Mathware & Soft Computing 4 (1997) 281-297

Towards Specifying with Inclusions[*]

J. Agustí, J. Puigsegur and W. M. Schorlemmer
Institut d'Investigació en Intel·ligència Artificial (CSIC)
Campus de la UAB, E-08193 Bellaterra, Catalonia
e-mail: {*agusti,jpf,marco*}*@iiia.csic.es*

Abstract

In this article we present a functional specification language based on inclusions between set expressions. Instead of computing with data individuals we deal with their classification into sets. The specification of functions and relations by means of inclusions can be considered as a generalization of the conventional algebraic specification by means of equations. The main aim of this generalization is to facilitate the incremental refinement of specifications. Furthermore, inclusional specifications admit a natural visual syntax which can also be used to visualize the reasoning process. We show that reasoning with inclusions is well captured by bi-rewriting, a rewriting technique introduced by Levy and Agustí [15]. However, there are still key problems to be solved in order to have executable inclusional specifications, necessary for rapid prototyping purposes. The article mainly points to the potentialities and difficulties of specifying with inclusions.

Keywords: diagrammatic reasoning, visual languages, declarative programming, formal specification.

1 Introduction

The systematic development of correct programs from complete formal specifications by means of verified refinement steps has attracted considerable effort [6]. Much less attention has been devoted to formal *Requirements Engineering*; i.e. the difficult process of creation of adequate formal and complete specifications from informal requirements. Specifications cannot be validated conclusively with respect to the real world, they can only be judged subjectively as adequate descriptions of a problem, maybe with the help of some theorem prover which computes consequences of the formulation. So formal specification languages to be widely used in *Requirements Engineering* [9], should not present undue difficulties of use and interpretation to the persons who create and read the specification, who usually are experts on the appli cation domain and not programmers. This article is based

[*]This work is supported by the project DISCOR (TIC 94-0847-C02-01) funded by the CICYT

on the research done in our group on the *Calculus of Refinements (COR)*, a functional specification language designed to shorten the distance between the informal description of a problem and its first formalization (preferably executable for rapid prototyping purposes). This research concentrates on an incremental approach to executable formal specification where preliminary specifications can be expressed in a notation accessible to non-specialists whilst providing clear points of refinement towards complete specificati ons in languages targeted to more specialised developers.

The basic and intuitive semantic notion of our language is that of set. Everything is viewed in some way as a set. Instead of dealing (reasoning or computing) with data individuals we work with their classification into types or sorts (also called classes, descriptions, approximations or concepts, depending on the area of interest). For instance, to handle numerical functions like in the example of Section 4.2, we create and reason with sorts like "*nat*" (for the set of natural numbers), "$mult(nat)$" (the multiples of naturals), etc. In our approach individuals are considered as a particular kind of set, namely as singletons. Even functions and relations are seen as sets of images (outputs) dependent on parameters which are also expressed as sets (sets of inputs). In Figure 3 of Section 2 we describe the *ancestor* relation and we consider it as a function returning the sets of ancestors of a given set of persons. Then, we give information about *ancestor* by giving subsets of it: for example the parents of the same persons. In this language set expressions are built as terms using constants, variables and function symbols. The formal details are given in the next section where we also show how these terms and formulas are represented in the alternative visual syntax of our set-based language.

The main predication on set expressions is the inclusion between them. To express the meaning of a set expression we give upper and lower bounds as superset and subset expressions, as can be seen in the example of Section 4.2. This specification of functions and relations by means of inclusions can be considered as a generalization of algebraic specifications by means of equations. The main aim of this generalization is to facilitate the incremental refinement of specifications. Inclusions can be seen as more expressive and flexible constraints than equations: upper and lower bounds can be successively refined by closer bounds towards an equality relation, usually between singleton expressions. The example about multiples of natural numbers in Section 4.2 shows briefly such kind of refinements.

Instead of a set-based language we could use the well known *First Order Logic* (FOL) to express the same information. However —we claim— that set expressions and inclusions provide frequently enough a more natural and less demanding sublanguage for non-logicians than the corresponding first order formulas. For instance, the simple inclusion $man \subseteq mortal$ should be expressed in FOL as $\forall x(man(x) \rightarrow mortal(x))$. The usual operations between set expressions, intersection and union, and the inclusion relation, we claim are easier to understand by non-logicians than the FOL connectives conjunction, disjunction and implication. It has also been shown in [18] how a similar set syntax for FOL has some advantages when automating deduction. Other similar taxonomic knowledge representation languages have also proved its methodological and technical superiority to deal with taxonomic knowledge, compared with FOL (see [17]). Furthermore, a way to make these languages more accessible to non-logicians is by means of its diagrammatic representation. One of the advantages of set-based languages like ours is that they admit a natural diagrammatic representation similar to Venn diagrams. This diagrams can be used naturally to express information on functions, and also allow some kind of automatic reasoning to be performed on them, as we will see below.

