known recursion patterns, avoiding the use of explicit recursion in situations matching these patterns. In the other hand, they satisfy a number of algebraic laws allowing the transformation of programs in a secure manner, from inefficient clear formulations to more-efficient, less-clear ones [2, 3].

Among these patterns, the function *fold* deserves special mention. The intuition behind it is to *reduce* a polymorphic data structure of type $T\,\alpha$ into a value of type β. In particular, β may coincide with $T\,\alpha$. Many functions getting information from a data structure or modifying it (in the sense of producing a modified copy of it), can be programmed as instances of *fold*. In fact, other higher order functions such as *map* and *filter* can be defined in terms of *fold*. In [6] the *fold* function is characterized as a homomorphism. However, not every function $f : T\,\alpha \longrightarrow \beta$ is a homomorphism. We present some examples of non homomorphic funtions on lists and trees.

Looking for a common pattern for these non homomorphic functions, we propose a modified version of *fold* for lists, named *nhl* in the paper, and another for binary trees, named *nht*, allowing to express some common non homomorphic accesses to these structures. Also some homomorphic functions very complex to express as such, can be defined in a simple way using these new reductions.

A second problem we address in the paper is the relationship between reductions in different data types. We follow an approach that considers a data structure as formed by two components: the data elements and the "structure" itself. We define a hierarchy of data types according to the quantity of "structural" information they possess. We show that some reductions of data structures with more structural information can be expressed as the composition of two homomorphisms: the one that forgets all or part of the structural information, and the one that reduces the target structure.

Prosecuting this idea, we compare the definition of some interesting reductions (inserting and deleting an element in a data structure) for different data structures. We find out the definitions so similar, that all of them could be obtained by transforming the reduction of the structure with no structural information (sets and multisets).

The organization of the paper is as follows: after this introduction, in section 2 we review some of the work done in the last few years on homomorphisms. We present some examples of homomorphisms over lists, sets, multisets and binary trees. In section 3 we present some non homomorphic functions on lists and binary search trees, and introduce our generalization of the *fold* function for these two structures. In particular, it is shown how several variants of insertion and deletion can be easily expressed in terms of the new higher order functions. Section 4 is devoted to presenting the hierarchy of data structures and the transformations between reductions of different data structures. Finally, section 5 provides a short conclusion and summarizes the lines along which this work can be continued in the near future.

2 Homomorphic reductions

Homomorphisms [2, 3, 6] constitute a uniform way of expressing functions that distribute (*promote* in words of R. Bird) their activity both to the data elements and to

the constructors of a data structure. Since most of the interesting data structures have recursive constructors, homomorphisms encapsulate a recursion scheme consisting of applying to all the substructures the same function applied to the main structure. This scheme is used by many functions working on the structure.

Definition 2.1 A function h defined on a data structure of elements of type α, (denoted $T\,\alpha$) is a *homomorphism* if there exist m reductor functions $\oplus_{K_i}$ $(1 \leq i \leq m)$, one for every constructor K_i, such that h is defined as:

$$h :: T\,\alpha \rightarrow \beta$$
$$h\,(K_i\ e_1 \ldots e_{r_i}) = \oplus_{K_i} e'_1 \ldots e'_{r_i}$$

where $e'_j = \begin{cases} e_j & \text{, if } e_j :: \alpha \\ h\ e_j & \text{, if } e_j :: T\,\alpha \end{cases}$

We will use the notation $h = H(\oplus_{K_1} \ldots \oplus_{K_m})$. □

2.1 Homomorphisms on lists

In most functional languages, lists are recursively defined in terms of two constructors: the empty list, denoted [], and the *Cons* constructor denoted $\_ : \_$.

$$list_r\ \alpha = [\,] \mid (:)\ \alpha\ (list_r\ \alpha)$$

We call these lists *forward lists* after [6] and, since they are reduced from right to left, we denote them by $list_r$. The homomorphisms over $list_r$ are then denotated $H(\oplus_:, \oplus_{[\,]})$. When referring to any kind of lists we will use HL instead of H. The symmetric definition (a constructor adding an element to the right of the list) leads us to the corresponding *backwards lists*, denoted $list_l$ as they are reduced from left to right. In FP [1] *sequences* encapsulate both types of lists and mechanisms for accessing and modifying both ends of a list are provided by the language.

