $G_1 \wedge G \sim G_2 \wedge G$ for any goal G. (This lifts also to the case where the simpler 'success' semantics is considered.)

The application of our 'tile' techniques to logic programming can serve as a basis for establishing useful connections and studying analogies with the process calculi paradigm. For example, it comes out that there is a strong resemblance between the parallel operator of many process calculi and the conjunction operator on goals. As another example, it would be interesting to transfer to logic programming concepts like 'explicit substitution' and 'term graph', which play important roles in the implementation of distributed systems.

Sources of inspiration

Before illustrating the organization of the material, we want to explain more precisely the intuition that motivated our research on interactive semantics for reduction systems and its application to logic programming.

As already pointed out, our sources of inspiration mostly come from contributions in the theory of communicating systems. The first fact to note is that there are two well recognized and widely studied schools of thought for giving semantics to process description calculi, namely via *reduction* rules (especially popular after Berry and Boudol's CHAM (Berry & Boudol, 1992)) and via *labeled transition systems* (LTS).

The first approach relies on the consideration that it is possible to observe and manipulate the global state of a complex system. In particular, the current state can be inspected for finding a redex, i.e., a candidate for the application of a reduction step. The redexes usually coordinate the activity of several logically distinct components of the system, and therefore, to some extent, the reduction step synchronizes their local activities into a global atomic move. Of course, for dealing with compositionality, one would be interested in deriving the semantics of a whole entity in terms of the semantics of its very basic component parts, which can become a hard task, as most of the times reductions are non-local actions. The problem is that a redex may lie in between a component and the external environment, and therefore to understand the behaviour of a component as a stand alone entity, we have to consider its interactions with all possible environments, in the style of *testing semantics* (De Nicola & Hennessy, 1984).

Instead, the point of view of observational equivalences based on LTS semantics is to use *observations* (the labels of the transitions) to derive observational equivalences on processes, as e.g., *bisimilarity* (Park, 1981; Milner, 1980). Moreover, the formats for specifying LTS operational semantics can exploit inductively the structure of a complex state to define its semantics in terms of the actions that can be accomplished by subcomponents, guaranteeing compositionality properties such as 'bisimilarity is a congruence.'

One emerging idea to provide reduction semantics with an interactive, observational view is that of 'observing contexts' (Milner, 1992; Montanari & Sassone, 1992; Milner, 1996; Bernstein, 1998; Sewell, 1998; Leifer & Milner, 2000; Cattani *et al.*, 2000). Basically, starting from a reduction system, one has to define the semantics of a local component by embedding it in all possible contexts and by considering those contexts as observations. Then, when several components are assembled together, it is possible to predict the semantics of the result simply by inspecting the behaviors of each component in the environment contributed by all the remaining components. This approach gives rise to a special kind of bisimulation, called *dynamic bisimilarity* (Montanari & Sassone, 1992), which is the coarsest congruence that is also an ordinary bisimulation. Though theoretically sound and satisfactory, this solution leaves open many operational questions, because the semantics must take into account all possible contexts and this should be somehow reflected in the implementation. This approach corresponds to some extent to give the possibility to dynamically reconfigure the system and has also some applications to open ended systems (Bruni *et al.*, 2000b).

Many people have tried to define a general and clever methodology for passing from reduction semantics to (compositional) LTS semantics (Sewell, 1998; Leifer & Milner, 2000; Cattani *et al.*, 2000). In particular, Leifer and Milner show in (Leifer & Milner, 2000) that such minimal set of contexts is definable whenever sufficiently many *relative pushouts* exist in the category of configurations. Roughly speaking, it must be the case that for any configuration t and any reduction rule in the systems with which t can react in a suitable environment C, then there exists a minimal observable context C' that makes such reduction possible. All mentioned contributions mainly concern compositionality results for processes taken up to bisimulation equivalences.

Dual to the problem of 'contextualization' is the problem of 'instantiation.' It arises when one wants to extend the compositionality from ground processes to open processes. In fact, the equivalence on open terms is usually defined via the equivalence on closed terms, by saying that two contexts are equivalent if their closures under all possible (ground) instantiations are so. Again, it is preferable to avoid the instantiation closure and find a more compact way to enforce the modularity of the framework. This issue has been pursued in two recent works (Rensink, 2000; Bruni *et al.*, 2000a) for providing general specification formats that guarantee the compositionality of open systems. They are based on the idea of recording in the transition labels not only the effects of each move, but also the triggers provided by the subcomponents for applying the transition to the global state. For transferring in a uniform way these techniques to reduction semantics, then instantiation should be represented as some sort of 'internal contextualization,' and correspondingly the system should be equipped with moves carrying instantiations as labels.

In the case of logic programming, many of the above concepts find a natural meaning. Thus, e.g., instantiation of a goal (i.e., application of a substitution) is a relevant internal contextualization that can modify the semantics of the goal (e.g., by making impossible the unification with the head of a clause which can otherwise be applicable). Another kind of closure is given by conjunction with other goals (it can be particularly interesting if multi-headed clauses are allowed).

Structure of the paper

We start by fixing the notation and recalling the necessary background in Section 1. Due to the heterogeneity of the material, we have decided to separate its presentation in four parts: Section 1.1 summarizes a few elementary definitions about signatures, substitutions and Horn clauses; Section 1.2 recalls the operational machinery of logic programming; Section 1.3 presents the tile notation and the categorical models based on double categories; Section 1.4 presents the concepts of Section 1.1 under a different light (exploiting Lawvere's pioneering work (Lawvere, 1963)), which will offer a more convenient notation for representing logic programs in tile logic. While the contents of Sections 1.1 and 1.2 are standard, the notions recalled in Sections 1.3 and 1.4 might be not so familiar to the LP community.

In Section 2 we recall the ways in which mgu's, equalizers and pullbacks intertwine. This should provide the reader with the formal knowledge for understanding the technical details of the correspondence between unification in logic programming and coordination via pullback tiles, which is explained in Section 3. In particular, we think that the results in Section 3.1 are the key to the application of tiles to logic programming.

Section 4 exploits the notation and results from Section 3 to establish the connection between logic programming and tile logic. The transformation is described in Section 4.1, together with a simple example that illustrates the correspondence between the two views. The main advantages of the tile approach are examined separately in Section 4.2 (connections with ongoing research on process calculi), Section 4.3 (formal correspondence with ordinary semantics), Section 4.4 (goal compositionality via abstract congruences), and in Section 4.5 (insights on concurrency and coordination). The compositionality of the resulting framework strongly depends on the representation results of Section 3.1, that allow one to decompose a complex coordination along its basic bits.

While the paper focuses on pure logic programs, we think that the approach can be extended to take into account many variants of logic programming. Some of these extensions are discussed in the concluding section.

1 Background

1.1 Notation

Let Σ be a two sorted signature over the set of sorts $\{\mathtt{t},\mathtt{p}\}$. Provided that the sort $\mathtt{p}$ does not appear in the arity of any operator, we call Σ a *logic program signature* and we denote by $\Sigma_\Phi = \bigcup_n \Sigma_\Phi^n$ and $\Sigma_\Pi = \bigcup_n \Sigma_\Pi^n$ the ranked sets of *function symbols* $f{:}\mathtt{t}^n \to \mathtt{t}$ and of *predicate symbols* $p{:}\mathtt{t}^n \to \mathtt{p}$, respectively.

As usual, given a set X of (term) variables, we denote with $\mathbb{T}_\Sigma(X)$ the free Σ-algebra generated by X. A *term over* X is an element of $\mathbb{T}_{\Sigma_\Phi}(X)$. The set of all *ground terms* (i.e., terms without variables) is called the *Herbrand universe for* Σ. An *atomic formula over* X has the form $p(t_1,\dots,t_n)$ where $p \in \Sigma^n_\Pi$ and $t_1,\dots,t_n$ are terms over X. A *conjunctive formula* is just a tuple of atomic formulas. The set of all ground atomic formulas is called the *Herbrand base* for Σ.

If $X = \{x_1,\dots,x_n\}$ and Y are sets of variables, a *substitution from* Y *to* X is a function $\sigma: X \to \mathbb{T}_{\Sigma_\Phi}(Y)$, usually denoted by $[\sigma(x_1)/x_1,\dots,\sigma(x_n)/x_n]$. If t is a term over X and σ is a substitution from Y to X then the term over Y obtained by *simultaneously* substituting in t all the occurrences of the variables in X with their images through σ is called the *application of* σ *to* t and written $\sigma; t$.

If σ is a substitution from Y to X and σ' is a substitution from Z to Y, their *composition* is the substitution $\sigma';\sigma$ from Z to X defined by applying σ' to each image of the variables in X through σ. A substitution σ is said to be *more general* than σ' if there exists a substitution θ such that $\sigma' = \theta;\sigma$.

It is worth noticing that since substitution application is associative with the identity substitutions $[x_1/x_1,\dots,x_n/x_n]$ as neutral elements, then substitutions form the arrows of a category having finite sets of variables as objects.

Two terms (also atomic formulas) t and s *unify* if there exists a substitution θ such that $\theta; t = \theta; s$. In this case θ is called a *unifier* of t and s. If t and s unify there exists also a *most general unifier* (unique up to variable renaming), *mgu* for short. The mgu can be computed by employing, e.g., the following (nondeterministic) algorithm that operates on a set of equations (at the beginning the set is the singleton $\{t = s\}$)

- *apply one of the following steps until stability is reached*
  1. *eliminate the equation* $x = x$ *from the set for some variable* x*;*
  2. *eliminate the equation* $f(t_1,\dots,t_n) = f(s_1,\dots,s_n)$ *from the set and insert the equations* $t_1 = s_1,\dots,t_n = s_n$ *in it;*
  3. *if the equation* $x = t$ *with* x *a variable not appearing in* t *is contained in the current set, apply the substitution* $[t/x]$ *to all the other equations (but do not remove* $x = t$*).*

The algorithm always terminates. It terminates with success if the resulting set of equations has the form $\{x_1 = t_1,\dots,x_n = t_n\}$ with x_i not appearing in t_j for all $i,j \in [1,n]$. The algorithm can be efficiently computed if it cyclically executes the sequence of steps 1, 3 (with $x = y$), 2, 3 (with $x = t$).

1.2 Syntax and operational semantics of logic programs

In this section we briefly recall the basics of the operational semantics of logic programs. We refer to (Lloyd, 1987) for a more detailed introduction to the subject.

A *definite Horn clause* c is an expression of the form

$$H :- B_1,\dots,B_n$$

with $n \geq 0$, where H is an atomic formula called the *head* of c and $\langle B_1,\dots,B_n\rangle$ is a (conjunctive) formula called the *body* of c. A logic program $\mathcal{P}$ is a finite collection of clauses $\{c_1,\dots,c_m\}$.

Table 1. *Operational rules for SLD-resolution (small step semantics).*

$\dfrac{}{\mathcal{P} \Vdash \ \Box, G \Rightarrow_{id} G}$	$\dfrac{}{\mathcal{P} \Vdash \ G, \Box \Rightarrow_{id} G}$	empty goal
$\dfrac{(H :- F) \in \mathcal{P} \quad \sigma = \mathrm{mgu}(A, \rho; H)}{\mathcal{P} \Vdash \ A \Rightarrow_{\sigma} \sigma; \rho; F}$		atomic goal
$\dfrac{\mathcal{P} \Vdash \ G \Rightarrow_{\sigma} F}{\mathcal{P} \Vdash \ G, G' \Rightarrow_{\sigma} F, (\sigma; G')}$	$\dfrac{\mathcal{P} \Vdash \ G \Rightarrow_{\sigma} F}{\mathcal{P} \Vdash \ G', G \Rightarrow_{\sigma} (\sigma; G'), F}$	conjunctive goal

A *goal* is an expression of the form

$$?- A_1, \ldots, A_n$$

where $G \equiv \langle A_1, \ldots, A_n \rangle$ is a (conjunctive) formula and the A_i's are the *atomic subgoals* of G. If $n = 0$, then G is called the *empty goal* and is denoted by '$\Box$.'

Given a goal $G \equiv \langle A_1, \ldots, A_k \rangle$ and a clause $c \equiv H :- B_1, \ldots, B_n$ (all variables in the latter possibly renamed to avoid confusion with those in G) an *(SLD)-resolution step* involves the selection of an atomic goal A_i such that H and A_i unify and the construction of their mgu θ. The step leads to a new goal

$$\begin{aligned} G' &\equiv \theta; \langle A_1, \ldots, A_{i-1}, B_1, \ldots, B_n, A_{i+1}, \ldots, A_k \rangle \\ &\equiv \theta; A_1, \ldots, \theta; A_{i-1}, \theta; B_1, \ldots, \theta; B_n, \theta; A_{i+1}, \ldots, \theta; A_k \end{aligned}$$

which is called the *resolvent* of G and c. In this case we say that G' is *derived from* G *and* c *via* θ and we write $G \Rightarrow_{c,\theta} G'$ or simply $G \Rightarrow_{\theta} G'$.

Given a logic program $\mathcal{P} = \{c_1, \ldots, c_m\}$ and a goal G_0, an *(SLD-)derivation* of G_0 in $\mathcal{P}$ is a (finite or infinite) sequence $G^0, G^1, G^2, \ldots$ of goals, a sequence $c_{i_1}, c_{i_2}, \ldots$ of (renamed) clauses and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that G^{i+1} is derived from G^i and c_{j_i} via θ_i. An *(SLD-)refutation* of G_0 is a finite derivation of G_0 ending with the empty goal. In this case the substitution $\theta = (\theta_l; \cdots; \theta_1)_{|Var(G)}$ is called a *computed answer substitution* for G, written $G \Rightarrow^*_{\theta} \Box$. The 'small-step' operational semantics is formalized in Table 1 (but in the rule for atomic goal we must be certain that ρ renames the variables in the clause by globally fresh names).

The inductive definition of computed answer substitution and refutation in the 'big-step' style is given by the rules in Table 2. The notation $\mathcal{P} \vdash_{\theta} G$ means that $\mathcal{P} \Vdash \ G \Rightarrow^*_{\theta} \Box$, i.e., that the goal G can be refuted by using clauses in the program $\mathcal{P}$. The first rule says that the empty goal can be always refuted with the empty computed answer substitution ε. The second clause says that an atomic goal can be refuted provided that it can be unified with the head H of a clause (suitably renamed by ρ to avoid name conflicts with A) in the program $\mathcal{P}$ via the mgu σ and that the goal obtained by applying σ to the (renamed) body F of the clause can be refuted with θ. The third rule says that a conjunctive goal can be refuted provided that its leftmost subgoal can be refuted first with σ, and then the goal

Table 2. *Operational rules for SLD-resolution (big step semantics).*

$$\frac{}{\mathcal{P} \vdash_{\varepsilon} \square}$$	empty goal
$$\frac{(H :- F) \in \mathcal{P} \quad \sigma = \mathrm{mgu}(A, \rho; H) \quad \mathcal{P} \vdash_{\theta} \sigma; \rho; F}{\mathcal{P} \vdash_{\theta;\sigma} A}$$	atomic goal
$$\frac{\mathcal{P} \vdash_{\sigma} A \quad \mathcal{P} \vdash_{\theta} \sigma; F}{\mathcal{P} \vdash_{\theta;\sigma} A, F}$$	conjunctive goal

obtained by applying σ to the other subgoals can be refuted with θ. Although imposing a sequentialization in the resolution process can appear as an arbitrary choice, the fact that refutation involves finite derivations and the well known switching lemma guarantee the completeness of the formal system.