Specifications based on inclusions showed interest in themselves from a theoretical point of view. Peter Mosses [20] was one of the first to consider them as a new framework, called *unified algebras*, for the algebraic specification of abstract data types. Independently we explored the same idea on a more general framework, that of higher order functional languages. The *Calculus of Refinements* (COR) we defined can be considered as an extension of λ-calculus with lattice operations where specifications are sets of inclusions between λ-terms [16]. In a series of papers and the doctoral thesis of J. Levy [13] we investigated the denotational semantics (the class of models of COR [1]) and its operational semantics based on rewrite techniques [14]. Based on this results we tried to use a first order version of COR as a requirements specification language in the framework of logic programming [22]. The present article tries to push this application a bit further. First by presenting a visual syntax based on higraphs, a topological diagrammatic formalism, as an alternative to the usual textual syntax. Second by showing the possibilities and difficulties of an executable specification language based on inclusions.

2 The Language

2.1 Syntax

As said in the introduction, our set-based language has a textual syntax and an alternative visual syntax. Effectively, the special features of our set-based functional language allow a direct topological visual representation. While the textual syntax of our language is similar to syntax used in many other equational logic languages, the visual syntax is a customization of higraphs, an open-ended topological formalism developed by David Harel [8]. Higraphs combine two well-known diagrammatic formalisms: graphs and Venn diagrams. A higraph is a graph whose nodes represent sets which are related by a binary relation: graphical inclusion. This diagrammatic language differs from traditional set-based diagrams, like Venn diagrams, in two ways: 1) It is possible to construct new sets by building compound terms through visual functional application. 2) It is also possible for an element or set of elements to be repeated in different places of the diagram at the same time. This is, if two boxes are graphically disjoint it does not imply that the associated sets are disjoint. The only relevant information in the diagrams is the graphical inclusion between boxes (sets) and circles (set variables) in boxes.

2.1.1 Terms

Terms are syntactically defined as usual in functional languages, with two extra constructors: union and intersection, and two extra elements: top and bottom (top

Figure 1: The Symbols of the Visual Syntax

$f(X, g(X, a))$ $\qquad$ $f(X) \cup g(X)$ $\qquad$ $f(a \cap g(X), X)$

Figure 2: Examples of Visual Terms

and bottom are not used by now in the visual syntax). All terms are interpreted as sets: constants and variables represent sets, and functions are extended over sets, i.e. they take them as arguments and return a set.

The construction of the corresponding visual terms as directed acyclic graphs (DAGs) is straightforward. In Figure 1, we show the graphical representation of nodes of the higraph (symbols). Circles denote sets which correspond to variables, while boxes denote sets associated to the other types of terms of the language. Every box is named after the outer-most function of the term it represents. Nodes are connected using arrows that point from the subterms to the node that represents the functor of the term. Arrows represent functional application. Note that in some DAG representations of terms, arrows go in the opposite direction. The reason of our way is that we want to stress the fact that functions map elements from the domain sets to elements of the image set. The structure of a term is therefore a DAG where nodes are boxes and circles, and circles do not have incoming arrows (the edges of the graph). In Figure 2 we show examples of visual terms together with their textual equivalents.

The graphical representation of terms using DAGs is one of the keypoints of our visual language. DAGs allow to share common parts of subterms, reducing the quantity of symbols in the visual terms. For instance, variables do not need to have

(a) $grandparent(X) \supseteq parent(parent(X))$
(b) $ancestor(X) \supseteq parent(X)$
$ancestor(X) \supseteq ancestor(parent(X))$
(c) $X \subseteq descendant(Y) \Leftarrow Y \subseteq ancestor(X)$

Figure 3: Examples of Visual Formulas

a name, and if they appear many times in a textual term, they will appear just once in the corresponding visual term, thus reducing the complexity of the term and making it easier to read in its visual form.

2.1.2 Formulas

An atomic formula of the textual specification language is an inclusion between terms, and a formula is either an atomic formula or an inclusional Horn Clause. They are defined in the following way:

Definition 1 ***Formulas***
– *an atomic formula is an* inclusion, $t_1 \subseteq t_2$*, where* t_1 *and* t_2 *are terms.*
– *a* clause $f \Leftarrow f_1 \wedge f_2 \wedge \ldots \wedge f_n$ *is a formula, where* f,f_1,... f_n *are atomic formulas and* $n \geq 1$.