The most important higher order function on lists is the reduction. It is called *foldr* (in Miranda and Haskell) for $list_r$, and *foldl* for $list_l$[1]. In general, the operator $(\oplus)$ is not associative . When it is so, and using Bird notation, *fold* is denotated $\oplus/$. It is defined as follows:

Definition 2.2 Let $(\oplus) : \alpha \rightarrow \alpha \rightarrow \alpha$ be associative and with neutral element $id_\oplus$. We define:

$$/ :: (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha$$
$$\oplus/\ [x_1 \ldots x_n] = x_1 \oplus \ldots \oplus x_n$$
$$\oplus/\ [\,] = id_\oplus$$

□

Bird considers lists as monoids (*join-lists* in [6]). The base cases are the empty list and the unit list. The recursive case is the list constructed by appending two

[1] Usually, *foldl* is defined in terms of the forward constructor (:). However, the reduction conceptually corresponds to a definition in terms of the backwards constructor.

lists with the $+\!\!+$ operator which is associative and has the empty list as its neutral element. We will denote them $list_m$.

$$list_m\ \alpha = [\,] \mid [.]\alpha \mid (+\!\!+)\ (list_m\,\alpha)\ (list_m\,\alpha)$$

The main advantage of $list_m$ is its generality: $list_r$ and $list_l$ are particular cases of it. In contrast, they have the drawback of not having free constructors, so the following laws must be settled amongst them:

$$\begin{array}{ll} \forall x, y, z \in list_m & (x +\!\!+ y) +\!\!+ z = x +\!\!+ (y +\!\!+ z) \\ & x +\!\!+ [\,] = [\,] +\!\!+ x = x \end{array}$$

Homomorphisms associated to $list_m$ must preserve the associativity of $+\!\!+$. A mapping h over $list_m$ is a homomorphism if there exists an associative function $\oplus_{+\!\!+} : \beta \to \beta \to \beta$ and functions $\oplus_{[.]} : \alpha \to \beta$ and $\oplus_{[\,]} : \beta$ such that

$$\begin{array}{l} h(l_1 +\!\!+ l_2) = (h\ l_1) \oplus_{+\!\!+} (h\ l_2) \\ h\ [x] = \oplus_{[.]}\ x \\ h\ [\,] = \oplus_{[\,]} \end{array}$$

The main law satisfied by these homomorphisms is the following:

$$h \text{ homomorphism over } list_m \Leftrightarrow h = \oplus/\ .\ f*$$

where $f*$ denotes the expression $map\ f$.

We will abbreviate after Bird $HL(\oplus_{+\!\!+}, \oplus_{[.]}, \oplus_{[\,]})$ by $B(\oplus, f)$, and we will call these homomorphisms B-homomorphisms, being $\oplus = \oplus_{+\!\!+}$, $f = \oplus_{[.]}$ and $id_{\oplus_{+\!\!+}} = \oplus_{[\,]}$[2]. It is inmmediate to express any B-homomorphisms as an $HL(\oplus_{:}, \oplus_{[\,]})$.

A large number of functions over finite lists can be expressed as B-homomorphisms:

$$\begin{array}{rcl} f* & = & B(+\!\!+, [.].f) \\ \oplus/ & = & B(\oplus, id_\alpha) \\ p \lhd & = & B(+\!\!+, p \to [.]; K[\,]) \\ all\ p & = & B(\wedge, p) \\ some\ p & = & B(\vee, p) \\ length & = & B(+, K1) \\ reverse & = & B(\tilde{+\!\!+}, [.]) \\ mergesort & = & B(merge, [.]) \\ id_{[\alpha]} & = & B(+\!\!+, [.]) \end{array}$$

where $K\ x\ y = x$, $merge$ merges two ordered lists, $a\ \tilde{+\!\!+}\ b = b +\!\!+ a$ and

$$(p \to f; g)\ x = \begin{cases} f\ x, & \text{if } p\ x \text{ holds} \\ g\ x, & \text{otherwise} \end{cases}$$

It must be noted that B-homomorphisms are strictly less general than forward list homomorphisms $HL(\oplus_{:}, \oplus_{[\,]})$ as the following example shows.

[2] When $id_{\oplus_{+\!\!+}}$ doesn't exist, the domain excludes the empty list.

Example 2.1 We wish to compute the integer represented by a list of digits, each one in the range 0..9, being the most significant digit the rightmost one.

$$\begin{array}{rcl} integer & = & HL(\oplus_{:}, 0) \\ & & \textbf{where } x \oplus_{:} s = s * 10 + x \end{array}$$ □

This function cannot be expressed as a B-homomorphism, because there is no way (with a homomorphism) of recording the number of zeroes in the middle of the computation.