1.3 Double categories and tile logic

The point of view of tile logic (Gadducci & Montanari, 2000; Bruni, 1999) is that the dynamics of a complex system can be better understood if we reason in terms of its basic components and of the interactions between them. Therefore, reductions must carry observable information about the triggers and the effects of the local step. This extends the point of view of rewriting logic, where reductions can be freely nested inside any context (and also freely instantiated): In tile logic, contextualization and instantiation are subordinated to the synchronization of the arguments with the environment, i.e., the effect of the tile defining the evolution of the former must provide the trigger for the evolution of the second. When the coordination is not possible, then the step cannot be performed.

An abstract account of the connections between states and dynamics can be given via (monoidal) *double categories*, by exploiting their two-fold representation: one dimension is for composing states and the second dimension is for composing computations. In fact, the models of tile logic are suitable double categories (Gadducci & Montanari, 2000; Bruni *et al.*, 2000c), and the basic tiles of a tile system provides a finitary presentation — which is more convenient to work with — of the initial model.

Since we do not want to introduce unnecessary complexity overhead to readers not acquainted with double categories, we will present a gentle introduction to the subject. For more details we refer to (Ehresmann, 1963a; Ehresmann, 1963b; Meseguer & Montanari, 1998; Bruni *et al.*, 1998; Bruni *et al.*, 2000c).

A double category contains two categorical structures, called *horizontal* and *vertical* respectively, defined over the same set of cells. More precisely, double categories admit the following naïve definition.

Definition 1.1 (*Double Category*)
A *double category* $\mathcal{D}$ consists of a collection $o, o_0, o', \ldots$ of *objects*, a collection $h, h_0, h', \ldots$

$$\begin{tikzcd} o_0 \arrow[r, "h_0"] \arrow[d, "v_0"'] \arrow[dr, phantom, "A"] & o_1 \arrow[d, "v_1"] \\ o_2 \arrow[r, "h_1"'] & o_3 \end{tikzcd}$$

Figure 3. Graphical representation of a cell.

$$\begin{tikzcd} \cdot \arrow[r, "h_0"] \arrow[d, "v_0"'] \arrow[dr, phantom, "A"] & \cdot \arrow[r, "h_2"] \arrow[d, "v_1"] \arrow[dr, phantom, "B"] & \cdot \arrow[d, "v_2"] \\ \cdot \arrow[r, "h_1"] \arrow[d, "v_3"'] \arrow[dr, phantom, "C"] & \cdot \arrow[r, "h_3"] \arrow[d, "v_4"] \arrow[dr, phantom, "D"] & \cdot \arrow[d, "v_5"] \\ \cdot \arrow[r, "h_4"'] & \cdot \arrow[r, "h_5"'] & \cdot \end{tikzcd}$$

Figure 4. Exchange law of double categories.

of *horizontal arrows*, a collection $v, v_0, v', \ldots$ of *vertical arrows* and a collection $A, B, C, \ldots$ of *cells*.

- Objects and horizontal arrows form the *horizontal 1-category* $\mathcal{H}$, with identity id_o for each object o, and composition $\_ * \_$.
- Objects and vertical arrows form also a category, called the *vertical 1-category* $\mathcal{V}$, with identity id_o for each object o, and composition $\_ \cdot \_$.
- Cells are assigned *horizontal source* and *target* (which are vertical arrows) and *vertical source* and *target* (which are horizontal arrows); furthermore sources and targets must be *compatible*, in the sense that they must satisfy the equalities on source and target objects graphically represented by the square-shaped diagram in Figure 3, for which we use the notation $A : h_0 \xrightarrow[v_1]{v_0} h_1$.
- Cells can be composed both horizontally ($\_ * \_$) and vertically ($\_ \cdot \_$) as follows: given $A : h_0 \xrightarrow[v_1]{v_0} h_1$, $B : h_2 \xrightarrow[v_2]{v_1} h_3$, and $C : h_1 \xrightarrow[v_4]{v_3} h_4$, then $A * B : h_0 * h_2 \xrightarrow[v_2]{v_0} h_1 * h_3$, and $A \cdot C : h_0 \xrightarrow[v_1 \cdot v_4]{v_0 \cdot v_3} h_4$ are cells. Moreover, given a fourth cell $D : h_3 \xrightarrow[v_5]{v_4} h_5$, horizontal and vertical compositions satisfy the following *exchange law* (see Figure 4):

  $$(A \cdot C) * (B \cdot D) = (A * B) \cdot (C * D)$$

  Under these rules, cells form both a horizontal category $\mathcal{D}^*$ and a vertical category $\mathcal{D}^{\cdot}$, with identities $1_v : id_o \xrightarrow[v]{v} id_{o'}$ and $1^h : h \xrightarrow[id_{o'}]{id_o} h$, respectively, with $1^{h_0 * h_1} = 1^{h_0} * 1^{h_1}$ and $1_{v_0 \cdot v_1} = 1_{v_0} \cdot 1_{v_1}$.
- Furthermore, horizontal and vertical identities of identities coincide, i.e., $1_{id_o} = 1^{id_o}$.

We shall use monoidal categories (see e.g., (MacLane, 1971) for basic definitions) for horizontal and vertical 1-categories. As a matter of notation, sequential composition and monoidal tensor product on 1-categories are denoted by $\_;\_$ and $\_ \otimes \_$, respectively. A *monoidal double category* is a double category together with an associative tensor product $\_ \otimes \_$ and a unit element e.

$$\frac{t_0 \xrightarrow[v]{u} s_0 \quad t_1 \xrightarrow[w]{v} s_1}{t_0;t_1 \xrightarrow[w]{u} s_0;s_1} \qquad \frac{t \xrightarrow[v_0]{u_0} s \quad s \xrightarrow[v_1]{u_1} r}{t \xrightarrow[v_0;v_1]{u_0;u_1} r} \qquad \frac{t_0 \xrightarrow[v_0]{u_0} s_0 \quad t_1 \xrightarrow[v_1]{u_1} s_1}{t_0 \otimes t_1 \xrightarrow[v_0 \otimes v_1]{u_0 \otimes u_1} s_0 \otimes s_1} \qquad \frac{t\colon o_0 \to o_1 \in \mathcal{H}}{t \xrightarrow[o_1]{o_0} t} \qquad \frac{u\colon o_0 \to o_1 \in \mathcal{V}}{o_0 \xrightarrow[u]{u} o_1}$$

Figure 5. Composition and identity rules for tile logic.

Tile logic gives a computational interpretation of double categories, where:

- the objects represent the interfaces through which system components can be connected;
- the arrows of $\mathcal{H}$ describe configurations, sources and targets corresponding to input and output interfaces;
- the arrows of $\mathcal{V}$ are the observations, sources and targets corresponding to initial and final interfaces;
- the double cells represent the possible transformations that can be performed by the system.

Thus, a cell $h_0 \xrightarrow[v_1]{v_0} h_1$ says that the state h_0 can evolve to h_1 via an action triggered by v_0 and with effect v_1. The way in which observations and configurations of a cell are connected via their interfaces expresses the locality of actions, i.e., the places where triggers are applied to and effects are produced by.

A basic distinction concerns whether one is interested in the cells or just in their borders. The second alternative has a more abstract flavor, in line with behavioral equivalences, and corresponds to the so called *flat tile logic*. In this paper we shall concentrate on flat tiles only.

Tile logic gives also the possibility of presenting in a constructive way the double category of interest. This is in some sense analogous to presenting a term algebra by giving only the signature: A standard set of rules tells how to build all the elements starting from the basic ones. For tile logic, the basic elements consist of: (i) the category $\mathcal{H}$ of configurations; (ii) the category $\mathcal{V}$ of observations; and (iii) the set of basic tiles (i.e., double cells on $\mathcal{H}$ and $\mathcal{V}$). Starting from basic tiles, more complex tiles can be constructed by means of horizontal, vertical and parallel composition. Moreover, the horizontal and vertical identities are always added and composed together with the basic tiles. All this is illustrated by the rules in Figure 5, where tiles are seen as logic sequents. As explained in the Introduction, each operation has a precise computational meaning: horizontal composition coordinates the evolution of a context with that of its arguments; parallel composition models concurrent activities; vertical composition appends steps to computations. For both terms and tiles, the operation of building all the elements starting from the basic ones can be represented by a universal construction corresponding to a left adjoint.

Definition 1.2 (*Tile system*)
A *tile system* is a tuple $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ where $\mathcal{H}$ and $\mathcal{V}$ are monoidal categories with the same set of objects $\mathbf{O}_{\mathcal{H}} = \mathbf{O}_{\mathcal{V}}$, N is the set of rule names and $R\colon N \to \mathbf{A}_{\mathcal{H}} \times \mathbf{A}_{\mathcal{V}} \times \mathbf{A}_{\mathcal{V}} \times \mathbf{A}_{\mathcal{H}}$ is a function such that for all $\alpha \in N$, if $R(\alpha) = \langle t, u, v, s\rangle$ then $t\colon o_0 \to o_1$, $u\colon o_0 \to o_2$, $v\colon o_1 \to o_3$, and $s\colon o_2 \to o_3$ for suitable objects o_0, o_1, o_2 and o_3 (see Figure 6). We will denote such rule by writing $\alpha\colon t \xrightarrow[v]{u} s$.

$$\begin{tikzcd} o_0 \arrow[r, "t"] \arrow[d, "u"'] \arrow[dr, phantom, "\alpha"] & o_1 \arrow[d, "v"] \\ o_2 \arrow[r, "s"'] & o_3 \end{tikzcd}$$

Figure 6. A generic tile α.

Depending on the chosen tile format, $\mathcal{H}$ and $\mathcal{V}$ can be specialized (e.g., to cartesian categories) and suitable *auxiliary tiles* are added and composed with basic tiles and identities in all the possible ways. The set of resulting sequents (*flat tiles*) define the *flat tile logic* associated to $\mathcal{R}$. We say that $t \xrightarrow[v]{u} s$ is *entailed* by the logic, written $\mathcal{R} \vdash t \xrightarrow[v]{u} s$, if the sequent $t \xrightarrow[v]{u} s$ can be expressed as the composition of basic and auxiliary tiles. Flat tiles form the cells of a suitable double category, which is freely generated by the tile system.

Being interested in tile systems where configurations and observations are freely generated by suitable horizontal and vertical signatures, in what follows we shall present tile systems as tuples of the form $\mathcal{R} = (\Sigma, \Lambda, N, R)$. In particular, we shall employ categories of substitutions on Σ and Λ as horizontal and vertical 1-categories. In the literature several tile formats have been considered (Gadducci & Montanari, 2000; Ferrari & Montanari, 2000; Bruni *et al.*, 1999; Bruni & Montanari, 1999; Bruni *et al.*, 2000a). They are all based on the idea of having as underlying categories of configurations and effects two categories that are freely generated starting from suitable (hyper)signatures whose operators model the basic components and observations, respectively. A comparison between the various formats is out of the scope of this presentation and can be found in (Bruni *et al.*, 2000a).

Ordinary trace semantics and bisimilarity semantics can be extended to tiles by considering the transition system whose states are (possibly open) configuration and whose transitions are the entailed tile sequents: a tile $t \xrightarrow[v]{u} s$ defines a transition from t to s with label (u, v). An interesting question concerns suitable conditions under which such abstract equivalences yields congruences (w.r.t. the operations of the underlying horizontal structure). *Tile decomposition* is one such condition that has a completely abstract formulation applicable to all tile systems.

Definition 1.3
A tile system $\mathcal{R} = (\mathcal{H}, \mathcal{V}, N, R)$ enjoys the *decomposition property* if for all arrows $t \in \mathcal{H}$ and for all sequents $t \xrightarrow[v]{u} s$ entailed by $\mathcal{R}$, then: (1) if $t = t_1; t_2$ then $\exists w \in \mathcal{V}$, $s_1, s_2 \in \mathcal{H}$ such that $\mathcal{R} \vdash t_1 \xrightarrow[w]{u} s_1$, $\mathcal{R} \vdash t_2 \xrightarrow[v]{w} s_2$ and $s = s_1; s_2$; (2) if $t = t_1 \otimes t_2$ then $\exists u_1, u_2, v_1, v_2 \in \mathcal{V}$, $s_1, s_2 \in \mathcal{H}$ such that $\mathcal{R} \vdash t_1 \xrightarrow[v_1]{u_1} s_1$, $\mathcal{R} \vdash t_2 \xrightarrow[v_2]{u_2} s_2$, $u = u_1 \otimes u_2$, $v = v_1 \otimes v_2$ and $s = s_1 \otimes s_2$.

Condition (1) is called *sequential decomposition* and condition (2) is called *parallel decomposition*. Decomposition property characterizes compositionality: It amounts to saying that if a system t can undergo a transition α, then for every subsystem t_1 of t there exists some transition α', such that α can be obtained by composing α' with a transition of the rest.

Proposition 1 (*cf. (Gadducci & Montanari, 2000)*)
If $\mathcal{R}$ enjoys the decomposition property, then tile bisimilarity (and also tile trace equivalence) are congruences.

When only instantiation/contextualization are considered as meaningful operations of the system, then sequential decomposition is enough for guaranteeing the congruence of tile bisimilarity and tile trace equivalence w.r.t. these closure operations.

1.4 Algebraic theories

An alternative presentation of the category of substitutions discussed in Section 1.1 can be obtained resorting to *algebraic theories* (Lawvere, 1963).

Remark 1.1
For simplicity we illustrate here the constructions for one-sorted signatures. This can be extended to many sorted signatures by considering the free strict monoid on the set of sorts (e.g., strings of sorts) in place of underlined natural numbers.

Definition 1.4 (*Algebraic theory*)
The free algebraic theory associated to a signature Σ is the category $\mathbf{Th}[\Sigma]$ defined below:

- its objects are 'underlined' natural numbers;
- the arrows from $\underline{m}$ to $\underline{n}$ are n-tuples of terms in the free Σ-algebra with (at most) m canonical variables, and composition of arrows is term substitution. The arrows of $\mathbf{Th}[\Sigma]$ are generated from Σ by the inference rules in Figure 7, modulo the axioms in Table 3.