The basic units of our visual language —equivalent to formulas in the textual language— are diagrams. A diagram is the smallest complete unit of description and it is composed of various visual terms related by graphical inclusion. In every diagram there is a goal set term, which is the term that is being partially defined. To define a set term we indicate which are its subset terms. The goal set term is marked by drawing its box using thick lines. Therefore a diagram is usually equivalent to an inclusional clause where the head is the main inclusion —the goal-set inclusion— and the rest of graphical inclusions in the diagram is the body of the clause. In [21] we presented a variant of this visual language more addressed to declarative programming.

In Figure 3 we find three examples of visual formulas and their equivalent textual formulas, representing the relations *grandparent*, *ancestor* and *descendant*. Let's now examine how the examples of Figure 3 are constructed. In all three diagrams we are representing relations as functions over sets that return sets. In diagram (a) the *grandparent* relation is represented as a function that applied to a set X (of persons) returns the set $grandparent(X)$ (of their grandparents). This relation is then defined by giving a subset of the image set: $parent(parent(X))$, i.e. "the parents of the parents of X are the grandparents of X". The functional notation allows us to compose relations, as it is done in this diagram applying twice the *parent* relation to the input set X. Diagram (b) defines the *ancestor* relation by stating that "the parents of X and the ancestors of the parents of X are ancestors of X". Note that in the visual syntax a diagram can represent more than one clause when there is no conflict with the conditions of each clause. In this case, since there are no conditions, we can represent both clauses in a single diagram. Finally, diagram (c) defines the relation *descendant*, the inverse relation of *ancestor*.

2.2 Semantics

As we have said the attempted intuitive meaning of set terms are sets built from constant sets and function application on them. The meaning of term inclusions is the usual set inclusion meaning. However the class of models satisfying the inclusional Horn clauses is wider and more abstract. In the following we define this class of models, the satisfaction relation between models and formulas and the entailment relation between formulas. All of them are particular first order cases of the general class of COR models, COR satisfaction and entailment relations [1].

Let Σ be the set of constants and function symbols and $\mathcal{V}$ the variables used to build our terms. Σ includes the constant symbols top ($\top$) and bottom ($\perp$) and the binary functions $\cup$ and $\cap$. A Σ-model $\mathcal{A}$ consists of a set $\|\mathcal{A}\|$ which is a lattice with $\cup_{\mathcal{A}}$ as join and $\cap_{\mathcal{A}}$ as meet, the element $\top_{\mathcal{A}}$ as top and the element $\perp_{\mathcal{A}}$ as bottom. Let $\leq_{\mathcal{A}}$ denote the partial order of the lattice. The other constant symbols $C \in \Sigma$ are interpreted as elements $C_{\mathcal{A}} \in \|\mathcal{A}\|$ and the function symbols f of arity n as functions $f_{\mathcal{A}} : \|\mathcal{A}\|^n \to \|\mathcal{A}\|$. All functions $f_{\mathcal{A}}$ are monotone with respect to $\leq_{\mathcal{A}}$. Using the previous interpretation to each valuation of variables $\rho : \|\mathcal{A}\| \to \|\mathcal{A}\|$ corresponds the homomorphic interpretation of terms on $\mathcal{A}$ defined as usual: $\Phi^{\rho}_{\mathcal{A}} : \mathcal{T}(\Sigma, \mathcal{V}) \to \|\mathcal{A}\|$. We say a model $\mathcal{A}$ satisfies the formula $t_1 \subseteq t_2$, $\mathcal{A} \models_{\rho} t_1 \subseteq t_2$ iff $\forall \rho : \mathcal{V} \to \|\mathcal{A}\| \;\; \Phi^{\rho}_{\mathcal{A}}(t_1) \leq_{\mathcal{A}} \Phi^{\rho}_{\mathcal{A}}(t_2)$.

The satisfaction of an inclusional Horn clause by a model is defined using the standard interpretation of FOL connectives. The defined class of models has initial models isomorphic to the standard term model. Power set algebras and its subalgebras can be considered also as intuitive models of inclusional specifications. However, they are not initial models as has been shown in [20].

The entailment relation between a set of formulas $\mathcal{I}$ and a formula φ is defined by the lattice axioms, the inclusion inference rules, reflexivity and transitivity; the variable substitution (replacement) and function monotonicity rules ; and the resolution rule [1]. This entailment relation so defined is not enough to mechanize the deduction with inclusions, necessary to our goal of having executable specifications. The rest of the article focuses on this key problem showing the advances reached till now and the remaining problems.