2.2 Homomorphisms over other usual monoids

Homomorphisms can be easily generalized to other data structures. The main requirement for them is that the reductor operators must preserve the constructor laws.

Multisets are commutative monoids. Their algebraic definition is as follows:

$$mset\ \alpha = \{\,\} \mid \{.\}\alpha \mid (\mathbf{U})\ (mset\ \alpha)\ (mset\ \alpha)$$

where the binary constructor $\mathbf{U}$ satisfies the associative and commutative laws. As a consequence, homomorphisms h over multisets must provide an associative, commutative reductor $\oplus_{\mathbf{U}}$ and satisfy:

$$h(m_1 \mathbf{U} m_2) = (h\ m_1) \oplus_{\mathbf{U}} (h\ m_2)$$

For instance, the cardinal of a multiset, or the sum of all their elements are homomorphisms. In both cases, $\oplus_{\mathbf{U}}$ is the $+$ operator over natural numbers.

Sets are commutative, idempotent monoids. Their algebraic definition is:

$$set\ \alpha = \{\,\} \mid \{.\}\ \alpha \mid (\cup)\ (set\ \alpha)\ (set\ \alpha)$$

where the binary constructor $\cup$ satisfies the associative, commutative and idempotent laws. As a consequence, homomorphisms h over sets must provide an associative, commutative, idempotent reductor $\oplus_{\cup}$ and satisfy:

$$h(s_1 \cup s_2) = (h\ s_1) \oplus_{\cup} (h\ s_2)$$

Functions computing the maximum element of a set, or the greatest common divisor of all of them can be expressed as homomorphisms over sets. Also, the predicates using $all\ p$, such as the section $(\subset S)$ for a given S, defined as $(\subset S) = all\ (\in S)$, or those using $some\ p$, such as the section $(x \in)$, defined as $(x \in) = some\ (x =)$, are homomorphisms.

2.3 Homomorphisms over binary trees

In this section we introduce the algebraic definition of binary trees and their associated homomorphisms:

$$Tree\ \alpha = \triangle \mid \text{⋔}\,(Tree\ \alpha)\ \alpha\ (Tree\ \alpha)$$

Then, homomorphisms over binary trees need the existence of two reductors.

Definition 2.3 We say that a function h : Tree $\alpha \to \beta$ is a *homomorphism* if there exist reductor functions $\oplus_{\curlywedge} : \beta \to \alpha \to \beta \to \beta$ and $\oplus_{\triangle} : \beta$ such that:

$$\begin{aligned} h\,\triangle &= \oplus_{\triangle} \\ h\,(\curlywedge\, l\, x\, r) &= \oplus_{\curlywedge}\,(h\,l)\,x\,(h\,r) \end{aligned}$$

We will use the notation $h = HT(\oplus_{\curlywedge}, \oplus_{\triangle})$ to denotate these homomorphisms. □

A number of homomorphic functions over lists have their homolog ones over trees:

$$\begin{aligned} f* &= HT(\oplus_{\curlywedge}, \triangle) \\ &\quad \text{where } \oplus_{\curlywedge}\, l\, x\, r = \curlywedge\, l\,(f\,x)\, r \\ all\ p &= HT(\bigwedge, True) \\ &\quad \text{where } \bigwedge\, b_l\, x\, b_r = (p\,x) \wedge b_l \wedge b_r \\ some\ p &= HT(\bigvee, False) \\ &\quad \text{where } \bigvee\, b_l\, x\, b_r = (p\,x) \vee b_l \vee b_r \\ size &= HT(\oplus_{\curlywedge}, 0) \\ &\quad \text{where } \oplus_{\curlywedge}\, i_l\, x\, i_r = i_l + 1 + i_r \\ height &= HT(\oplus_{\curlywedge}, (-1)) \\ &\quad \text{where } \oplus_{\curlywedge}\, i_l\, x\, i_r = 1 + max\ i_l\ i_r \\ flatten &= HT(+\!\!+_3, [\,]) \\ &\quad \text{where } +\!\!+_3\, l\, x\, r = l +\!\!+ [x] +\!\!+ r \\ id_{Tree\,\alpha} &= HT(\curlywedge, \triangle) \end{aligned}$$

3 Non homomorphic reductions

In this section, two higher order functions are defined —one for forward lists and another one for binary trees— that generalize common non homomorphic functions on these structures.