The category $\mathbf{Th}[\Sigma]$ is isomorphic to the category of finite substitutions on Σ (with canonical sets of variables), and the arrows from $\underline{0}$ to $\underline{1}$ are in bijective correspondence with the closed terms over Σ.

An object $\underline{n}$ (interface) can be thought of as representing the n (ordered) canonical variables $x_1,\ldots,x_n$. This allows us to denote $[t_1/x_1,\ldots,t_n/x_n]$ just by the tuple $\langle t_1,\ldots,t_n\rangle$, since a standard naming of substituted variables can be assumed. We omit angle brackets if no confusion can arise.

Remark 1.2
To avoid confusion, it must be clear that the canonical variables are just placeholders, i.e., their scope is only local. For example in $[f(x_1)/x_1]$ the two x_1 are different, while in $[f(x_1)/x_1, g(x_1)/x_2]$ only the two occurrences of x_1 in $f(x_1)$ and $g(x_1)$ refer the same placeholder. Note that $[f(x_1)/x_1, g(x_2)/x_1]$ is inconsistent (because x_1 is assigned twice) and in fact cannot be expressed in the language.

The rule `op` defines basic substitutions $[f(x_1,\ldots,x_n)/x_1] = f(x_1,\ldots,x_n)$ for all $f\in\Sigma_n$. The rule `id` yields identity substitutions $\langle x_1,\ldots,x_n\rangle$. The rule `seq` represents application of α to β. The rule `mon` composes substitutions in parallel (in $\alpha\otimes\beta$, α goes from $x_1,\ldots,x_n$ to $x_1,\ldots,x_m$, while β goes from $x_{n+1},\ldots,x_{n+k}$ to $x_{m+1},\ldots,x_{m+l}$). Three 'auxiliary' operators (i.e., not dependent on Σ) are introduced that recover the cartesian structure (rules `sym`, `dup` and `dis`). The *symmetry* $\gamma_{\underline{n},\underline{m}}$ is the permutation $\langle x_{n+1},\ldots,x_{n+m},x_1,\ldots,x_n\rangle$. The *duplicator* $\nabla_{\underline{n}} = \langle x_1,\ldots,x_n,x_1,\ldots,x_n\rangle$ introduces sharing and hence nonlinear substitutions. The *discharger* $!_{\underline{n}}$ is the empty substitution on $x_1,\ldots,x_n$, recovering cartesian projections.

$$\texttt{op}\ \frac{f \in \Sigma_n}{f\colon \underline{n} \to \underline{1}} \qquad \texttt{id}\ \frac{n \in \mathbb{N}}{id_{\underline{n}}\colon \underline{n} \to \underline{n}} \qquad \texttt{seq}\ \frac{\alpha\colon \underline{n} \to \underline{m} \quad \beta\colon \underline{m} \to \underline{k}}{\alpha;\beta\colon \underline{n} \to \underline{k}} \qquad \texttt{mon}\ \frac{\alpha\colon \underline{n} \to \underline{m} \quad \beta\colon \underline{k} \to \underline{l}}{\alpha\otimes\beta\colon \underline{n+k} \to \underline{m+l}}$$

$$\texttt{sym}\ \frac{n, m \in \mathbb{N}}{\gamma_{\underline{n},\underline{m}}\colon \underline{n+m} \to \underline{m+n}} \qquad \texttt{dup}\ \frac{n \in \mathbb{N}}{\nabla_{\underline{n}}\colon \underline{n} \to \underline{n+n}} \qquad \texttt{dis}\ \frac{n \in \mathbb{N}}{!_{\underline{n}}\colon \underline{n} \to \underline{0}}$$

Figure 7. The inference rules for the generation of **Th**[Σ].

Table 3. *Axiomatization of* ***Th***[Σ].

category	$\alpha;(\beta;\delta) = (\alpha;\beta);\delta$		$\alpha; id_{\underline{m}} = \alpha = id_{\underline{n}};\alpha$
tensor product	$(\alpha;\alpha')\otimes(\beta;\beta') = (\alpha\otimes\beta);(\alpha'\otimes\beta')$ $\alpha\otimes(\beta\otimes\delta) = (\alpha\otimes\beta)\otimes\delta$		$id_{\underline{n+m}} = id_{\underline{n}}\otimes id_{\underline{m}}$ $\alpha\otimes id_{\underline{0}} = \alpha = id_{\underline{0}}\otimes\alpha$
symmetries	$\gamma_{\underline{n},\underline{m+k}} = (\gamma_{\underline{n},\underline{m}}\otimes id_{\underline{k}});(id_{\underline{m}}\otimes\gamma_{\underline{n},\underline{k}})$	$\gamma_{\underline{n},\underline{0}} = id_{\underline{n}}$	$\gamma_{\underline{n},\underline{m}};\gamma_{\underline{m},\underline{n}} = id_{\underline{n+m}}$
duplicators	$\nabla_{\underline{n+m}} = (\nabla_{\underline{n}}\otimes\nabla_{\underline{m}});(id_{\underline{n}}\otimes\gamma_{\underline{n},\underline{m}}\otimes id_{\underline{m}})$ $\nabla_{\underline{n}};(id_{\underline{n}}\otimes\nabla_{\underline{n}}) = \nabla_{\underline{n}};(\nabla_{\underline{n}}\otimes id_{\underline{n}})$		$\nabla_{\underline{0}} = id_{\underline{0}}$ $\nabla_{\underline{n}};\gamma_{\underline{n},\underline{n}} = \nabla_{\underline{n}}$
discharger	$!_{\underline{n+m}} = !_{\underline{n}}\otimes !_{\underline{m}}$	$!_{\underline{0}} = id_{\underline{0}}$	$\nabla_{\underline{n}};(id_{\underline{n}}\otimes !_{\underline{n}}) = id_{\underline{n}}$
naturality	$(\alpha\otimes\beta);\gamma_{\underline{m},\underline{l}} = \gamma_{\underline{n},\underline{k}};(\beta\otimes\alpha)$	$\alpha;\nabla_{\underline{m}} = \nabla_{\underline{n}};(\alpha\otimes\alpha)$	$\alpha;!_{\underline{m}} = !_{\underline{n}}$

Let us briefly comment on the axiomatization in Table 3. The first two rows say that **Th**[Σ] is a strict monoidal category, with tensor product $\otimes$ and neutral element $id_{\underline{0}}$. The third row and the naturality axiom for symmetries (first axiom of the last row) say that **Th**[Σ] is also symmetric. In particular, the axioms in the third row states the coherence of symmetries $\gamma_{\underline{n},\underline{m}}$, namely that all the equivalent ways of swapping the first n variables with the following m variables built out of the basic symmetry $\gamma_{\underline{1},\underline{1}}$ (that swaps two adjacent variables) are identified. The axioms in the fourth row accomplish a similar task for duplicators, and those in the fifth row for dischargers. The naturality of duplicators and dischargers (second and third axioms of the last row) makes **Th**[Σ] cartesian.

This presentation shows that all the auxiliary structure can be generated by composing together three basic constructors ($\gamma_{\underline{1},\underline{1}}$, $\nabla_{\underline{1}}$ and $!_{\underline{1}}$), i.e., it admits a finitary specification. Moreover, we think that this construction nicely separates the syntactic structure of the signature from the additional auxiliary structure common to all cartesian models.

This presentation has been exploited in (Bruni *et al.*, 2000a) for defining a taxonomy of tile formats as certain axioms or operators are omitted from the configuration and observation categories. In particular the auxiliary tiles needed in all such formats can be characterized as the bidimensional counterparts of symmetries, duplicators and dischargers. For example, let us mention that the auxiliary tiles of *term tile logic* (Bruni *et al.*, 1998) (where the category of configurations and observations are freely generated cartesian categories **Th**[Σ] and **Th**[Λ]) are the commuting squares of the category **Th**[∅] generated by the empty signature.

$$
\begin{tikzcd}
o' \arrow[drr, bend left, "q_0"] \arrow[dr, dotted, "q"] \arrow[ddr, bend right, "q_1"'] & & \\
& o \arrow[r, "p_0"] \arrow[d, "p_1"'] & o_0 \arrow[d, "h"] \\
& o_1 \arrow[r, "g"'] & o_2
\end{tikzcd}
$$

Figure 8. The pullback of h and g.

2 Resolution via pullbacks

As we have briefly recalled in the Introduction, the construction of the mgu has a clear mathematical meaning: It can be formulated in the terminology of category theory as a well-known universal construction called pullback (taken in a suitable category).

Universal constructions play a fundamental role in category theory, as they express the best way to accomplish a certain task. They usually involve a diagram that imposes certain constraints on the construction and then require the existence and uniqueness in the category of certain arrows satisfying such constraints, i.e., representing a possible solution to the problem. Among these solutions, one is of course interested in the optimal one (if it exists), e.g., the least upper bound. Categorically speaking, this is achieved by taking the solution that uniquely factorizes all the other solutions. Note that since many such optimal solutions can exist, any of them is completely equivalent to all the others (they are indeed pairwise isomorphic). Universal constructions allow one to recast ordinary set-theoretic constructions (e.g., cartesian product, disjoint sum) in a more general, abstract formulation that can serve as a uniform guide for catching analogies and pursuing comparison between different frameworks.

Definition 2.1 (*Pullback*)
Given a category $\mathcal{C}$ and two arrows $h\colon o_0 \to o_2$ and $g\colon o_1 \to o_2$ in $\mathcal{C}$, the *pullback* of h and g in $\mathcal{C}$ is an object o together with two projections $p_0\colon o \to o_0$ and $p_1\colon o \to o_1$ such that

1. $p_0; h = p_1; g$, and
2. for any object o' and arrows $q_0\colon o' \to o_0$ and $q_1\colon o' \to o_1$ such that $q_0; h = q_1; g$, then there must exist a unique arrow $q\colon o' \to o$ such that $q; p_0 = q_0$ and $q; p_1 = q_1$.

The two arrows h and g encode the instance of the problem, posing constraints on the admissible solutions. The first condition says that o, p_0 and p_1 yield a solution (called a *cone* in category theory). The second condition states that o, p_0 and p_1 form the best solution among those contained in $\mathcal{C}$. The commuting diagram in Figure 8 illustrates the definition (as usual in category theory, universal arrows are dotted).

Example 2.1 (*Pullbacks in* **Set**)
The category **Set** has sets as objects and functions as arrows. Given $h\colon X \to Z$ and $g\colon Y \to Z$, then their pullback is the set $U = \{(x, y) \in X \times Y \mid h(x) = g(y)\}$ with the obvious projections on the first and second components of each pair in U.

The category we are interested in is the category of substitutions on the signature Σ. It is

$$\begin{tikzcd} o \arrow[r, "p"] & o_1 \arrow[r, bend left, "h"] \arrow[r, bend right, "g"'] & o_2 \\ o' \arrow[u, dotted, "q'"] \arrow[ur, "q"'] & & \end{tikzcd}$$

(a)

$$\begin{tikzcd} o \arrow[dr, dotted, "r"] \arrow[drr, bend left, "p"] \arrow[ddr, bend right, "p"'] & & \\ & o_1' \arrow[r, "p_1"] \arrow[d, "p_1'"'] & o_1 \arrow[d, "h"] \\ & o_1 \arrow[r, "g"'] & o_2 \end{tikzcd}$$

(b)

Figure 9. The equalizer (a) and the pullback of h and g in the same homset (b).

well known that the mgu of a set of equations is an equalizer in the category of substitutions (see e.g., (Burstall & Rydeheard, 1985), though there the authors work with the opposite category $\mathbf{Th}[\Sigma]^{\mathrm{op}}$ of $\mathbf{Th}[\Sigma]$, and therefore the mgu's are given by coequalizers).

Definition 2.2 (*Equalizer*)
Given a category $\mathcal{C}$ and two arrows $h\colon o_1 \to o_2$ and $g\colon o_1 \to o_2$, the *equalizer* of h and g in $\mathcal{C}$ is an object o together with a projection $p\colon o \to o_1$ such that

1. $p;h = p;g$, and
2. for any object o' and arrow $q\colon o' \to o_1$ such that $q;h = q;g$, then there must exist a unique arrow $q'\colon o' \to o$ such that $q';p = q$.

The diagram summarizing the equalizer construction is given in Figure 9(a).

Example 2.2 (*Equalizers in* **Set**)
Given $h\colon X \to Z$ and $g\colon X \to Z$, then their equalizer is the subset $U = \{x \in X \mid h(x) = g(x)\}$ of X with the obvious inclusion $U \hookrightarrow X$ as projection.

Note that, in general, for $h,g\colon o_1 \to o_2$ the pullback of h and g is not isomorphic to their equalizer. Moreover, when both exist, the cone (o,p,p) (obtained by taking twice the projection of the equalizer) uniquely factorizes through the pullback (o_1',p_1,p_1') (see Figure 9(b)).

For a given set of equations $\{t_1 = s_1, t_2 = s_2, \ldots, t_n = s_n\}$, we can consider the substitutions $\sigma = [t_1/z_1, t_2/z_2, \cdots, t_n/z_n]$ and $\sigma' = [s_1/z_1, s_2/z_2, \cdots, s_n/z_n]$, where the z_i's are fresh variables not appearing in the t_i's and s_i's. The substitutions σ and σ' can be seen as arrows going from the set of variables appearing in the t_i's and s_i's to the set $Z = \{z_1, z_2, \ldots, z_n\}$. To see how the definition of equalizer mathches that of mgu, just observe that (1) it requires the existence of a substitution θ such that $\theta;\sigma = \theta;\sigma'$ and (2) the fact that θ is the most general such substitutions corresponds to the universal property of equalizers.

However, in the case of logic programming, we are not really interested in finding the mgu of a generic set of equations, because we know that the variables in the head of the selected clause have been renamed on purpose to be different from those in the selected goal, i.e., they are fresh. Thus, we want to find the mgu of a set of equations $\{t_1 = s_1, t_2 = s_2, \ldots, t_n = s_n\}$ such that the variables appearing in t_i and s_j are disjoint for $i,j \in [1,n]$. Then, we can consider the substitutions $\sigma_* = [t_1/z_1, t_2/z_2, \cdots, t_n/z_n]$ and $\sigma_*' = [s_1/z_1, s_2/z_2, \cdots, s_n/z_n]$, where the z_i's are fresh variables not appearing in the t_i's

$$
\begin{tikzcd}
U \arrow[r, "\theta"] & X \cup Y \arrow[r, shift left, "\sigma"] \arrow[r, shift right, "\sigma'"'] & Z \\
V \arrow[u, dotted, "\phi"] \arrow[ur, "\rho_*"'] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & & X \arrow[dr, "\sigma_*"] & \\
V \arrow[urr, bend left, "\rho"] \arrow[r, dotted, "\phi"] \arrow[drr, bend right, "\rho'"'] & U \arrow[ur, "\theta_X"] \arrow[dr, "\theta_Y"'] & & Z \\
 & & Y \arrow[ur, "\sigma'_*"'] &
\end{tikzcd}
$$

Figure 10. From equalizer to pullback.

and s_i's. If we denote by X the set of variables appearing in the t_i's, by Y the set of variables used in the s_i's, and by Z the set $\{z_1, z_2, \ldots, z_n\}$, then we can write $\sigma_*: X \to Z$ and $\sigma'_*: Y \to Z$. Their pullbacks (when it exists) is thus given by a pair of substitutions $\psi_*: U_* \to X$ and $\psi'_*: U_* \to Y$ such that

- $\psi_*;\sigma_* = \psi'_*;\sigma'_*$, and
- for any substitutions $\rho: V \to X$ and $\rho': V \to Y$ such that $\rho;\sigma_* = \rho';\sigma'_*$, then there must exist a unique substitution $\phi: V \to U_*$ such that $\phi;\psi_* = \rho$ and $\phi;\psi'_* = \rho'$.