3 Reasoning with Inclusions

In order to validate preliminary specifications one solution is, as mentioned in the introduction, to make a first implementation by *rapid prototyping.* We want therefore a specification to be executable, in order to verify properties on it. In this section we show the techniques for mechanizing the deduction in those logical theories underlying our specifications, namely in inclusional theories.

Recently it has been shown that term rewriting techniques, which have turned out to be among the more successful approaches to equational theorem proving, are suitable for defining specialized proof calculi for inclusional theories. It is well known that rewriting implicitly captures the transitivity and congruence properties of the equality relation in a natural way, and avoids the explicit use of the equality axioms, which pose severe problems in the design of efficient automated t heorem provers. But, since rewrite rules rewrite terms in one direction, it is not only in reasoning with the equality relation where these techniques naturally apply, but in reasoning with arbitrary, possibly *non-symmetric*, transitive relations, as e.g. inclusions . In fact, Meseguer noticed that the logic underlying rewrite systems in not equational logic but *rewriting logic* [19]. Levy and Agustí where the first in using techniques of term rewriting to define a decision procedure for theories with non-symmetric relations [14]. They generalized to inclusional theories the notions of confluence and termination of term rewrite systems based on equational theories, by introducing the so called *bi-rewrite systems* [15].

In the same sense in that the theory of rewrite systems has provided us with an efficient operational semantics for equational logic, its generalization to bi-rewriting will be a suitable basis for an operational semantics for our specification language b ased on inclusional theories.

3.1 Theorem proving with inclusions: Bi-rewrite systems and ordered chaining

The fundamental idea lying behind bi-rewrite systems is to orient a given set of inclusions, following a reduction ordering on terms. We get in this way *two* independent rewrite systems, one containing those rewrite rules which rewrite terms into "bigger" ones (with respect to the inclusion relation), and the other one containing rewrite rules which rewrite terms into "smalle r" ones. We will distinguish the two separate rewrite relations by denoting the first one with $\xrightarrow{\subseteq}$ and the second one with $\xrightarrow{\supseteq}$. Consider for example the inclusional theory presentation I consisting of the following axioms:

$$\begin{array}{l} f(a,x) \subseteq x \\ f(x,c) \subseteq x \\ b \subseteq f(a,c) \end{array}$$

If we orient these inclusions, following e.g. a lexicographic path ordering based on the signature precedence $f \succ c \succ b \succ a$[1], we obtain the following two rewrite system s:

$$R_1 = \left\{ \begin{array}{l} f(a,x) \xrightarrow{\subseteq} x \\ f(x,c) \xrightarrow{\subseteq} x \end{array} \right. \qquad R_2 = \left\{ \; f(a,c) \xrightarrow{\supseteq} b \right.$$

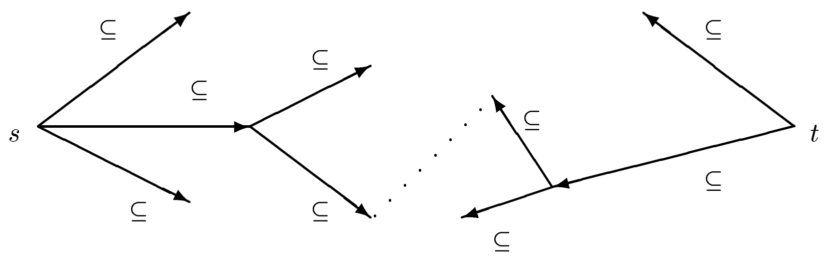
We say that these two rewrite systems form a *bi-rewrite system* $B = \langle R_1, R_2 \rangle$.

In order to have a decision procedure for the word problem of an inclusional theory we need the bi-rewrite system to be *convergent*, i.e. it has to satisfy two properties: *Church-Rosser* and *termination*[2]. The system is Church-Rosser if whenever we have two terms s and t such that, $I \vdash s \subseteq t$[3] a *bi-rewrite proof* between these terms exists, consisting of two paths, one using rules of R_1 and the other using rules of R_2, which join together in a common term:

$$s \xrightarrow{\subseteq} \cdots \xrightarrow{\subseteq} u \xleftarrow{\subseteq} \cdots \xleftarrow{\subseteq} t$$

The system is terminating, if no infinite sequences of rewrites with rules in R_1 (or R_2) can be built. Termination is guaranteed when the rewrite orderings defined by R_1 and R_2 respectively are contained in a unique reduction ordering on terms. Bi-rewrite systems fulfilling termination and the Church-Rosser property are said to be *convergent.*

A decision procedure[4] for the word problem in convergent bi-rewrite systems is then straightforward: To check if $I \vdash s \subseteq t$ we reduce s and t applying rewrite rules of each rewrite system, and exploring all possible paths, until a common term is reached:



The conditions put on the rewrite relations in order to guarantee termination also avoid the possibility of infinite branching.