The underlying idea on these functions comes from realizing that many functions on a structure, first search along the structure looking for some property to hold; then some special treatment is done, and then the rest of the structure is ignored. In terms of reduction, this idea can be interpreted as follows: the structure is divided into two parts, the active one and the passive one with the aid of one or more predicates. Reducing the active part amounts to say that the reductor operator does the search, and if the search succeeds applies the special treatment. Reducing the passive part consists of applying a different reductor operator, which in many cases simply copies the structure.

We are specially interested in those operations that return a structure of the same type as the original one, i.e. we pay attention to insertion and deletion operations.

3.1 Lists

We study ordered and unordered lists, with and without multiple copies of elements. For *ordered lists without repetitions* both operations are homomorphisms. Their definitions follow.

Example 3.1

```
insert x = B(⊕,f)
           where f y = [x,y] , if x<y
                     = [y,x] , if x>y
                     = [y]   , if y=x
                 (l1++[x1]) ⊕ ([x2]++l2) = l1++[x2]++l2 , if x1=x
                                         = l1++[x1]++l2 , if x2=x

delete x = (x≠) ◁
```

□

We see that expressing *insert* as a homomorphism is little intuitive: we first insert a copy of x either to the right or to the left of *each* element. Then we remove all the copies except the one that must remain.

For *unordered lists without repetitions* both operations are also homomorphisms. In fact, *delete* is the same function of example 3.1. The *insert* function is the following homomorphism:

Example 3.2

```
insert x = B(⊕, f)
           where f y = [x,y] , if x≠y
                     = [y]   , otherwise
                 l1 ⊕ ([x2]++l2) = l1++l2
```

□

Again, the technique consists of inserting a copy of x for every element in the list not equal to x, and then removing all the copies except the leftmost one.

However, for *lists with repetitions* some operations are not homomorphisms. Insertion of a new copy indeed it is, and can be defined by slightly modifying the **otherwise** clause of example 3.2. But to *delete* the leftmost copy of a value x is not a homomorphism as the following lemma shows.

Lemma 3.1 The function that deletes the leftmost copy of a value x in an ordered list with repetitions *is not* a homomorphism.

Proof. Let us assume that h implements that function an h is a homomorphism. Then, there exists a reductor $\oplus$ such that $h\,(y{:}ys) = y \oplus h\ ys$.

Let xs be any list without occurrences of the value x. Then, $h\ xs = xs$, and $h\,(x{:}xs) = xs$. But $h\,(x{:}xs) = x \oplus h\ xs$, then $x \oplus xs = xs$. As a consequence, we have $h\,(x{:}x{:}xs) = x \oplus h\,(x{:}xs) = x \oplus xs = xs$, in contradiction with what the hypothesis says: that h only deletes one copy of x. □

For the case of unordered lists the proof is very similar, but it must be noted that $h\ xs$ needs not be equal to xs but to a permutation of the elements of xs.

Despite the fact that some of these functions are not homomorphisms or, if they are, it is difficult to express them as such, it is possible to define a homomorphism, named *split*, which does part of the work of *insert* and *delete*. What remains to be done is simple enough to make the programming of these two functions an easy task. The homomorphism *split*, defined below, applies a predicate q to the list and splits it into two lists in such a way that the first element of the second list (in the case the list is not empty) is the leftmost element of the original list satisfying q. For example, *split* (>5) [1,3,5,7,4]=[[1,3,5],[7,4]].

Definition 3.1

```
split q = B(⊕,f)
          where f x = [[],[x]] , if q x
                    = [[x],[]] , otherwise
        [l1,[]] ⊕ [l2,m2]  = [l1++l2, m2]
        [l1,m1] ⊕ [l2,m2]  = [l1,m1++l2++m2]
```

□

Let us remark that the neutral element of operation $\oplus$ in this case is [[],[]]. With the aid of *split*, deletion of the leftmost copy of x can be programmed as:

```
delete x = del.(split (x=))
           where del [l1 , x:l2] = l1 ++l2
                 del [l1 , []]   = l1
```

Unfortunately, *split* is not a solution for all the problems concerning insertion and deletion. For instance, the homomorphism that deletes all the copies of an element is not easily constructed with *split*. Besides, it is not clear how to generalize *split* to other data structures. So, we define a more general non homomorphic reduction for lists based on the idea presented at the beginning of this section.