Since the fact that the notion of pullback for σ_* and σ'_* coincide with the notion of equalizer for σ and σ' is not completely straightforward, we illustrate below such a correspondence.

From equalizers to pullbacks. Consider the arrows $\sigma: X \cup Y \to Z$ and $\sigma': X \cup Y \to Z$ that are defined exactly as σ_* and σ'_* but have different domains. Then, we know that their mgu is the equalizer $\theta: U \to X \cup Y$ discussed above. Since θ is a substitution and since $X = \{x_1, \ldots, x_k\}$ and $Y = \{y_1, \ldots, y_h\}$ are disjoint, then θ must have the form

$$[r_1/x_1, \cdots, r_k/x_k, r'_1/y_1, \cdots, r'_k/y_h].$$

Then, $\theta_X = [r_1/x_1, \cdots, r_k/x_k]: U \to X$ and $\theta_Y = [r'_1/y_1, \cdots, r'_k/y_h]: U \to Y$ satisfy $\theta_X;\sigma_* = \theta_Y;\sigma'_*$. We want to show that U, θ_X and θ_Y define a pullback for σ_* and σ'_*. In fact, suppose that there exist V with $\rho: V \to X$ and $\rho': V \to Y$ such that $\rho;\sigma_* = \rho';\sigma'_*$, then since X and Y are disjoint, ρ and ρ' can be combined together in a substitution $\rho_*: V \to X \cup Y$ such that $\rho_*;\sigma = \rho_*;\sigma'$. By definition of equalizer, then there exists a unique arrow $\phi: V \to U$ such that $\phi;\theta = \rho_*$. But the last condition is equivalent to impose that $\phi;\theta_X = \rho$ and $\phi;\theta_Y = \rho'$ concluding the proof. All this is illustrated in Figure 10.

From pullbacks to equalizers. It remains to show that to each pullback (U', ψ_X, ψ_Y) of σ_* and σ'_* there corresponds an mgu (i.e., an equalizer) for σ and σ'. By arguments similar to those employed above, it is evident that ψ_X and ψ_Y can be merged to define a substitution $\psi_*: U' \to X \cup Y$ such that $\psi_*;\sigma = \psi_*;\sigma'$. Then, we must show that this candidate is indeed an equalizer. Thus, we assume the existence of V and $\rho_*: V \to X \cup Y$ such that $\rho_*;\sigma = \rho_*;\sigma'$. As before, we can decompose ρ_* into $\rho_X: V \to X$ and $\rho_Y: V \to Y$ with $\rho_X;\sigma_* = \rho_Y;\sigma'_*$. By definition of pullback, then there exists a unique arrow $\phi_*: V \to U'$ such that $\phi_*;\psi_X = \rho_X$ and $\phi_*;\psi_Y = \rho_Y$, and the last two conditions are equivalent to the constraint $\phi_*;\psi_* = \rho_*$, concluding the proof. All this is illustrated in Figure 11.

$$
\begin{tikzcd}
& & X \arrow[dr, "\sigma_*"] & \\
V \arrow[r, dotted, "\phi_*"] \arrow[urr, bend left, "\rho_X"] \arrow[drr, bend right, "\rho_Y"'] & U' \arrow[ur, "\psi_X"'] \arrow[dr, "\psi_Y"] & & Z \\
& & Y \arrow[ur, "\sigma'_*"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
U' \arrow[r, "\psi_*"] & X \cup Y \arrow[r, shift left, "\sigma"] \arrow[r, shift right, "\sigma'"'] & Z \\
V \arrow[u, dotted, "\phi_*"] \arrow[ur, "\rho_*"'] & &
\end{tikzcd}
$$

Figure 11. From pullback to equalizer.

$$
\begin{tikzcd}
o' \arrow[r, "q_0"] \arrow[d, "q_1"'] & o \arrow[r, "p_0"] \arrow[d, "p_2"] & o_0 \arrow[d, "h"] \\
o_1 \arrow[r, "g_1"'] & o_2 \arrow[r, "g_2"'] & o_3
\end{tikzcd}
$$

Figure 12. Composition of pullbacks.

Of course, it might well be the case that no such arrows ψ and ψ' exist, e.g., when one tries to solve the sets $\{f(x) = f'(y)\}$ or $\{f(x) = x\}$ for unary operation symbols f and f'. There can also exist more solutions than one, and this is always the case as the names of the variables in U are not important at all.

The different flavors obtained by moving from equalizers to pullbacks and vice versa can be better understood by reasoning in terms of two equivalent but yet opposite views. The first view, corresponding to the equalizer construction, assumes that one is working with all variables at his/her disposal, i.e., not only one consider $X \cup Y$, but really taking into account the universe $\mathcal{X}$ of *all* variables. The second view is completely decentralized (as far as it can) and takes control only over the variables of interest for a particular mgu construction.

3 The double category of pullbacks

In this section we show that the construction of pullbacks in the category of substitutions can be presented in a modular way, by composing together a finite set of basic pullbacks. We start by showing that the pullback squares in a category $\mathcal{C}$ form a double category. To see this, let us remind a few classical results.

Proposition 2
Given a category $\mathcal{C}$ and three arrows $h\colon o_0 \to o_3$, $g_1\colon o_1 \to o_2$ and $g_2\colon o_2 \to o_3$, let (o, p_0, p_2) be the pullback of h and g_2, and let (o', q_0, q_1) be the pullback of p_2 and g_1 (see Figure 12). Then, $(o', q_0; p_0, q_1)$ is a pullback for h and $g_1; g_2$.

Proposition 3
Given a category $\mathcal{C}$ and three arrows $h\colon o_0 \to o_3$, $g_1\colon o_1 \to o_2$ and $g_2\colon o_2 \to o_3$, let (o, p_0, p_2) be the pullback of h and g_2, and let $(o', q_0; p_0, q_1)$ be the pullback for h and $g_1; g_2$. Then, (o', q_0, q_1) is a pullback of p_2 and g_1.

$$\begin{tikzcd} o \arrow[r, "p_0"] \arrow[d, "p_1"'] & o_0 \arrow[d, "h"] \\ o_1 \arrow[r, "g"'] & o_2 \end{tikzcd}$$

Figure 13. Pullback square as double cell.

Proposition 4
The pullback of $h: o_0 \to o_1$ and $id_{o_1}: o_1 \to o_1$ exists in any category. Moreover, (o_0, id_{o_0}, h) is a pullback for h and id_{o_1}.

Definition 3.1 (*Double category of pullbacks*)
Given a category $\mathcal{C}$, the double category of pullbacks in $\mathcal{C}$, denoted by $\mathscr{P}(\mathcal{C})$, is defined as follows:

- its objects are the objects of $\mathcal{C}$;
- its horizontal 1-category is $\mathcal{C}$;
- its vertical 1-category is $\mathcal{C}$;
- the double cells are the squares (p_0, p_1, h, g) (see Figure 13) such that p_0 and p_1 define a pullback for h and g;
- given two double cells (q, q', p', g_1) and (p, p', h, g_2) their horizontal composition is the double cell $(q;p, q', h, g_1;g_2)$;
- given two double cells (q, q', h_1, p) and (p, p', h_2, g) their vertical composition is the double cell $(q, q';p', h_1;h_2, g)$.

To see that $\mathscr{P}(\mathcal{C})$ is indeed a double category, observe that both horizontal and vertical compositions of double cells return pullback squares (Proposition 2). Moreover, the trivial pullback squares $(id_{o_0}, p, p, id_{o_1})$ and $(p, id_{o_0}, id_{o_1}, p)$ behave as identities w.r.t. the horizontal and vertical composition, respectively. The exchange law of double categories holds trivially, as the cells are completely identified by their borders.

For the arguments presented in the previous section, it follows that pullbacks are a fundamental ingredient in the operational semantics, as they provide a characterization of the mgu construction, which clearly separates the goal dimension (horizontal) from the resolution mechanism (the vertical dimension) focusing on their interaction (the substitutions yielding the pullback).

However, dealing with all this machinery at the computational level is too heavy, as there are infinitely many pullbacks. Therefore, a finitary presentation of $\mathscr{P}(\mathcal{C})$ is a main issue.

3.1 Finitary presentation of pullbacks

Our first contribution consists of recovering in a finitary way the double category $\mathscr{P}(\mathbf{Th}[\Sigma])$. We start by focusing on the small set of commuting squares depicted in Figure 14. We want to show that any pullback can then be obtained by composing these basic squares (and horizontal and vertical identity pullbacks).

There are a few points for which some explanation is worth. First of all, note that we have depicted the pullback cells with the direction of the arrows reversed with respect

$$\begin{tikzcd} \underline{1} & \underline{n} \arrow[l, "f"'] \arrow[dl, phantom, "R_f"] \\ \underline{n} \arrow[u, "f"] & \underline{n} \arrow[u, "id_{\underline{n}}"'] \arrow[l, "id_{\underline{n}}"] \end{tikzcd} \qquad \begin{tikzcd} \underline{2} & \underline{n}+1 \arrow[l, "f\otimes id_{\underline{1}}"'] \\ {} \arrow[r, phantom, "D_f"] & \underline{2n} \arrow[u, "id_{\underline{n}}\otimes f"'] \\ \underline{1} \arrow[uu, "\nabla_{\underline{1}}"] & \underline{n} \arrow[u, "\nabla_{\underline{n}}"'] \arrow[l, "f"] \end{tikzcd} \qquad \begin{tikzcd} \underline{2} & & \underline{1} \arrow[ll, "\nabla_{\underline{1}}"'] \\ \underline{n}+1 \arrow[u, "f\otimes id_{\underline{1}}"] & \underline{2n} \arrow[l, "id_{\underline{n}}\otimes f"] \arrow[u, phantom, "\hat{D}_f"] & \underline{n} \arrow[u, "f"'] \arrow[l, "\nabla_{\underline{n}}"] \end{tikzcd} \quad (\text{for all } f \in \Sigma^n)$$

$$\begin{tikzcd} \underline{2} & \underline{1} \arrow[l, "\nabla_{\underline{1}}"'] \arrow[dl, phantom, "R_\nabla"] \\ \underline{1} \arrow[u, "\nabla_{\underline{1}}"] & \underline{1} \arrow[u, "id_{\underline{1}}"'] \arrow[l, "id_{\underline{1}}"] \end{tikzcd} \qquad \begin{tikzcd} \underline{2} & \underline{2} \arrow[l, "\gamma_{\underline{1},\underline{1}}"'] \arrow[dl, phantom, "R_\gamma"] \\ \underline{2} \arrow[u, "\gamma_{\underline{1},\underline{1}}"] & \underline{2} \arrow[u, "id_{\underline{2}}"'] \arrow[l, "id_{\underline{2}}"] \end{tikzcd} \qquad \begin{tikzcd} \underline{3} & \underline{2} \arrow[l, "\nabla_{\underline{1}}\otimes id_{\underline{1}}"'] \arrow[dl, phantom, "D_\nabla"] \\ \underline{2} \arrow[u, "id_{\underline{1}}\otimes\nabla_{\underline{1}}"] & \underline{1} \arrow[u, "\nabla_{\underline{1}}"'] \arrow[l, "\nabla_{\underline{1}}"] \end{tikzcd}$$

Figure 14. The basic pullbacks.

$$\begin{tikzcd} \underline{0} & \underline{1} \arrow[l, "!_{\underline{1}}"'] \arrow[dl, phantom, "R_!"] \\ \underline{1} \arrow[u, "!_{\underline{1}}"] & \underline{1} \arrow[u, "id_{\underline{1}}"'] \arrow[l, "id_{\underline{1}}"] \end{tikzcd} \qquad \begin{tikzcd} \underline{3} & \underline{2} \arrow[l, "id_{\underline{1}}\otimes\nabla_{\underline{1}}"'] \arrow[dl, phantom, "\hat{D}_\nabla"] \\ \underline{2} \arrow[u, "\nabla_{\underline{1}}\otimes id_{\underline{1}}"] & \underline{1} \arrow[u, "\nabla_{\underline{1}}"'] \arrow[l, "\nabla_{\underline{1}}"] \end{tikzcd} \qquad \begin{tikzcd} \underline{2} & & \underline{1} \arrow[ll, "\nabla_{\underline{1}}"'] \\ \underline{1}+\underline{n} \arrow[u, "id_{\underline{1}}\otimes f"] & \underline{2n} \arrow[l, "f\otimes id_{\underline{n}}"] \arrow[u, phantom, "\hat{D}'_f"] & \underline{n} \arrow[u, "f"'] \arrow[l, "\nabla_{\underline{n}}"] \end{tikzcd}$$

Figure 15. **Q**: Are these three basic cells missing? **A**:No.

to the usual presentation. The reason for this will become much clearer in Section 4, where we will show that this representation matches the intuitive direction of computation flow (from up to down) and also of contextualization of a program (that compose to the right; but remember that we deal with internal contextualization that would otherwise add contexts to the left of current states). From the point of view of the notation this is not problematic in tile logic, as we can assume to work with opposite categories of configuration and observations. For sake of clarity, the tiles in Figure 14 can be written as the sequents $R_f : f \xrightarrow[id_{\underline{n}}]{f} id_{\underline{n}}$, $D_f : f \otimes id_{\underline{1}} \xrightarrow[\nabla_{\underline{1}};(id_{\underline{1}}\otimes f)]{\nabla_{\underline{1}}} f$, $\hat{D}_f : \nabla_{\underline{1}} \xrightarrow[f]{f\otimes id_{\underline{1}}} \nabla_{\underline{1}};(id_{\underline{1}}\otimes f)$, $R_\nabla : \nabla_{\underline{1}} \xrightarrow[id_{\underline{1}}]{\nabla_{\underline{1}}} id_{\underline{1}}$, $R_\gamma : \gamma_{\underline{1},\underline{1}} \xrightarrow[id_{\underline{2}}]{\gamma_{\underline{1},\underline{1}}} id_{\underline{2}}$, and $D_\nabla : \nabla_{\underline{1}} \otimes id_{\underline{1}} \xrightarrow[\nabla_{\underline{1}}]{id_{\underline{1}}\otimes\nabla_{\underline{1}}} \nabla_{\underline{1}}$.