Convergent bi-rewrite systems finitely encode the reflexive, transitive and monotone closure of the inclusion relation $\subseteq$: All possible consequences of a set of inclusions I using transitivity, reflexivity and monotonicity can be represented by a bi-rewrite proof.

[1]We refer to [7] for a survey on termination orderings.

[2]To be rigorous we only need quasi-termination [14], but for the sake of simplicity, termination is required.

[3]The entailment relation $\vdash$ is defined by the reflexivity, transitivity and monotonicity inference rules of the inclusion relation $\subseteq$.

[4]Actually it is a decision algorithm because of termination.

An arbitrary bi-rewrite system, obtained by orienting the inclusions of an inclusional theory presentation I is non-convergent in general. But, like in the equational case, there exist necessary and sufficient conditions for a terminating bi-rewrite system to be Church-Rosser, which were stated by Levy and Agustí [15] adapting the original results of Knuth and Bendix [12]. Following the same ideas proposed by Knuth and Bendix, one can attempt to complete a non-convergent terminating bi-rewrite system, by means of adding new rewrite rules to the systems R_1 or R_2.

Within the context of resolution-based theorem proving with full first-order clauses there have been attempts to use additional inferences in order to avoid the transitivity axiom of arbitrary transitive relations, like paramodulation in first-order theor ies with equality. Slagle [24] introduced for these purposes the chaining inference rule. Chaining can be seen as the generalization of paramodulation for arbitrary transitive relations. As paramodulation, chaining has the drawback that it explicitly generates the transitive closure of the binary relation. But like ordering restrictions on the paramodulation inference have led to the superposition calculus [10, 4] (which in essence generalizes the computation of new equations during the Knuth-Bendix completion process), so ordering restrictions on the chaining inference rule take the advantages of rewrite techniques resul ting from bi-rewrite systems, and avoid generating the whole closure using bi-rewrite proofs to prove the validity of a transitive relation. The result is the calculus presented by Bachmair and Ganzinger [5], which is based on the *ordered chaining* inference rule between two clauses:

$$\textbf{Ordered Chaining:}\quad \frac{C \vee u < s \quad D \vee t < v}{C\sigma \vee D\sigma \vee u\sigma < v\sigma}$$

where σ is a most general unifier of s and t, and the following ordering restrictions between terms, and literals hold: $u\sigma$[5] $\not\succ s\sigma$, $v\sigma \not\succ t\sigma$, $u\sigma < s\sigma$ is the strictly maximal literal with respect to the rest $C\sigma$ of the clause, and $v\sigma < t\sigma$ is the strictly maximal literal with respect to the rest $D\sigma$ of the clause[6].

3.2 Towards an operational semantics: Some drawbacks for efficiency

In the same sense in that the theory of rewrite systems has provided us with an efficient operational semantics for equational logic, we are convinced that its generalization to bi-rewriting is a suitable basis for an operational semantics of a specificat ion language based on inclusional theories. It is well known that if we want to use logic as a programming language we need to choose a suitable specialized proof system which will mark the difference between a hopeless theorem prover and an efficient programming language. The results on proof theory with arbitrary transitive relations are still too weak for defining powerful deduction strategies that may serve us for this purpose. Though we have seen that term rewriting is a suitable technique for reasoning with arbitrary transitive relations, several important differences to equational term rewriting appear, which are important for the practicability of deductions with inclusions.

[5]Application of substitution σ on terms, literals or clauses is denoted in postfix notation.

[6]It is possible to extend a simplification ordering on terms over literals and clauses (see [7]).

Figure 4: A Visual Program

First of all, instead of dealing with a single rewrite relation we have to manage a bi-rewrite system. Furthermore, no rewriting within equivalence classes of terms is done, making a notion of unique normal form on which equational term rewriting is based, meaningless. Consequently the order of application of rewrite rules is now significant, making term rewriting don't know nondeterministic: backtracking is needed for a rewrite proof to be found.

But the most important difference to the equational case, in the sense of practicability of the inference system, appears when reasoning with functions which are monotonic with respect to the transitive relation, which is the case in our specification lan guage. Besides chaining on proper subterms also *variable subterm chaining* is necessary which leads to a quite inefficient proof calculus if completeness results are wanted.

It is therefore necessary to restrict this general ordered chaining calculus if we want to avoid prolific first-order variable subterm chaining. As we will see later a way to improve the efficiency of the inference system is by studying certain algebraic structures, which can help us to consider only certain cases of variable subterm chaining. It is known, e.g. that in dense total orderings without endpoints, variable chaining can be avoided completely [3].