Definition 3.2 Non Homomorphic reduction function for Lists (*nhl*):

```
nhl q ⊕a ⊕d b []= b
nhl q ⊕a ⊕d b (x:xs) = x ⊕a (nhl q ⊕ ⊕d b xs)
                       where ⊕ = ⊕a , if ¬ (q x)
                               = ⊕d , otherwise
```

□

Reductor operators $\oplus_a$ (active) and $\oplus_d$ (default or passive) correspond to the (:) constructor, and b corresponds to the [] constructor. We will use the following notation which mimics that used for homomorphisms:

$$\text{nhl } q \oplus_a \oplus_d b = \text{NHL}(q,\oplus_:^a,\oplus_:^d,\oplus_{[]})$$

The function *nhl* "inserts" an active reductor after each element of the list including the first, if any, satisfying the q predicate. From that point to the end of the list, *nhl* inserts the passive reductor. The following expression pictures this idea:

$$\text{nhl q } \oplus_a \oplus_d \text{ b } [x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_{n-1}, x_n] =$$
$$x_1 \oplus_a (x_2 \oplus_a (\ldots x_i \oplus_a (x_{i+1} \oplus_d (\ldots x_{n-1} \oplus_d (x_n \oplus_d b) \ldots)$$

being $i = min\{1 \ldots n\}$ such that $q\ x_i$ holds.

Trivially, any homomorphism $HL(\oplus_:, \oplus_{[]})$ on forward lists can be expressed as the non homomorphic reduction $NHL(q, \oplus_:^a, \oplus_:^d, \oplus_{[]})$ making $\oplus_:^a = \oplus_:^d = \oplus_:$, and being q any boolean function. Some awkward homomorphic functions can be elegantly expressed as a non homomorphic reduction. For instance, inserting a value x in an ordered list without repetitions is now:

Example 3.3

```
insert x = nhl (≥x) ⊕ (:) []
           where y ⊕ ys = x:y:ys , if x < y
                        = y:x:[] , if ys=[] ∧ x > y
                        = y:ys   , otherwise
```

□

We assume that the argument list is non empty. If it is empty, then *insert* x []= [x]. The other insertion functions can be similarly programmed. For deletion, the following function is enough for all the cases presented in this section.

Example 3.4

```
delete x = nhl (=x) ⊕ (:) []
           where y ⊕ ys = ys    , if x = y
                        = y:ys , otherwise
```

□

Let us note in these examples that the passive reductor is (:) and the base case is [], so they reconstruct (or copy) the original list.

3.2 Binary trees

For these structure, we only consider binary search trees without multiple occurrences of elements. The treatment of multiple occurrences does not present special problems in this structure.

The non homomorphic reduction we are proposing here encapsulates as a particular case the usual recursion scheme on binary search trees: the search propagates either to the left or to the right subtree, never to both of them at the same time.

Definition 3.3 Non Homomorphic reduction function for Trees (*nht*):

```
nht ql qr ⊕a ⊕d b △ = b
nht ql qr ⊕a ⊕d b (Ѧ l x r) = ⊕a (nht ql qr ⊕l ⊕d b l) x (nht ql qr ⊕r ⊕d b r)
                              where ⊕l = ⊕a , if ¬ (ql x)
                                       = ⊕d , otherwise
                                    ⊕r = ⊕a , if ¬(qr x)
                                       = ⊕d , otherwise
```

□

Predicates ql and qr trigger the change from active to passive mode in the corresponding subtrees, the first time they hold. The active part of the tree extends from the root to the nearest elements satisfying ql or qr. The figure 1 shows an example of execution.

Reductor operators $\oplus_a$ and $\oplus_d$ correspond to the Ѧ constructor, and the base case b corresponds to the △ constructor. We will use then the following notation:

$$\text{nht } ql\ qr \oplus_a \oplus_d b = \text{NHT}(ql,qr,\oplus_Ѧ^a,\oplus_Ѧ^d,\oplus_△)$$

Like in the case of lists, we could express the insertion and deletion as homomorphisms. Unfortunately, the resulting functions are not so easy to understand. By using the non homomorphic reduction, we see below that the result is very simple.