The second point to notice is that the cells R_∇, R_ρ, and D_∇ do not depend on Σ, i.e., they form in some sense the intrinsic auxiliary structure of the pullback construction.

Finally there are some cells that one might expect to see in Figure 14 but are instead missing. Trying to guess the intuition of the reader, we have listed some of them in Figure 15.

The first cell we consider is $R_!$. Its absence might be surprising, because there are analogous cells for all the other basic constructors (operators $f \in \Sigma$, symmetries and duplicators). But $R_!$ is not a pullback. In fact the pullback of $!_{\underline{1}}$ and $!_{\underline{1}}$ is $(\underline{2}, id_{\underline{1}}\otimes !_{\underline{1}}, !_{\underline{1}} \otimes id_{\underline{1}})$, yielding

$$\begin{tikzcd}[column sep=large]
\underline{3} \arrow[dr, phantom, "A_1"] & \underline{3} \arrow[l, "(\gamma_{\underline{1},\underline{1}}\otimes id_{\underline{1}});\gamma_{\underline{2},\underline{1}}"'] \arrow[dr, phantom, "A_2"] & \underline{3} \arrow[l, "\gamma_{\underline{1},\underline{1}}\otimes id_{\underline{1}}"'] \arrow[drr, phantom, "A_3"] & \underline{3} \arrow[l, "\gamma_{\underline{1},\underline{2}}"'] & \underline{2} \arrow[l, "id_{\underline{1}}\otimes\nabla_{\underline{1}}"'] \\
\underline{3} \arrow[u, "(id_{\underline{1}}\otimes\gamma_{\underline{1},\underline{1}});\gamma_{\underline{1},\underline{2}}"] \arrow[dr, phantom, "A_4"] & \underline{3} \arrow[l, "id_{\underline{3}}" description] \arrow[u, "id_{\underline{3}}" description] \arrow[ddrr, phantom, "A_5"] & \underline{3} \arrow[l, "\gamma_{\underline{1},\underline{1}}\otimes id_{\underline{1}}" description] \arrow[u, "id_{\underline{3}}" description] & \underline{2} \arrow[l, "\nabla_{\underline{1}}\otimes id_{\underline{1}}" description] \arrow[dr, phantom, "A_6"] & \underline{2} \arrow[l, "\gamma_{\underline{1},\underline{1}}" description] \arrow[u, "id_{\underline{2}}"'] \\
\underline{3} \arrow[u, "id_{\underline{1}}\otimes\gamma_{\underline{1},\underline{1}}"] \arrow[ddr, phantom, "A_7"] & \underline{3} \arrow[l, "id_{\underline{3}}" description] \arrow[u, "id_{\underline{1}}\otimes\gamma_{\underline{1},\underline{1}}" description] & & \underline{2} \arrow[u, "\gamma_{\underline{1},\underline{1}}" description] \arrow[dr, phantom, "A_8"] & \underline{2} \arrow[l, "id_{\underline{2}}" description] \arrow[u, "id_{\underline{2}}"'] \\
\underline{3} \arrow[u, "\gamma_{\underline{2},\underline{1}}"] & \underline{2} \arrow[u, "id_{\underline{1}}\otimes\nabla_{\underline{1}}" description] \arrow[dr, phantom, "A_9"] & \underline{2} \arrow[l, "\gamma_{\underline{1},\underline{1}}" description] \arrow[dr, phantom, "A_{10}"] & \underline{1} \arrow[l, "\nabla_{\underline{1}}" description] \arrow[u, "\nabla_{\underline{1}}" description] \arrow[dr, phantom, "A_{11}"] & \underline{1} \arrow[l, "id_{\underline{1}}" description] \arrow[u, "\nabla_{\underline{1}}"'] \\
\underline{2} \arrow[u, "\nabla_{\underline{1}}\otimes id_{\underline{1}}"] & \underline{2} \arrow[l, "id_{\underline{2}}"] \arrow[u, "\gamma_{\underline{1},\underline{1}}" description] & \underline{2} \arrow[l, "id_{\underline{2}}"] \arrow[u, "id_{\underline{2}}" description] & \underline{1} \arrow[l, "\nabla_{\underline{1}}"] \arrow[u, "id_{\underline{1}}" description] & \underline{1} \arrow[l, "id_{\underline{1}}"] \arrow[u, "id_{\underline{1}}"']
\end{tikzcd}$$

Figure 16. How to compose the cell $\hat{D}_{\nabla}$.

a cell that can in fact be obtained by composing in parallel the horizontal and vertical identities of $!_{\underline{1}}$ (i.e., by putting $id_{\underline{0}} \xrightarrow[!_{\underline{1}}]{!_{\underline{1}}} id_{\underline{1}}$ in parallel with $!_{\underline{1}} \xrightarrow[id_{\underline{1}}]{id_{\underline{0}}} !_{\underline{1}}$).

The second cell $\hat{D}_{\nabla}$ defines a pullback, but it can be obtained by composition of other basic cells and identities, as illustrated in Figure 16. Let us comment on the composition. At the centre of the figure we find the cell $A_5 = D_{\nabla}$, in fact we remind that by the coherence axioms for duplicators (cf. Table 3) we have $\nabla_{\underline{1}};\gamma_{\underline{1},\underline{1}} = \nabla_{\underline{1}}$, and by functoriality of tensor product we have e.g.,

$$(id_{\underline{1}} \otimes \nabla_{\underline{1}});(id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}}) = (id_{\underline{1}};id_{\underline{1}}) \otimes (\nabla_{\underline{1}};\gamma_{\underline{1},\underline{1}}) = id_{\underline{1}} \otimes \nabla_{\underline{1}}.$$

On the bottom-right part of the figure, we find the cells A_8 and A_{10} that are the horizontal and vertical identities of $\nabla_{\underline{1}}$, while A_{11} is the trivial identity for the object $\underline{1}$. Then, note that $A_6 = A_9 = R_{\gamma}$. The cell A_7 is an horizontal identity, in fact by naturality of the symmetries, we have

$$\gamma_{\underline{1},\underline{1}};(id_{\underline{1}} \otimes \nabla_{\underline{1}}) = (\nabla_{\underline{1}} \otimes id_{\underline{1}});\gamma_{\underline{2},\underline{1}}.$$

Likewise, the cell A_3 is a vertical identity. Also A_2 and A_4 are obvious identities. The tile A_1 deserves more attention. The first thing to note is that by naturality of symmetries we have that

$$(\gamma_{\underline{1},\underline{1}} \otimes id_{\underline{1}});\gamma_{\underline{2},\underline{1}} = \gamma_{\underline{2},\underline{1}};(id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}})$$

and then, by coherence of symmetries, we have

$$\begin{array}{rcl}
\gamma_{\underline{2},\underline{1}} & = & (id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}});(\gamma_{\underline{1},\underline{1}} \otimes id_{\underline{1}}) \\
\gamma_{\underline{1},\underline{2}} & = & (\gamma_{\underline{1},\underline{1}} \otimes id_{\underline{1}});(id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}})
\end{array}$$

and therefore it follows that

$$\gamma_{\underline{2},\underline{1}};(id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}}) = (id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}});(\gamma_{\underline{1},\underline{1}} \otimes id_{\underline{1}});(id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}}) = (id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}});\gamma_{\underline{1},\underline{2}}.$$

$$
\begin{tikzcd}
{} & {} \arrow[l] & {} \arrow[l] & {} \arrow[l] \\
{} \arrow[u] \arrow[ur, phantom, "A_{11}\otimes R_\gamma"] & {} \arrow[l] \arrow[u] \arrow[ur, phantom, "A_2"] & {} \arrow[l] \arrow[u] \arrow[ur, phantom, "A_2'"] & {} \arrow[l] \arrow[u] \\
{} \arrow[u] \arrow[ur, phantom, "A_4'"] & {} \arrow[l] \arrow[u] \arrow[ur, phantom, "R_\gamma\otimes A_{11}"] & {} \arrow[l] \arrow[u] \arrow[ur, phantom, "A_2'"] & {} \arrow[l] \arrow[u] \\
{} \arrow[u] \arrow[ur, phantom, "A_4"] & {} \arrow[l] \arrow[u] \arrow[ur, phantom, "A_4"] & {} \arrow[l] \arrow[u] \arrow[ur, phantom, "A_{11}\otimes R_\gamma"] & {} \arrow[l] \arrow[u]
\end{tikzcd}
$$

Figure 17. How to obtain the cell A_1 in Figure 16.

$$
\begin{tikzcd}
\underline{2} \arrow[dr, phantom, "R_\gamma"] & \underline{2} \arrow[l, "\gamma_{\underline{1},\underline{1}}"'] & \underline{2} \arrow[l, "\gamma_{\underline{1},\underline{1}}"'] & & \underline{1} \arrow[ll, "\nabla_{\underline{1}}"'] \\
\underline{2} \arrow[u, "\gamma_{\underline{1},\underline{1}}"] & \underline{2} \arrow[l, "id_{\underline{2}}"] \arrow[u, "id_{\underline{2}}"] & \underline{2} \arrow[l, "\gamma_{\underline{1},\underline{1}}"] \arrow[u, "id_{\underline{2}}"] \arrow[dr, phantom, "\hat{D}_f"] & & \underline{1} \arrow[ll, "\nabla_{\underline{1}}"] \arrow[u, "id_{\underline{1}}"'] \\
\underline{2} \arrow[u, "\gamma_{\underline{1},\underline{1}}"] \arrow[dr, phantom, "B_3"] & \underline{n+1} \arrow[u, "f\otimes id_{\underline{1}}"] \arrow[drr, phantom, "B_2"] & \underline{2n} \arrow[l, "id_{\underline{n}}\otimes f"] & \underline{2n} \arrow[l, "\gamma_{\underline{n},\underline{n}}"] & \underline{n} \arrow[l, "\nabla_{\underline{n}}"] \arrow[u, "f"'] \\
 & \underline{n+1} \arrow[l, phantom] \arrow[u, "id_{\underline{n+1}}"] \arrow[dr, phantom, "B_1"] & \underline{1+n} \arrow[l, "\gamma_{\underline{1},\underline{n}}"] & \underline{2n} \arrow[l, "f\otimes id_{\underline{n}}"] \arrow[u, "id_{\underline{2n}}"] & \underline{n} \arrow[l, "\nabla_{\underline{n}}"] \arrow[u, "id_{\underline{n}}"] \\
\underline{1+n} \arrow[uu, "id_{\underline{1}}\otimes f"] & \underline{1+n} \arrow[l, "id_{\underline{1+n}}"'] \arrow[u, "\gamma_{\underline{1},\underline{n}}"] & \underline{1+n} \arrow[l, "id_{\underline{1+n}}"'] \arrow[u, "id_{\underline{1+n}}"] & \underline{2n} \arrow[l, "f\otimes id_{\underline{n}}"'] \arrow[u, "id_{\underline{2n}}"] & \underline{n} \arrow[l, "\nabla_{\underline{n}}"] \arrow[u, "id_{\underline{n}}"]
\end{tikzcd}
$$

Figure 18. How to compose the cell $\hat{D}_f'$.

We can thus construct A_1 as illustrated in Figure 17, where A_2' is the vertical identity of $id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}}$ and A_4' is the horizontal identity of $\gamma_{\underline{1},\underline{1}} \otimes id_{\underline{1}}$ (for simplicity we omit to specify the borders of the cells, as they should be evident from the discussion above). To conclude that the composition in Figure 16 yields $\hat{D}_\nabla$ we have to check that their borders are equal, and in fact observe that

$$
\begin{aligned}
\gamma_{\underline{1},\underline{2}};(\gamma_{\underline{1},\underline{1}} \otimes id_{\underline{1}});(\gamma_{\underline{1},\underline{1}} \otimes id_{\underline{1}});\gamma_{\underline{2},\underline{1}} &= \gamma_{\underline{1},\underline{2}};\gamma_{\underline{2},\underline{1}} = id_{\underline{3}} \\
\gamma_{\underline{2},\underline{1}};(id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}});(id_{\underline{1}} \otimes \gamma_{\underline{1},\underline{1}});\gamma_{\underline{1},\underline{2}} &= \gamma_{\underline{2},\underline{1}};\gamma_{\underline{1},\underline{2}} = id_{\underline{3}}.
\end{aligned}
$$

The third cell $\hat{D}_f'$ illustrated in Figure 15 is a pullback, but it can be composed starting from $\hat{D}_f$ as shown in Figure 18, where unnamed cells are obvious (horizontal or vertical) identities. In writing the border of $\hat{D}_f$ we have exploited the coherence axiom

$$\nabla_{\underline{n}};\gamma_{\underline{n},\underline{n}} = \nabla_{\underline{n}}.$$

The cells B_3 and B_2 are identities that exploits the naturality of symmetries. Finally, the cell B_1 is obtained by a construction analogous to that of A_1, employing R_γ as a building block.

We hope that the few examples above can help the reader in understanding the compositional mechanism of basic cells, as it will be especially useful in Section 4.

For instance, we can state a few technical lemmata that can be proved by tile pastings similar to the cell compositions discussed above.

Lemma 1
Given any arrow $t\colon \underline{n} \to \underline{m} \in \mathbf{Th}[\Sigma]$ that can be obtained without using dischargers, the cell $t \xrightarrow[id_{\underline{n}}]{t} id_{\underline{n}}$ can be obtained by composition of basic cells.

Note that by adding the cells R_f just for the operators of the signature, then we are able to construct the analogous cells for all contexts t.

Lemma 2
Given any arrows $t\colon \underline{h} \to \underline{m}$ and $s\colon \underline{m} \to \underline{n}$ in $\mathbf{Th}[\Sigma]$, the cells $\gamma_{\underline{n},\underline{k}} \xrightarrow[s\otimes t]{t\otimes s} \gamma_{\underline{m},\underline{h}}$ and $t \otimes s \xrightarrow[\gamma_{\underline{m},\underline{h}}]{\gamma_{\underline{n},\underline{k}}} s \otimes t$ can be obtained by composition of basic cells.

Lemma 3
Given any arrow $t\colon \underline{m} \to \underline{n} \in \mathbf{Th}[\Sigma]$, the cells $\nabla_{\underline{n}} \xrightarrow[t]{t\otimes t} \nabla_{\underline{m}}$ and $t \otimes t \xrightarrow[\nabla_{\underline{m}}]{\nabla_{\underline{n}}} t$ can be obtained by composition of basic cells.

Theorem 1
Any pullback can be obtained as the horizontal, vertical and parallel composition of the pullbacks in Figure 14 together with horizontal and vertical identities.