For a more extended survey on the state of the art of theorem proving with transitive relations we refer to [23].

3.3 Visually solving queries

As mentioned in section 2.1, our visual notation is based on using graphical set inclusion between diagrams instead of implication. The transitivity of the inclusion relation, which is also implicit in the chaining inference rule is trivially captured in our diagrams. It is therefore not surprising that inferences based on chaining are well presented in our visual notation. In other words, we can visually show the operational behaviour of the query solving process within our graphically stated specifications.

Figure 5: Query solving process

We sketch some of our ideas through an example: Given the visual specification of Figure 4, suppose we want to know, who are the grandparents of Charly. We express this query with the *query diagram* of Figure 5 (a). It differs from ordinary diagrams in that no box is marked by thick lines (because we are not defining a function) and that the unknown variables to be computed are noted with an interrogation mark. Queries are existentially quantified and solved by refuting their negation. This is achieved by transforming a *query diagram* into an *answer diagram* applying visual inferences. The idea behind this diagram transformation is to complete the query with the trace of its proof, and with the instantiation of its unknown variables. This query solving process is shown in Figure 5.

First, we match the *grandparent* box with its definition in Figure 4. The circle —the argument of grandparent— matches box *charly*, and the boxes referring to *parent* are added to the query diagram giving diagram (b). This inference step is actually a *chaining* step, since we are going to proof the existence of a subset in the *grandparent* box by means of its membership in the *parent* box contained in it. Therefore we place the unknown variable within the *parent* box. Next we match the *parent*(*charly*) box with its definition in Figure 4. This step corresponds to an ordinary *resolution* step and leads to diagram (c). In order to further solve the query we use the monotonicity property of function *parent* with respect to set inclusion: Since *bob* is subset of *parent*(*charly*), *parent* applied to *bob* is a subset of *parent* applied to *parent*(*charly*). The same happens to *parent* applied to *ann*. In fact, there are two alternatives to further compute the grandparents of Charly, and we choose to represent both in one single diagram (d). This corresponds to a breadth-first strategy. The unknown variable appears twice, showing that maybe two possible solutions can be computed. In this case the inference step are actually chaining steps, since we again are going to proof the existence of a subset in the *parent*(*parent*(*charly*)) box by means of its membership in the *parent*(*bob*) or *parent*(*ann*) box contained in it. Further transformation by resolution brings us finally to *answer diagram* (e), where the original unknown variable appears fully instantiated, presenting in this case four different solutions to the original query, namely that John, Mary, Tom and Sa lly are grandparents of Charly.

Several advantages of the visual notation are present in this simple example: We can keep the trace of the proof in our diagram during the query solving process in an elegant way, and different alternative answers can easily be presented within one single diagram. We are still in the very beginning of giving an operational semantics to our specification paradigm, and there is a lot to be explored about these or other advantages of the visual notation and its resulting open questions, as well as about the chaining-b ased operational semantics of our language.

4 Executing Specifications

We have seen in the previous section that it is necessary to restrict in some way the general ordered chaining calculus if we want to avoid some of the major drawbacks for efficient computation. One way to improve this is by studying certain algebraic structures. This options appears to be more promising, since several specification frameworks based on partial orders are based on specific algebraic structures.

4.1 A language based on lattices

As mentioned in section 2.2 the models of our specifications are lattices. Other specification paradigms are also based on lattices as e.g. Mosses' *unified algebras* [20]. Completion of the inclusional theory of free lattices to a convergent bi-rewrite system is possible [13], and this fact suggests to consider the properties of this specific algebraic structure for improving the deduction with inclusions. However this approach is not useful enough for a practical use of inclusions, as we will see later.

Lattices have also been chosen as interpretations for a variety of much more concrete logic programming languages, in which partial orders play a central role. Aït-Kaci and Podelski make use of order-sorted feature terms as basic data structure of the programming language LIFE [2], generalizing in this way the flat first-order terms normally used as unique data structure in logic programming. An order-sorted feature term is a compact way to represent the collection of elements of a given non-empty domain which satisfy the constraint encoded by the term, and therefore may be interpreted itself as a sort, like in *unified algebras* or in CO R, being LIFE one of the first proposals of sorts as values. Algebraically, a term denotes an element of a meet semi-lattice with a top $\top$ and a bottom $\perp$, which in essence is a subalgebra of the power set of the considered domain. But, deduction in LIFE is quite poor, because of the restricted use of terms within the definition of the partial order. Deduction reduces to unification of order-sorted feature terms and can be seen as the meet operation in the semi-lattice. It is performed by normalizing the conjunction of the constraints encoded in the terms to be unified, and is equivalent to intersecting the collections of elements the terms represent.