Example 3.5

```
insert x = nht (≤x) (≥x) ⊕ Ѧ △
           where ⊕ l y r = Ѧ l y (Ѧ △ x △) , if r=△ ∧ x>y
                         = Ѧ (Ѧ △ x △) y r , if l=△ ∧ x<y
                         = Ѧ l y r         , otherwise
```

□

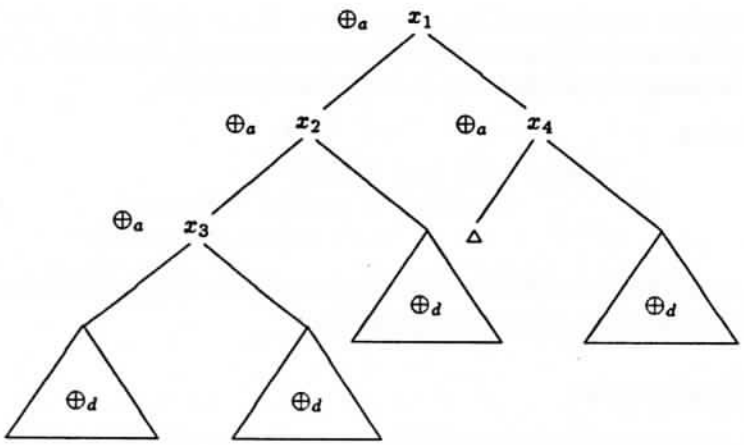

Figure 1: An example of *nht* execution.

Example 3.6

$$\begin{aligned}\text{delete x} = \text{nht } (\leq x)\ (\geq x) \oplus \text{\Large\Lightning}\ \triangle\\ \text{where } \oplus\ l\ y\ r \ &= \text{\Large\Lightning}\ l\ y\ r, \text{ if } x \neq y\\ \oplus\ \triangle\ y\ r \ &= r\\ \oplus\ l\ y\ r \ &= \text{\Large\Lightning}\ (\text{delmax } l)\ (\text{selmax } l)\ r\end{aligned}$$

□

The functions *delmax* and *selmax*, respectively deletes and selects the maximum element of a tree and could also be expressed as NHT reductions.

4 Transformations between reductions

Any data structure, in particular those studied in the previous sections, may be considered as composed of two disjoint pieces of information:

- Its *contents*, i.e. the data elements stored in it. We will refer to them as its set *set* α or multiset *mset* α of elements of type α.
- The *structure* itself, i.e. the spatial disposition of the elements inside the structure. We will call *structural information* to this aspect of the data structure.

According to this view, we classify data structures based on the *amount* of structural information they contain. We establish a hierarchy in which structures on top of it contain the least structural information (i.e. sets and multisets), and structures at the bottom contain the most. The aspect of this hierarchy is shown in figure 2.

Let us note that, in this hierarchy, ordered lists are located above unordered lists. The idea is to be able to define unique homomorphisms from structures located at low levels of the hierarchy to structures located at higher levels. The meaning of these homomorphisms is "lost of structural information". They are indicated in the figure as filled arrows. The homomorphism going from unordered lists to ordered ones means "sort the list",and it is an n to 1 mapping. The dotted arrows of figure 2

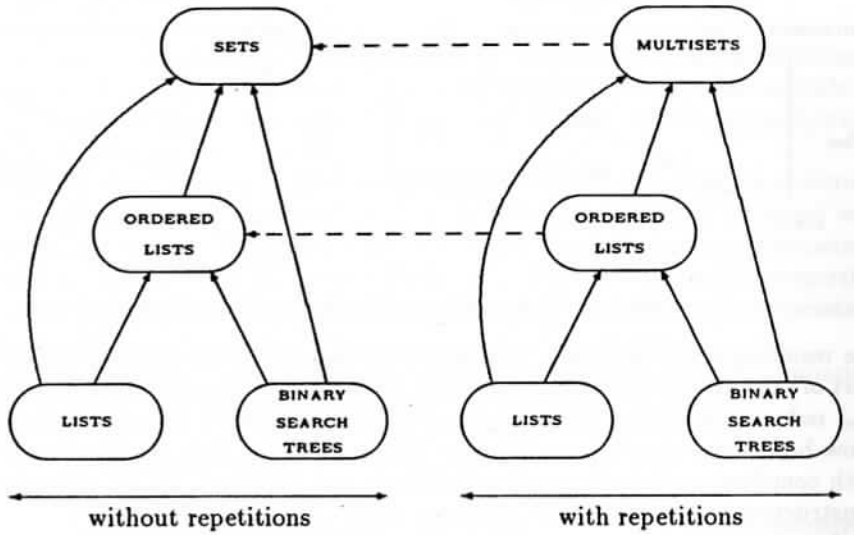

Figure 2: data structure hierarchy

correspond to the homomorphisms "eliminate multiple copies of all the elements". In principle, as they represent lost of information about the data elements, we are not very interested on them.