Proof
The fact that all composed cells are pullbacks is straightforward, as all basic tiles are pullbacks and such a property is preserved by the three operations of the tile model (horizontal and vertical sequential compositions and parallel composition).

The proof that all pullbacks can be obtained in this way is more subtle. We exploit the fact that, in the category $\mathbf{Th}[\Sigma]$, whenever the pullback of σ and θ exists and σ can be decomposed as $\sigma_1;\sigma_2$, then also the pullback of σ_2 and θ exists (because σ_2 is less instantiated than σ). The proof proceeds by observing that each arrow σ in $\mathbf{Th}[\Sigma]$ can be finitely decomposed as $\sigma_1;\sigma_2;\cdots;\sigma_n$ where $\sigma_i = id_{\underline{k_i}} \otimes t_i \otimes id_{\underline{m_i}}$, with $t_i \in \Sigma \cup \{\gamma_{\underline{1},\underline{1}}, \nabla_{\underline{1}}, !_{\underline{1}}, id_{\underline{1}}\}$. Then, the pullback of θ and σ, if it exists, can be computed stepwise, and this is the task accomplished by the basic cells in Figure 14 that cover all possible cases. In fact, the proof is by induction on the length n of the decomposition of σ. $\square$

3.2 Pullbacks as tiles

The finitary presentation of pullbacks can be straightforwardly used to build a tile system that generates the double category of pullbacks.

Definition 3.2 (*Tile system for pullbacks*)
Given a signature Σ, we define the tile system $\mathcal{R}_{PB(\Sigma)}$ such that its horizontal category is $\mathbf{Th}[\Sigma]^{\mathrm{op}}$, its vertical category is $\mathbf{Th}[\Sigma]^{\mathrm{op}}$ and the basic cells are those in Figure 14 (remember that horizontal and vertical identity tiles will be freely generated in the model).

The representation theorem can then be rephrased as below.

Theorem 2
A cell $t \xrightarrow[v]{u} s$ is in $\mathscr{P}(\mathcal{C})$ if and only if $\mathcal{R}_{PB(\Sigma)} \vdash t \xrightarrow[v]{u} s$.

4 Tile systems for logic programs

The idea is to transform a logic program into a tile system which is able to compute the same computed answer substitutions for each goal. To this aim, we will exploit the tiles presented for building pullbacks in the category of substitutions, which provide the unification mechanism for goal resolution.

4.1 From logic programs to a logic of tiles

The tile system that we propose can be sketched as follows:

Definition 4.1 (*Tile system for logic programming*)
Given a pure logic program $\mathcal{P}$ on the alphabet Σ, we denote by $\mathcal{R}_{\mathcal{P}}$ the tile system specified by the following rules:

- There are two basic sorts t (for terms) and p (for predicates). Correspondingly, the interfaces are elements of $\{\mathsf{t},\mathsf{p}\}^*$ (as a matter of notation, we let ε denote the empty string of sorts, and denote by t^n the string composed by n occurrences of t, and similarly for p).
- To each functional symbol f with arity n in the alphabet, we associate an operator $f\colon\mathsf{t}^n \to \mathsf{t}$ in the signature of configurations, and to each predicate symbol p (here $\Box$ can be viewed as a nullary predicate) with arity k in the alphabet, we associate an operator $p\colon\mathsf{t}^k \to \mathsf{p}$ in the signature of configurations. Then, we add the symbol $\_\wedge\_\colon\mathsf{p}^2 \to \mathsf{p}$ for modeling conjunction. We will show that, without loss of generality, the conjunction operator can be assumed to be associative and with unit $\Box$. The configurations are the arrows (of the op-category) of the free algebra generated by the signature of configurations.
- To each functional symbol f with arity n in the alphabet, we associate an operator $f\colon\mathsf{t}^n \to \mathsf{t}$ in the signature of observations (note that the symbol f is thus overloaded, since it also appears in the horizontal dimension, however, this will not create any confusion). Then, the observations are the arrows (of the op-category) of the free cartesian category generated by the signature of observations.
- To each clause

$$c \equiv p(t_1,\ldots,t_k)\ :-\ q_1(\vec{s}_1),\ldots,q_m(\vec{s}_m)$$

(over n variables $\{x_1,\ldots,x_n\}$) in the logic program we associate a basic tile T_c in our system whose initial configuration is $p\colon\mathsf{t}^k \to \mathsf{p}$ (representing the predicate symbol in the head of c), whose final configuration is $q_1(\vec{s}_1)\wedge\ldots\wedge q_m(\vec{s}_m)\colon\mathsf{t}^n \to \mathsf{p}$ (representing the body of the clause), whose trigger is the identity $id_{\mathsf{p}}\colon\mathsf{p}\to\mathsf{p}$ and whose effect is the tuple $\langle t_1,\ldots,t_k\rangle\colon\mathsf{t}^n \to \mathsf{t}^k$ (representing the pattern to be matched by the arguments of predicate p). Note that the body of the clause may contain variables not appearing in the head (i.e., some of the x_i's might not appear in the t_i's) and consequently some discharger will be used in the effect of the tile. Moreover, since the same variable can be used more than once, duplicators can also be necessary (this is to remark that the tupling $\langle t_1,\ldots,t_k\rangle$ is different from $t_1\otimes\cdots\otimes t_k$, as the former involves the sharing of variables). Since we take the op-categories the

$$\begin{tikzcd}
\mathsf{p} & \mathsf{t}^k \arrow[l, "p"'] \\
\mathsf{p} \arrow[u, "id"] \arrow[ur, phantom, "T_c"] & \mathsf{t}^n \arrow[l, "q_1(\vec{s}_1)\wedge\ldots\wedge q_m(\vec{s}_m)"] \arrow[u, "\langle t_1,\ldots,t_k\rangle"']
\end{tikzcd}$$

Figure 19

direction of all arrows is reversed w.r.t. the standard representation (see the tile T_c in Figure 19).

- Finally, we add the basic tiles contained in $\mathcal{R}_{PB(\Sigma)}$ (see Figure 14) for building pullbacks. Remind that just three of them depend on the alphabet under consideration, while the other three are common to all programs, i.e., they can be considered auxiliary to the framework.

Remark 4.1
When it is obvious from the context, we shall abuse the notation by avoiding to specify the involved sorts in the subscripts of id, ∇, γ and $!$, writing just the numbers of involved arguments (e.g., instead of $\gamma_{\mathsf{tp},\mathsf{p}}$ we shall write $\gamma_{\underline{2},\underline{1}}\colon \mathsf{tpp} \to \mathsf{ptp}$).

We can assume the operator $\wedge$ to be associative and with unit $\Box$ because all the basic tiles associated to the clauses have an identity as trigger. This, together with the fact that they are the only rewrite rules involving predicate symbols, means that rewrites are always enabled for predicates nested in conjunctions. For example in the expression $q_1(\vec{s}_1)\wedge\ldots\wedge q_m(\vec{s}_m)$ it is not important the way in which the q_i's are conjoined, as their evolutions do not interact with the 'tree' of conjunctions. Thus, $q_0\wedge(q_1\wedge q_2)$ is equivalent to $(q_0\wedge q_1)\wedge q_2$. Since the are no rules for the $\Box$ operator, we make it the unit for $\wedge$. These assumptions do not alter the 'behaviour' of the system, but allow us to simplify the notation and the presentation of main results.

The intuition is that for each goal, we can compute the answer in the tile system by starting from the associated configuration and considering the effects of each tile that ends in a configuration of the form $\Box\otimes !_{\underline{n}}$ (without monoidality of $\wedge$ we should have considered as final configurations for termination all the possible finite conjunctions of empty goals). The discharger $!_{\underline{n}}$ keeps track of the fresh variables that have been introduced in the computer answer substitution. The tiles with initial input interface p and final configuration $\Box\otimes !_{\underline{n}}$ for $n\in\mathbb{N}$ are called *refutation tiles*.

The following example should illustrate how the tile system can simulate logic programming computations.

Example 4.1
Let us consider the simple alphabet consisting of constants a and b, unary function symbol f, unary predicate q and binary predicates p and r.

Given the logic program $\mathcal{P}$ defined by the three clauses

$c_1 \equiv$ `p(f(X1),X2) :-  q(X1), r(X1,X2) .`
$c_2 \equiv$ `r(a,a) .`
$c_3 \equiv$ `q(b) .`

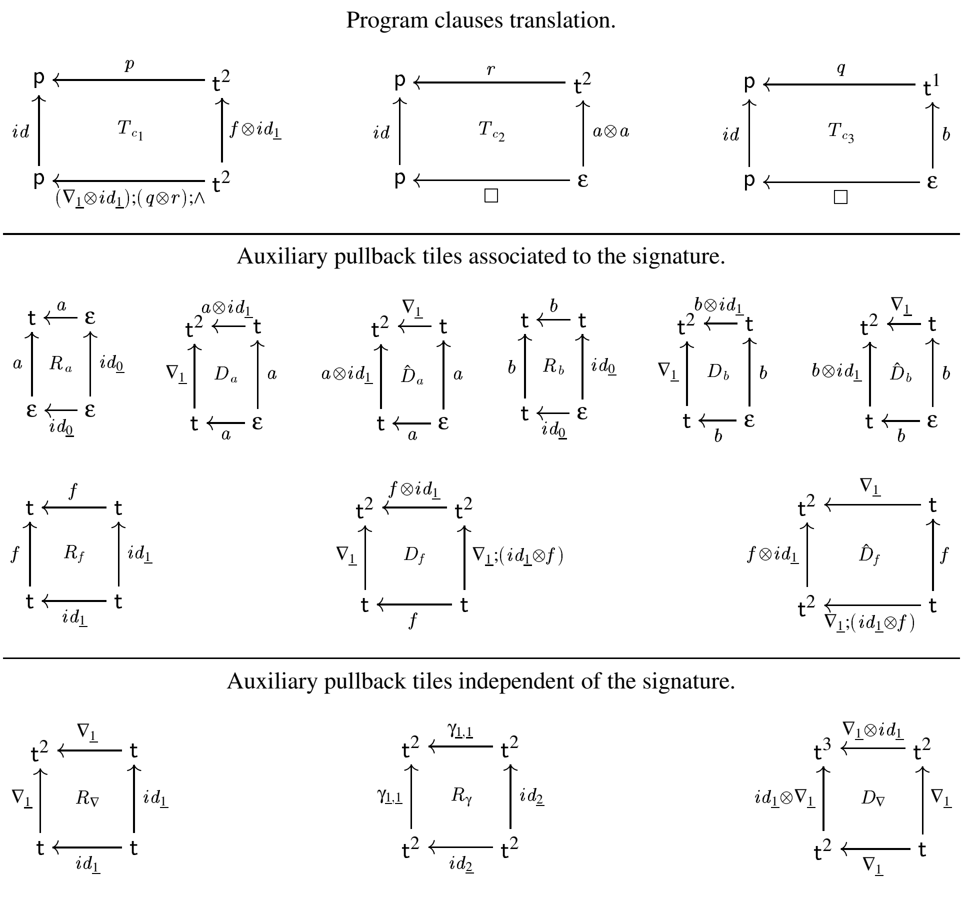

Figure 20. The tile system associated to the logic program $\mathcal{P}$.

the corresponding tile system is illustrated in Figure 20. The tiles in the first row are those associated to the three clauses of the program. The tiles in the second row are the basic pullbacks associated to the constants a, b of the alphabet, while the tiles in the third row are the basic pullbacks associated to the unary function symbol f of the alphabet. The tiles in the fourth row are the auxiliary tiles common to all representations of logic programs.

Now suppose one wants to compute the goal $?-\ p(x_1, x_2)$. The idea is to compute all possible tiles that have $p\colon \mathsf{t}^2 \to \mathsf{p}$ as initial configuration and the empty goal as conclusion. The effect of such tiles should in fact correspond to the computed answer substitutions of the program execution on the given goal. It is easy to argue that no such tile exists for the given goal. In fact, the only tile having p as initial configuration is T_{c_1} that leads to the configuration $(\nabla_{\underline{1}} \otimes id_{\underline{1}});(q \otimes r);\wedge$. Then T_{c_3} and T_{c_2} can be (concurrently) applied respectively to q and to r, but the computation cannot be completed, as the coordination of the two resolutions is not possible. In fact the pullback of $b \otimes a$ and $\nabla_{\underline{1}}$ does not exist, and hence also the pullback of $b \otimes a \otimes a$ and $(\nabla_{\underline{1}} \otimes id_{\underline{1}})$ does not exist as well. The partial computation is illustrated in Figure 21.

If the third clause c_3 is replaced by $c_3' \equiv \mathtt{q(a)}$, then we can compute the tile refutation

$$
\begin{tikzcd}
\mathsf{p} \arrow[rrrd, phantom, "T_{c_1}"] & & & \mathsf{t}^2 \arrow[lll, "p"'] \\
\mathsf{p} \arrow[u, "id"] & \mathsf{p}^2 \arrow[l, "\wedge"] \arrow[rd, phantom, "T_{c_3}\otimes T_{c_2}"] & \mathsf{t}^3 \arrow[l, "q\otimes r"] & \mathsf{t}^2 \arrow[l, "\nabla_{\underline{1}}\otimes id_{\underline{1}}"] \arrow[u, "f\otimes id_{\underline{1}}"'] \\
\mathsf{p} \arrow[u, "id"] & \mathsf{p}^2 \arrow[l, "\wedge"] \arrow[u, "id"] & \varepsilon \arrow[l, "\square\otimes\square"] \arrow[u, "b\otimes a\otimes a"'] &
\end{tikzcd}
$$

Figure 21. The incomplete derivation for p.

$$
\begin{tikzcd}
\mathsf{p} \arrow[rrrd, phantom, "T_{c_1}"] & & & \mathsf{t}^2 \arrow[lll, "p"'] \\
\mathsf{p} \arrow[u, "id"] & \mathsf{p}^2 \arrow[l, "\wedge"] \arrow[rd, phantom, "T_{c'_3}\otimes T_{c_2}"] & \mathsf{t}^3 \arrow[l, "q\otimes r"] \arrow[rd, phantom, "\alpha\otimes 1_a"] & \mathsf{t}^2 \arrow[l, "(\nabla_{\underline{1}}\otimes id_{\underline{1}})"] \arrow[u, "f\otimes id_{\underline{1}}"'] \\
\mathsf{p} \arrow[u, "id"] & \mathsf{p}^2 \arrow[l, "\wedge"] \arrow[u, "id"] & \varepsilon \arrow[l, "\square\otimes\square"] \arrow[u, "a\otimes a\otimes a"] & \varepsilon \arrow[l, "id_{\underline{0}}"] \arrow[u, "a\otimes a"']
\end{tikzcd}
$$

Figure 22. The refutation for p.

illustrated in Figure 22, where the tile $\alpha\colon \nabla_{\underline{1}} \xrightarrow[a]{a\otimes a} id_{\underline{0}}$ can be obtained in any of the two ways illustrated in Figure 23 (remind Lemma 3). The computed answer substitution $f(a) \otimes a$ (representing $[f(a)/x_1, a/x_2]$) is given by the effect of the composed tile. Note that c'_3 and c_2 can be applied concurrently, i.e., the order in which they are applied is not relevant and moreover, they can also be performed in parallel, their outputs being coordinated by means of the tile α. The two ways of building α show that the coordination mechanism does not depend on the order of execution of $T_{c'_3}$ and T_{c_2}, which is in fact immaterial.