Also Jayaraman, Osorio and Moon base their *partial order programming* paradigm on a lattice structure, and are specially interested on the complete lattice of finite sets [11]. In their paradigm they pursue the aim to integrate sets into logic programming, and to consider them as basic data structure on which the paradigm relies. But in this framework no deduction mechanisms are given to validate order related functional expressions.

4.2 Functions as sort constructing operators

But besides efficiency issues, it is also necessary to think about how functional expressions are supposed to be evaluated, whenever functions are specified by axioms with partial orders instead of equations, as for example the following preliminary speci fication of the *mult* function, which given a set of natural numbers returns the set of their multiples[7]. Constant *nat* denotes the sort of all natural numbers:

$$\begin{array}{l} mult(nat) \subseteq nat \\ mult(nat) \supseteq nat * nat \end{array}$$

The first axiom specifies the function's type, while the second one gives a first approximation of its computational behaviour. A further refinement of both axioms could be as follows:

$$\begin{array}{rcl} mult(X) \subseteq nat & \Leftarrow & X \subseteq nat \\ mult(X) \supseteq X * nat & \Leftarrow & X \subseteq nat \end{array}$$

As said in the introduction, our aim during the refinement process, is to ultimately specify how the function will operate on elements. An additional refinement to the second axioms will suffice for this purpose:

$$mult(X) \supseteq X * Y \quad \Leftarrow \quad X \subseteq nat \wedge Y \subseteq nat$$

Recall that when rewriting is done on an arbitrary transitive relation, no rewriting inside equivalence classes of terms is done, and therefore, in a purely inclusional

[7]This example may serve only to clarify our intuitions; it isn't in any way a satisfactory specification of the multiple function.

programming language, function evaluation cannot be seen as normal form computation by reduction. Furthermore, rewriting on non-symmetric transitive relations becomes don't know nondeterministic.

In the approach followed by LIFE, functions are defined by rewrite rules to be interpreted as equations, as usual, and therefore expressions are reduced by equational rewriting to their normal forms. Maybe if we use partial orders to constrain functions we could exploit the semi-lattice structure of LIFE terms in order to not only evaluate functional expressions, but also to use them as new sort definitions, much more in the spirit of unified algebras or the calculus of refinements. For example, we would like expression $div(Y)$ to represent the sort of those elements that are the divisors of elements in Y (for this to have sense Y should be subset of nat). Such a sort expression would be useful, e.g. to add to the definition o f function $mult$ the following axiom:

$$mult(X) \supseteq Y \quad \Leftarrow \quad Y \subseteq nat \wedge X \subseteq div(Y)$$

Functions would be then, besides user-defined functions, also sort constructing operators. In such a context rewriting of functional expressions becomes a kind of sort checking, because now our interest would lie on knowing which are the upper or lower bounds of the functional expression, and in this way to check the correctness of the performe d refinement steps. For instance, we would like to infer that, given a natural number k (i.e. k is a singleton set such that $k \subseteq nat$), $k \subseteq mult(k) \subseteq nat$. But, though we have studied specific theorem proving techniques in order to perform such sort checking (see Section 3.1), we still lack of specific methodologies for doing so.

In Jayaraman's et al. partial order programming paradigm, though orders are used explicitly for the definition of functions, functional expressions are equated to the greatest lower bound (or least upper bound) of all irreducible terms reached by rewritin g on the partial order on which the functions are defined. These terms must be expressed by constructors of a previously fixed lattice structure, and usually will be set expressions. As in LIFE, functions themselves cannot be used as sort constructing operators. Therefore, the use of partial orders in this paradigm appears simply to be an elegant and compact way to define functions on the given lattice. Functional expressions are ultimately *equated* to specific terms built with these lattice constructors.

We think that the main drawback of the view that all functions are sort constructing operators, and of functional evaluation interpreted as the computation of lower and upper bounds, is that the behaviour of this computation is not clear to the programmer . In order to have an executable specification language based on inclusional theories, we must be able to see from the structure of the specification, i.e. from the axioms expressed as inclusions, how the computation will be done. With the knowledge up to now about deduction with arbitrary transitive relations, we are still too close to the definition of a theorem prover for inclusional theories without any concrete strategy and methodology, and therefore being far away from having an efficient operational semantics for our specification language.

5 Conclusions

Our aim has been to show that inclusions are in principle a flexible and expressive language for preliminary specification. They are a generalization of equations (the conventional specification language for abstract data types) and have a communicatory visual syntax to represent and reason with information about functions.