To speak properly about homomorphisms it is mandatory to say which are the correspondent operations in both domains. For instance, in the homomorphisms from unordered lists to ordered ones, $+\!\!+$ in the first domain is the homolog of *merge* in the second domain. Sometimes we need to add appropriate operations in one of the domains to be able to define the homomorphism. For instance, the one from binary search trees without repetitions to sets can be defined:

$$\begin{aligned}\psi_{ts} = \text{HT}(\oplus_{\text{\Large\Lightning}}, \{\})\\ \text{where } \oplus_{\text{\Large\Lightning}}\ sl\ x\ sr = sl \cup \{x\} \cup sr\end{aligned}$$

So, we define an operation on sets $\cup^{\{\}}\cup : \mathit{set} \rightarrow \mathit{elem} \rightarrow \mathit{set} \rightarrow \mathit{set}$ that joins two sets and one element producing a new set. Then, this new operation corresponds in the homomorphism to the binary tree constructor $\text{\Large\Lightning}$.

The homomorphism going from unordered lists with possible repetitions to multisets is defined as follows:

$$\psi_{lm} = \text{HL}(\cup, \{.\}, \{\!|\ |\!\})$$

In this case we use the normal constructors of multisets as homolog operations of the constructors of list_m.

Many of the data structures reductions studied in previous sections depend on the contents but not on the structural information. This is the case when computing the maximum element of a binary tree or the sum of all the elements of a list. In these cases, the reduction to apply can be obtained as the composition of two homomorphisms: the one abstracting the structural information and the one reducing

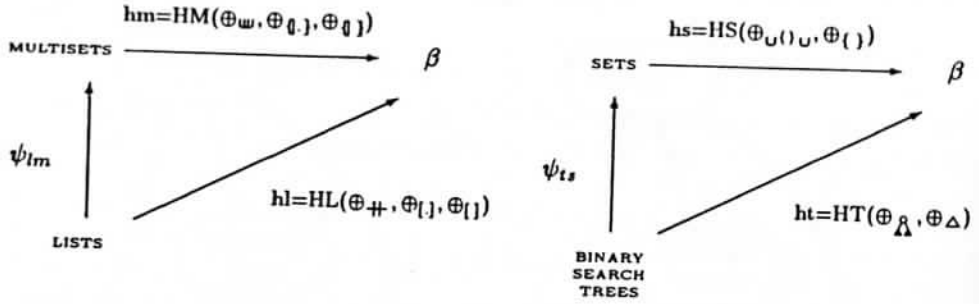

Figure 3: Composition of homomorphic reductions

the resulting set or multiset. We denote by ψ_{xy} the homomorphism abstracting (a part of) the structural information from a structure x to a structure y. For instance, ψ_{lm} reduces unordered lists with repetitions to multisets. The diagrams of figure 3 show how homomorphisms compose. There, $HM(\oplus_\omega, \oplus_{\{.\}}, \oplus_{\{|\}})$ reduces multisets with constructors homolog to those of lists, and $HS(\oplus_{\cup(|)\cup}, \oplus_{\{\}})$ reduces sets with constructors homolog to those of binary trees.

Let us imagine a homomorphism $hl = HL(\oplus_{\#}, \oplus_{[.]}, \oplus_{[]})$ from lists to a value of type β which is independent of the structural information of lists. If we express hl as a composition of homomorphisms, the reductor operators relate each other in the following way:

$$\begin{array}{rcll} \oplus_{[]} & = & hl\,[\,] & = (hm.\psi_{lm})\,[\,] = hm\,\{|\} = \oplus_{\{|\}} \\ \oplus_{[.]}x & = & hl\,[x] & = (hm.\psi_{lm})\,[x] = hm\,\{|x|\} = \oplus_{\{.\}}x \\ hl\,xs \oplus_{\#} hl\,ys & = & hl\,(xs \# ys) & = (hm.\psi_{lm})\,(xs \# ys) \\ & & & = hm\,(\psi_{lm}\,xs \mathbf{U} \psi_{lm}\,ys) \\ & & & = (hm.\psi_{lm})\,xs \oplus_{\omega} (hm.\psi_{lm})\,ys \\ & & & = hl\,xs \oplus_{\omega} hl\,ys \end{array}$$

Then, it holds that $\oplus_{[]} = \oplus_{\{|\}}, \oplus_{[.]} = \oplus_{\{.\}}$ and $\oplus_{\#} = \oplus_{\omega}$. Let us note that $\oplus_{\omega}$ is associative and commutative as it is a reductor operator of multisets, and so homolog to **U** constructor.