Notice that the use of tiles, thanks to its abstract flavour, completely frees the user from managing fresh variables, taking care in an automatic way of all the problems connected to name handling via the use of local placeholders.

4.2 From clauses to tiles

We try to explain here informally the intuition that lies behind the definition of $\mathcal{R}_{\mathcal{P}}$. Basically, it is strictly related to the idea of building an LTS out of a reduction system to study the interactions between composed components. From one point of view, it is evident that the reduction system of goal resolution summarized in Section 1.2 considers the whole goal as an atomic entity, whose parts must all be coordinated. From this point of view, the clauses define the basic reduction steps that can be conveniently instantiated and contextualized. Indeed, the reduction perspective of logic programs has been investigated in (Corradini & Montanari, 1992). However, to accomplish this view, one usually assumes to start with a set of variables large enough to contain all names that will be needed by all clause instances used in the refutation, as their dynamic creation cannot be modeled.

$$
\begin{tikzcd}
\mathsf{t}^2 & \mathsf{t} \arrow[l, "\nabla_{\underline{1}}"'] \\
\mathsf{t} \arrow[u, "a\otimes id_{\underline{1}}"] \arrow[ur, phantom, "\hat{D}_a"] & \varepsilon \arrow[l, "a"] \arrow[u, "a"'] \\
\varepsilon \arrow[u, "a"] \arrow[ur, phantom, "R_a"] & \varepsilon \arrow[l, "id_{\underline{0}}"] \arrow[u, "id_{\underline{0}}"']
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{t}^2 & \mathsf{t} \arrow[l, "\nabla_{\underline{1}}"'] \\
\mathsf{t} \arrow[u, "id_{\underline{1}}\otimes a"] \arrow[ur, phantom, "\hat{D}'_a"] & \varepsilon \arrow[l, "a"] \arrow[u, "a"'] \\
\varepsilon \arrow[u, "a"] \arrow[ur, phantom, "R_a"] & \varepsilon \arrow[l, "id_{\underline{0}}"] \arrow[u, "id_{\underline{0}}"']
\end{tikzcd}
$$

Figure 23. Two ways for composing the tile α of Figure 22.

This is a very strong assumption that somehow clashes against the desirable constructive presentation of computation.

In (Sewell, 1998; Leifer & Milner, 2000) it is suggested that instead of studying the behaviour of a process in all possible contexts, the basic reduction rules of the system can be used to catch the least set of contexts that should be considered. This is obtained by considering all subterms of the sources of reduction rules. For example, if a reduction rewrites $f(g(a))$ to $h(b)$, then the essential contexts are $f(\_)$ and $f(g(\_))$, but not $h(\_)$, because only by embedding a term within these contexts a reduction may happen (unless it is already enabled inside the term itself). Unfortunately, this task is hard to accomplish in general, as the reduction semantics for process calculi usually impose suitable structural axioms on the processes. Nevertheless, the presence of sufficiently many *relative pushouts* in the category of states is enough for guaranteeing that the universal constructions exist (Leifer & Milner, 2000).

For logic programming, the problem of contextualization is reversed to the problem of instantiation, and we know in advance what are the interesting 'internal' contexts, namely the pullback projections. This allows us to transform all clauses (seen as reduction rules) by moving as much internal context as possible to the observational part. We separate the topmost operator (the predicate symbol) from the partial instantiation of its arguments and then the basic pullbacks allow us to build incrementally all the other decompositions (in particular, we are speaking about tiles R_f, R_∇ and R_γ).

Proposition 5

For each tile $T_c\!:\!p \xrightarrow[t]{id} G$ and each arrows t_1, t_2 such that $t = t_1; t_2$ (with t_2 not involving dischargers), then the tile $t_2; p \xrightarrow[t_1]{id} G$ is entailed by $\mathcal{R}_{\mathcal{P}}$.

Proof

The proof follows from Lemma 1, i.e., from the existence of the tile $t_2 \xrightarrow[id]{t_2} id$ that can be vertically composed with $id \xrightarrow[t_1]{t_1} id$ (the horizontal identity for t_1), and with T_c being horizontally composed with the result (see Figure 24). □

Example 4.2

Let us consider the simple program

$c_1 \equiv$ `sum(0,x1,x1) .`

$c_2 \equiv$ `sum(s(x1),x2,s(x3)) :- sum(x1,x2,x3) .`

$$
\begin{tikzcd}
\mathsf{p} & \mathsf{t}^k \arrow[l, "p"'] & \mathsf{t}^m \arrow[l, "t_2"'] \\
T_c & \mathsf{t}^m \arrow[u, "t_2"'] & \mathsf{t}^m \arrow[l, "id_{\underline{m}}"] \arrow[u, "id_{\underline{m}}"'] \\
\mathsf{p} \arrow[uu, "id_{\underline{1}}"] & \mathsf{t}^n \arrow[l, "G"] \arrow[u, "t_1"'] & \mathsf{t}^n \arrow[l, "id_{\underline{n}}"] \arrow[u, "t_1"']
\end{tikzcd}
$$

Figure 24. Graphical proof of Proposition 5.

over the signature consiting of constant 0, unary symbol s and ternary predicate symbol *sum*. The possible interactive decompositions of the heads of the two clauses are:

- for c_1: (1) $sum(0,x_1,x_2)$ with observation $\nabla_{\underline{1}}$, (2) $sum(x_1,x_2,x_2)$ with observation $0\otimes id_{\underline{1}}$, (3) $sum(x_1,x_2,x_3)$ with observation $0\otimes\nabla_{\underline{1}}$, and (4) $sum(0,x_1,x_1)$ with observation $id_{\underline{1}}$;
- for c_2: (1) $sum(s(x_1),x_2,x_3)$ with observation $id_{\underline{2}}\otimes s$, (2) $sum(x_1,x_2,s(x_3))$ with observation $s\otimes id_{\underline{2}}$, (3) $sum(x_1,x_2,x_3)$ with observation $s\otimes id_{\underline{1}}\otimes s$, and finally (4) $sum(s(x_1),x_2,s(x_3))$ with observation $id_{\underline{3}}$.

However, the basic tiles of the tile system associated to the program just consider decompositions of kind (3), which are the most general. In fact, by parallel and sequential composition with (basic) pullback tiles, the tile logic associated to the tile system will entail all the other decompositions.

Note that tiles allow one to move contexts along states and observations in a very natural and uniform way.

The interactivity of the tile representation relies on the fact that the effects of basic tiles associated to the clauses must be accepted by the current instantiation of the matched predicate in the goal, otherwise the step cannot take place.

Theorem 3 (*Correspondence between (SLD-)derivations and tiles*)
Let $\mathcal{P}$ be a logic program and G a goal. Then,

1. if $\mathcal{P}\Vdash\ G\Rightarrow_\sigma G'$, then $\mathcal{R}_\mathcal{P}\vdash G\xrightarrow[\theta]{id} G'$ with $\theta=\sigma_{|Var(G)}$;
2. if $\mathcal{R}_\mathcal{P}\vdash G\xrightarrow[\theta]{id} G'$, then there exists σ with $\theta=\sigma_{|Var(G)}$ such that $\mathcal{P}\Vdash\ G\Rightarrow^*_\sigma G'$.

Proof
The proof of point 1 proceeds by rule induction. For the 'empty goal' rules we rely on the the fact that $\square$ is the unit for $\wedge$ and that the vertical identities always exist. For the 'atomic goal' we rely on the results of Section 2 on the correspondence between mgu's and pullbacks while applying the tile $T_{H:-F}$ to the goal A. For the 'conjunctive goal' rules, the difficulty is that G and G' might share some variables. In fact, by inductive hypothesis we can assume that $\mathcal{R}_\mathcal{P}\vdash G\xrightarrow[\sigma]{id} F$ and therefore we must employ the pullback tiles for propagating σ to G'. This can be done by exploiting the tiles $\hat{D}_f$ and D_∇.

For proving the point 2, we fix a decomposition of $G\xrightarrow[\theta]{id} G'$ in terms of basic tiles of $\mathcal{R}_\mathcal{P}$

and then we proceed by induction on the number of tiles T_c used for building $G \xrightarrow[\theta]{id} G'$.
□

Note that to a tile it may correspond a sequence of derivation steps.

Corollary 1
Let $\mathcal{P}$ be a logic program and G a goal. Then,

1. if $\mathcal{P} \vdash_\sigma G$, then $\mathcal{R}_\mathcal{P} \vdash G \xrightarrow[\theta]{id} \Box \otimes !_{\underline{n}}$ with $\theta = (\sigma)_{|Var(G)}$ and n the number of free variables in θ;
2. if $\mathcal{R}_\mathcal{P} \vdash G \xrightarrow[\theta]{id} \Box \otimes !_{\underline{n}}$, then there exists σ with $\theta = \sigma_{|Var(G)}$ such that $\mathcal{P} \vdash_\sigma G$.

4.3 Recovering ordinary semantics

The tile system $\mathcal{R}_\mathcal{P}$ is able to recover several well-known semantics for logic programs.

- The *minimal Herbrand model* is given by refutation tiles whose initial configuration is a ground atomic goal:

$$\mathrm{Op}_1(\mathcal{P}) = \{A : \varepsilon \to \mathsf{p} \mid \mathcal{R}_\mathcal{P} \vdash A \xrightarrow[!_{\underline{n}}]{id_{\underline{1}}} \Box \otimes !_{\underline{n}}\}.$$

- The *correct answer substitutions* are given by the instances of initial configurations of refutation tiles that are (possibly not ground) atomic goals:

$$\mathrm{Op}_2(\mathcal{P}) = \{A : \mathsf{t}^k \to \mathsf{p} \mid \mathcal{R}_\mathcal{P} \vdash A \xrightarrow[id_{\underline{k}} \otimes !_{\underline{k+n}}]{id_{\underline{1}}} \Box \otimes !_{\underline{n}}\}.$$

- The *computed answer substitutions* can be immediately obtained by considering the refutation tiles with a single predicate as initial configuration:

$$\mathrm{Op}_3(\mathcal{P}) = \{\theta; p : \mathsf{t}^k \to \mathsf{p} \mid p \in \Sigma_\Pi,\ \mathcal{R}_\mathcal{P} \vdash p \xrightarrow[\theta \otimes !_{\underline{n}}]{id_{\underline{1}}} \Box \otimes !_{\underline{k+n}}\}.$$

- The *resolvents* can be obtained by considering also non refutation tiles:

$$\mathrm{Op}_4(\mathcal{P}) = \{\theta; p :- G \mid p \in \Sigma_\Pi,\ \mathcal{R}_\mathcal{P} \vdash p \xrightarrow[\theta]{id_{\underline{1}}} G\}.$$

All the correspondences above follow as easy corollaries to the representation Theorem 3.

4.4 Goal compositionality

Though compositionality issues for the classical semantics have been extensively studied in the literature, we want to focus here on compositionality of goals w.r.t. the two main operations discussed in the Introduction, namely instantiation and conjunction (AND-compositionality). To some extent we concentrate the attention to compositionality 'in the small' (studying goal equivalence given the program $\mathcal{P}$) as opposed to compositionality of programs ('in the large').[2] Thus, the main questions are: (1) When are two goals equivalent? (2) Is equivalence a congruence?

[2] However, one can consider an infinite program built as the disjoint union of all finite programs.

Inspired by the connections with the area of process description calculi that motivated our approach, and having at hand an established theory developed for trace equivalence and bisimilarity in the tile setting, the natural step is to try to apply general existing techniques to our special case. Therefore we can answer question (1) by defining the two equivalences:

a. $G \simeq_{\mathcal{P}} G'$ if $\mathcal{T}_{\mathcal{P}}(G) = \mathcal{T}_{\mathcal{P}}(G')$, where $\mathcal{T}_{\mathcal{P}}(G) \stackrel{\text{def}}{=} \{\theta \mid \mathcal{R}_{\mathcal{P}} \vdash G \xrightarrow[\theta]{id_1} \Box \otimes id_{\underline{n}}\}$.
b. $G \cong_{\mathcal{P}} G'$ if G and G' are tile bisimilar in $\mathcal{R}_{\mathcal{P}}$.

These equivalences are reminiscent of the analogous notions on the processes of a fixed process description calculus modeled with tiles. The interactive part of the underlying tile system tells what can be observed during the computation, and then the equivalences arise naturally as behavior-based concepts.

Now, question (2) corresponds to ask whether $G \simeq G'$ implies that $G \wedge F \simeq G' \wedge F$ and $\sigma; G \simeq \sigma; G'$ for all F and σ or not (the same for $\cong$).

Though proving these property is not too complicate, we can exploit Proposition 1 and just prove that the tile system $\mathcal{R}_{\mathcal{P}}$ enjoys the decomposition property for any logic program $\mathcal{P}$.

Proposition 6
For any logic program $\mathcal{P}$, the corresponding tile system $\mathcal{R}_{\mathcal{P}}$ enjoys the sequential decomposition property.

Proof
We want to prove that for any goal $\sigma; G$ and tile $\mathcal{R}_{\mathcal{P}} \vdash \sigma; G \xrightarrow[\theta]{id_1} F$, there exist θ', σ' and F' such that $\mathcal{R}_{\mathcal{P}} \vdash G \xrightarrow[\theta']{id_1} F'$ and $\mathcal{R}_{\mathcal{P}} \vdash \sigma \xrightarrow[\theta]{\theta'} \sigma'$. Fixed a decomposition of $\sigma; G \xrightarrow[\theta]{id_1} F$ in terms of basic tiles, the proof proceeds by induction on the number of tiles associated to the clauses that are considered in the decomposition. $\Box$

Corollary 2
For any logic program $\mathcal{P}$, the equivalences $\simeq_{\mathcal{P}}$ and $\cong_{\mathcal{P}}$ are congruences with respect to conjunction of goals and instantiation of free variables.

4.5 Concurrency and causality

If we look at the the system $\mathcal{R}_{\mathcal{P}}$ from a concurrent viewpoint then atomic goals can be regarded as distributed components that can evolve separately and where variable sharing provides the mean to exchange information between components. According to this perspective, it is essential to keep track of the causal dependencies among components, e.g., for backtracking.

To accomplish this view we slightly modify the associated tile system for defining more concrete observation on the causal dependencies among replaced and inserted goals. To this aim, to each clause

$$c \equiv p(t_1, \ldots, t_k) \ :- \ q_1(\vec{s}_1), \ldots, q_m(\vec{s}_m)$$

in the logic program, we associate an operator $c\colon \mathsf{p}^m \to \mathsf{p}$ in the signature of observations.