Very limited uses of inclusions have been proposed in the area of declarative programming. The use of inclusions proposed in this article goes much further. On the one hand, inclusions facilitate the incremental refinement of specifications, from typing information given by upper bounds to partial computing information by lower bounds. They can be successively refined by closer bounds toward equality specifications restricted on singletons. On the other hand, inclusions have a natural visual syntax which can also be used to capture and represent the reasoning process in a compact way.

Because our ultimate goal is to have executable inclusional specifications we have reviewd the state of the art in automatic deduction with inclusions. We have seen that reasoning with inclusions is well captured by bi-rewriting. But the prolific variable subterm chaining inference, which is required for the completeness of the calculus, is an important drawback for finding an efficient operational seman tics for an executable specification language based on inclusions. It is therefore necessary to restrict this general ordered chaining calculus. Future work will focus mainly on restricting the language, finding clear strategies for the deduction in inclusional theories, and defining specific methodologies for solving problems by means of inclusions, that will make us shift towards the specificati on paradigm we pursue.

Acknowledgements

We acknowledge D. Robertson and J. Levy for their collaboration in the work presented in this article.

References

[1] J. Agustí, F. Esteva, P. García, and J. Levy. A Calculus of Refinements: its class of models. In *$1^{\underline{o}}$ Congreso de Programación Declarativa, ProDe'92*, pages 118–126, 1992.

[2] H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. *Journal of Logic Programming*, 16:195–234, 1993.

[3] L. Bachmair and H. Ganzinger. Ordered chaining for total orderings. In A. Bundy, editor, *Automated Deduction — CADE'12*, LNAI 814, pages 435–450. Springer-Verlag, 1994.

[4] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):1–31, 1994.

[5] L. Bachmair and H. Ganzinger. Rewrite Techniques for Transitive Relations. In *Proc., 9th IEEE Symposium on Logic in Computer Science*, pages 384–393, 1994.

[6] M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, and D. Sannella. *Algebraic System Specification and Development. A Survey and Annotated Bibliography.* LNCS 501. Springer-Verlag, 1991.

[7] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[8] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[9] H. F. Hofmann. Requirements engineering: A survey of methods and tools. Research Report 93.05, Institut für Informatik der Universität Zürich, 1993.

[10] J. Hsiang and M. Rusinowitch. Proving refutational completeness of theorem proving strategies: The transfinite semantic tree method. *Journal of the ACM*, 38(3):559–587, 1991.

[11] B. Jayaraman, M. Osorio, and K. Moon. Partial order programming (revisited). In *Proc. Algebraic Methodology and Software Technology (AMAST)*, pages 561–575, 1995.

[12] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[13] J. Levy. *The Calculus of Refinements: a Formal Specification Model Based on Inclusions.* PhD thesis, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, 1994.

[14] J. Levy and J. Agustí. Bi-Rewriting, a Term Rewriting Technique for Monotonic Order Relations. In C. Kirchner, editor, *Rewriting Techniques and Applications*, LNCS 690, pages 17–31. Springer-Verlag, 1993.

[15] J. Levy and J. Agustí. Bi-rewrite systems. *Journal of Symbolic Computation*, 1996. To be published.

[16] J. Levy, J. Agustí, F. Esteva, and P. García. An ideal model for an extended λ-calculus with refinements. Technical Report ECS-LFCS-91-188, Laboratory for Foundations of Computer Science, Edinburgh, 1991.

[17] J. Levy, J. Agustí, and F. Mañá. Functional lattices for taxonomic reasoning. Research Paper DAI-593, Department of Artificial Intelligence, University of Edinburgh, 1992.

[18] D. McAllester, B. Givan, and T. Fatima. Taxonomic syntax for first order inference. In *Proc. of the First Int. Conf. on Principles of Knowledge Representation and Reasoning*, pages 289–300, 1989.

[19] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Journal of Theoretical Computer Science*, 96:73–155, 1992.

[20] P. Mosses. Unified algebras and institutions. In *Principles of Programming Languages Conference*, pages 304–312. ACM Press, 1989.

[21] J. Puigsegur, J. Agustí, and D. Robertson. A Visual Logic Programming Language. In *Proc., 12th IEEE Symposium on Visual Languages*, 1996.

[22] D. Robertson, J. Agustí, J. Hesketh, and J. Levy. Expressing program requirements using refinement lattices. *Fundamenta Informaticae*, 21:163–183, 1994.

[23] W. M. Schorlemmer and J. Agustí. Theorem proving with transitive relations from a practical point of view. Research Report IIIA 95/12, Institut d'Investigació en Intel·ligència Artificial (CSIC), 1995.

[24] J. R. Slagle. Automated theorem proving for theories with built-in theories including equality, partial orderings and sets. *Journal of the ACM*, 19:120–135, 1972.