Then, since we want to be aware of the components distributed in the system, we do not consider the operator $\wedge$ and associate to each clause C the tile

$$C_c = p \xrightarrow[\langle t_1,\ldots,t_k\rangle]{c} \langle q_1(\vec{s}_1),\ldots,q_m(\vec{s}_m)\rangle.$$

Note that by using the trigger c we can now observe that the initial configuration p generates m new components that causally depends on it.

For the rest, we add as before the pullbacks tiles that provide the coordination mechanism about local instantiations. In the new setting, equivalent computations from the point of view of concurrency and coordination are identified, whereas computations that return the same computed answer substitution but that employ different concurrent reduction strategies are distinguished. This also allows one to observe causal dependencies among resolution steps, since the triggers of refutation tiles describe the 'concurrent strategy' employed for achieving the result.

Of course, the notion of refutation tile slightly changes according to the above modification: A refutation tile is an entailed tile of the form $G \xrightarrow[\theta]{s} \square^m \otimes !_{\underline{n}}$, i.e., empty goals become *nil* processes distributed around system locations.

The trigger of the refutation tile for a generic goal is a tuple of terms (without shared variables), i.e., it corresponds to an ordered forest of (ordered) trees, whose nodes are labeled with clauses names. We denote by $\sqsubseteq$ the obvious partial order on nodes such that $x \sqsubseteq y$ iff y descends from x. Moreover, since the tree is ordered, we have an immediate correspondence between each clause instance and the subgoal it was applied to. Then, we can characterize the concurrency of the framework by means of the following theorem.

Theorem 4

Let $G \xrightarrow[\theta]{s} \square^m \otimes !_{\underline{n}}$ be a tile refutation for the goal G, and let $\sqsubseteq$ be the partial order (forest) associated to s. Moreover, let $\preceq$ be any total order that extends $\sqsubseteq$. Then by applying the clauses associated to the nodes of the tree in the order specified by $\preceq$ we obtain again the computed answer substitution θ.

The proof is based on the compositional properties of pullbacks and expresses the 'complete concurrency' (from the trigger side, not from the effect side) of the framework.

Since application of clauses that do not depend on each other (in $\sqsubseteq$) can be executed in any order by choosing suitable total orders $\preceq$, it follows that the order in which they are executed is not important. Note however that this depends on the fact that the coordination mechanism via pullbacks takes care of the side-effects of each clause application.

Conclusions and future work

In this paper we have used tile logic to model the coordination and interaction features of logic programming. Our approach differs from that of (Corradini & Montanari, 1992), based on structured transition systems, by taking into account the computed answer substitutions instead of just the correct answer substitutions. In fact, in (Corradini & Montanari, 1992), the clauses are seen as rewrite rules that can be further instantiated in all possible ways and the computational model of a program is a suitable 2-category (i.e., a special kind of double category, where the vertical category is discrete and thus only identities are allowed as observations). This means that if there exists a refutation for the goal G with computed answer substitution θ, then in the 2-category model we can find a refutation for $\theta; G$ but not necessarily one for G. The main advantages of our approach w.r.t. the one in (Corradini & Montanari, 1992) are a finite branching operational semantics and the built-in unification mechanism. Moreover, the drawback of using 2-categories instead of double categories is that the dynamic creation of fresh variables cannot be modeled, i.e., the variables to be used must all be present at the beginning of the computation.

As noted in the Introduction, the usage of tiles emphasizes the duality of instantiation and contextualization of goals, allowing for a uniform treatment of both. In particular, while instantiation plays a fundamental role since it can affect the behaviour of the goal, here contextualization (i.e., conjunction with disjoint goals) does not increase the distinguishing power of the semantics, and therefore transitions labeled with external contexts can be avoided in the model. In fact, each atomic goal can be studied in isolation from other goals and the tiles for putting a goal in any possible conjunction are not necessary. The reason for this absence is that goals cannot be conjoined in Horn clauses' heads, whereas if multi-headed clauses were considered, then, in general, abstract semantics would not turn out to be a congruence unless transitions for external contexts are added. Indeed, we believe that our approach can be extended to other frameworks where multi-head clauses are allowed (see for example the CHR formalism (Frühwirth, 1995)), giving us the key for dealing with contextualization features — dually to the instantiation via pullbacks considered here — as first class citizens.

We also would like to generalize our approach to model constraint logic programming (Marriott & Stuckey, 1998); in this more general case, probably the algebraic structures would need some reworking, and pullbacks could be not enough to describe constraints. However, there could be special classes of constraints, like those for which constraint sets have a normal solved form, for which pullbacks or slightly generalized concepts could be enough.

Exploiting the built-in synchronization features of tile logic, we are confident that our framework can be naturally extended to deal also with sequentialized commits (i.e., goals of the form $G_1; G_2; \cdots; G_k$ where the possibly non-atomic subgoals G_i must be resolved in the order given by their indices i, giving the possibility to the user of specifying more efficient resolution strategies). Moreover, the higher-order version of tile logic presented in (Bruni & Montanari, 1999) may find application to the modeling of higher-order logic programming (e.g., *lambda prolog*) (Miller, 1995; Miller & Nadathur, 1998).

Finally, let us mention that the abstractness of the unification via pullbacks makes the tile approach suitable for considering unification in equational theories rather than in term algebras. For example, this would allow to develop a computational model for rewriting logic (and hence for reduction systems on processes up to structural congruence) based on unification.

Acknowledgements

The authors would like to thank Paolo Baldan and Fabio Gadducci for their comments on a preliminary version of this paper.

This research has been supported by CNR Integrated Project *Progettazione e Verifica di*

Sistemi Eterogenei; by Esprit WG *CONFER2* and *COORDINA*; and by MURST project *TOSCA*.

References

Bernstein, K. (1998). A congruence theorem for structured operational semantics of higher-order languages. *Pages 153–164 of: Proceedings LICS'98, 13th Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press.

Berry, G., & Boudol, G. (1992). The chemical abstract machine. *Theoret. comput. sci.*, **96**(1), 217–248.

Bloom, B., Istrail, S., & Meyer, A.R. (1995). Bisimulation can't be traced. *Journal of the acm*, **42**(1), 232–268.

Bossi, A., Gabbrielli, M., Levi, G., & Meo, M.C. (1994). A compositional semantics for logic programs. *Theoretical computer science*, **122**(1-2), 3–47.

Brogi, A., Lamma, E., & Mello, P. (1992). Compositional model-theoretic semantics for logic programs. *New generation computing*, **11**(1), 1–21.

Bruni, R. (1999). *Tile logic for synchronized rewriting of concurrent systems.* Ph.D. thesis, Computer Science Department, University of Pisa.

Bruni, R., & Montanari, U. (1999). Cartesian closed double categories, their lambda-notation, and the pi-calculus. *Pages 246–265 of: Proceedings LICS'99, 14th Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press.

Bruni, R., & Montanari, U. (2000). Zero-safe nets: Comparing the collective and individual token approaches. *Information and computation*, **156**, 46–89.

Bruni, R., Meseguer, J., & Montanari, U. (1998). *Process and term tile logic.* Tech. rept. SRI-CSL-98-06. SRI International. Also Technical Report TR-98-09, Computer Science Department, University of Pisa.

Bruni, R., Meseguer, J., & Montanari, U. (1999). Executable tile specifications for process calculi. *Pages 60–76 of:* Finance, J.-P. (ed), *Proceedings FASE'99, Fundamental Approaches to Software Engineering.* Lect. Notes in Comput. Sci., vol. 1577. Springer Verlag.

Bruni, R., de Frutos-Escrig, D., Martì-Oliet, N., & Montanari, U. (2000a). Bisimilarity congruences for open terms and term graphs via tile logic. *Pages 259–274 of:* Palamidessi, C. (ed), *Proceedings CONCUR 2000, 11th International Conference on Concurrency Theory.* Lect. Notes in Comput. Sci., vol. 1877. Springer Verlag.

Bruni, R., Montanari, U., & Sassone, V. (2000b). Open ended systems, dynamic bisimulation and tile logic. *Pages 440–456 of:* van Leeuwen, J., Watanabe, O., Hagiya, M., Mosses, P.D., & Ito, T. (eds), *Proceedings IFIP TCS2000, IFIP Int. Conf. on Theoretical Computer Science.* Lect. Notes in Comput. Sci., vol. 1872. Springer Verlag.

Bruni, R., Meseguer, J., & Montanari, U. (2000c). Symmetric monoidal and cartesian double categories as a semantic framework for tile logic. *Mathematical structures in computer science.* To appear.

Burstall, R.M., & Rydeheard, D.E. (1985). A categorical unification algorithm. *Pages 493–505 of:* Abramsky, S., Pitt, D., Poigne, A., & Rydeheard, D. (eds), *Proceedings Workshop on Category Theory and Computer Programming.* Lect. Notes in Comput. Sci., vol. 240. Springer Verlag.

Cattani, G.L., Leifer, J.J., & Milner, R. (2000). *Contexts and embeddings for a class of action graphs.* Tech. rept. 496. Computer Laboratory, University of Cambridge.

Corradini, A., & Montanari, U. (1992). An algebraic semantics for structured transition systems and its application to logic programs. *Theoret. comput. sci.*, **103**, 51–106.

De Nicola, R., & Hennessy, M. (1984). Testing equivalences for processes. *Theoret. comput. sci.*, **34**, 83–133.

De Simone, R. (1985). Higher level synchronizing devices in MEIJE–SCCS. *Theoret. comput. sci.*, **37**, 245–267.

Ehresmann, E. (1963a). Catégories structurèes: I–II. *Annales école normal superieur*, **80**, 349–426.

Ehresmann, E. (1963b). Catégories structurèes: III. *Cahiers de topologie ed géométrie différentielle*, **5**.

Ferrari, G.L., & Montanari, U. (2000). Tile formats for located and mobile systems. *Information and computation*, **156**, 173–235.

Frühwirth, T.W. (1995). Constraint handling rules. *Pages 90–107 of:* Podelski, A. (ed), *Constraint Programming: Basics and Trends*. Lect. Notes in Comput. Sci., vol. 910. Springer Verlag.

Gadducci, F., & Montanari, U. (1996). Rewriting rules and CCS. *Proceedings WRLA'96, workshop on Rewriting Logic and its Applications*. Elect. Notes in Th. Comput. Sci., vol. 4. Elsevier Science.

Gadducci, F., & Montanari, U. (2000). The tile model. Plotkin, G., Stirling, C., & Tofte, M. (eds), *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press. Also Technical Report TR-27/96, Dipartimento di Informatica, Università di Pisa, 1996.

Gaifman, H., & Shapiro, E. (1989). Fully abstract compositional semantics for logic programs. *Pages 134–142 of: Proceedings POPL'89*. ACM.

Groote, J.F., & Vaandrager, F. (1992). Structured operational semantics and bisimulation as a congruence. *Information and computation*, **100**, 202–260.

Larsen, K.G., & Xinxin, L. (1990). Compositionality through an operational semantics of contexts. *Pages 526–539 of:* Paterson, M.S. (ed), *Proceedings ICALP'90, 17th International Colloquium on Automata, Languages and Programming*. Lect. Notes in Comput. Sci., vol. 443. Springer Verlag.

Lawvere, F.W. (1963). Functorial semantics of algebraic theories. *Proc. National Academy of Science*, **50**, 869–872.

Leifer, J.J., & Milner, R. (2000). Deriving bisimulation congruences for reactive systems. *Pages 243–258 of:* Palamidessi, C. (ed), *Proceedings CONCUR 2000, 11th International Conference on Concurrency Theory*. Lect. Notes in Comput. Sci., vol. 1877. Springer Verlag.

Lloyd, J.W. (1987). *Foundations of logic programming*. Springer.

MacLane, S. (1971). *Categories for the working mathematician*. Springer.

Mancarella, P., & Pedreschi, D. (1987). An algebra of logic programs. *Pages 1006–1023 of:* R.A., Kowalski, & A., Bowen K. (eds), *Proceedings ICLP'88, 5th International Conference on Logoc Programming*. MIT Press.

Marriott, K., & Stuckey, P.J. (1998). *Programming with constraints: an introduction*. MIT Press.

Meseguer, J. (1992). Conditional rewriting logic as a unified model of concurrency. *Theoret. comput. sci.*, **96**, 73–155.

Meseguer, J., & Montanari, U. (1998). Mapping tile logic into rewriting logic. *Pages 62–91 of:* Parisi-Presicce, F. (ed), *Proceedings WADT'97, Recent Trends in Algebraic Development Techniques*. Lect. Notes in Comput. Sci., vol. 1376. Springer Verlag.

Miller, D. (1995). *lambda prolog: An introduction to the language and its logic*. Unpublished. Available at `http://www.cse.psu.edu/~dale/lProlog/docs.html`.

Miller, D., & Nadathur, G. (1998). Higher-order logic programming. *Pages 499–590 of:* Gabbay, D.M., Hogger, C.J., & Robinson, J.A. (eds), *Handbook of logics for artificial intelligence and logic programming*, vol. 5. Clarendon Press.

Milner, R. (1980). *A Calculus of Communicating Systems*. Lect. Notes in Comput. Sci., vol. 92. Springer Verlag.

Milner, R. (1992). The polyadic pi-calculus (abstract). *Page 1 of:* Cleaveland, R. (ed), *Proceedings CONCUR '92, 3rd International Conference on Concurrency Theory*. Lect. Notes in Comput. Sci., vol. 630. Springer Verlag.

Milner, R. (1996). Calculi for interaction. *Acta informatica*, **33**(8), 707–737.

Montanari, U., & Sassone, V. (1992). Dynamic congruence vs. progressing bisimulation for CCS. *Fundamenta informaticae*, **16**, 171–196.

Montanari, U., & Talcott, C. (1998). Can actors and π-agents live together ? *Proceedings HOOTS'98, 2nd Workshop on Higher Order Operational Techniques in Semantics*. Elect. Notes in Th. Comput. Sci., vol. 10. Elsevier Science.

Park, D. (1981). Concurrency and automata on infinite sequences. *Pages 167–183 of: Proceedings Fifth G-I Conference*. Lect. Notes in Comput. Sci., vol. 104. Springer Verlag.

Plotkin, G. (1981). *A structural approach to operational semantics*. Tech. rept. DAIMI FN-19. Aarhus University, Computer Science Department.

Rensink, A. (2000). Bisimilarity of open terms. *Information and computation*, **156**, 345–385.

Sewell, P. (1998). From rewrite rules to bisimulation congruences. *Pages 269–284 of:* Sangiorgi, D., & de Simone, R. (eds), *Proceedings CONCUR'98*. Lect. Notes in Comput. Sci., vol. 1466. Springer Verlag.