Using the constructors for sets that are homolog to those of binary trees, we can define the reduction on sets as the following homomorphism $hs = HS(\oplus_{\cup(|)\cup}, \oplus_{\{\}})$

$$\begin{array}{rcl} hs\,\{\} & = & \oplus_{\{\}} \\ hs\,(\cup^{(|)}\cup\, s1\; x\; s2) & = & \oplus_{\cup(|)\cup}\,(hs\,s1)\;x\;(hs\,s2) \end{array}$$

If we wish to reduce a tree into a value of type β by using a structure-independent homomorphism $ht = HT(\oplus_{\Lambda}, \oplus_{\triangle})$, we can do it by composing the two homomorphisms ψ_{ts} and hs. The reductor operations relate each other as follows:

$$\begin{array}{rcll} \oplus_{\triangle} & = & ht\,\triangle & = (hs.\psi_{ts})\,\triangle = hs\{\,\} = \oplus_{\{\}} \\ \oplus_{\Lambda}\,(ht\,l)\;x\;(ht\,r) & = & ht(\Lambda\, l\; x\; r) & = (hs.\psi_{ts})\,(\Lambda\, l\; x\; r) \\ & & & = hs(\cup^{(|)}\cup(\psi_{ts}\,l)\;x\;(\psi_{ts}\,r)) \\ & & & = \oplus_{\cup(|)\cup}\,((hs.\psi_{ts})\,l)\;x\;((hs.\psi_{ts})\,r) \\ & & & = \oplus_{\cup(|)\cup}(ht\,l)\;x\;(ht\,r) \end{array}$$

That is, $\oplus_{\Lambda} = \oplus_{\cup(|)\cup}$ and $\oplus_{\triangle} = \oplus_{\{\}}$.

In other cases, the reduction of the data structure depends only on the structural information, or on both the contents and the structural information. An example of the first type is to compute the height of a tree. An example of the second type is the example 2.1 for lists.

In particular, very interesting are those reductions giving as a result a structure of the same type as the one being reduced, as it is the case with the insertion and deletion operations studied in section 3. In these cases, the reductor operator cannot add essential properties (such as commutativity or idempotency) over the properties already satisfied by the constructor. Normally, the reductor will be a slight variation of the constructor.

We wish to detect common patterns in these types of reductions by comparing how they behave for different data structures.

4.1 Lists

We use the free constructors on forward lists [] and _ : _ and consider unordered multiple-copies ones. Abstracting the structural information we get multisets. To establish a homomorphism we add the _↪_ operation to multisets. It adds an element to a multiset and satisfies the following *permutative* law:

$$x{\hookrightarrow}(y{\hookrightarrow}m) = y{\hookrightarrow}(x{\hookrightarrow}m)$$

The reduction homomorphism for multisets is defined then:

$$\begin{array}{rcl} hm\,\{|\} & = & \oplus_{\{|\}} \\ hm\,(x{\hookrightarrow}m) & = & x \oplus_{\hookrightarrow} hm\,m \end{array}$$

where $\oplus_{\hookrightarrow}$ must satisfy the permutative law.

Let us consider inserting in an unordered multiple-copies list. We choose to insert at the end of it:

$$\begin{array}{l} \text{insert}_l\ \text{x} = \text{HL}(\oplus_{:}, [\,]) \\ \qquad \textbf{where}\ \text{y} \oplus_{:} [\,] = \text{y:x:[]} \\ \qquad\qquad \text{y} \oplus_{:} \text{ys} = \text{y:ys} \end{array}$$

If we interpret to insert "at the end" of a multiset as inserting before inserting any other element, we obtain:

$$\begin{array}{l} \text{insert}_m\ \text{x} = \text{HM}(\oplus_{\hookrightarrow}, \{|\}) \\ \qquad \textbf{where}\ \text{y} \oplus_{\hookrightarrow} \{|\} = \text{y}{\hookrightarrow}\text{x}{\hookrightarrow}\{|\} \\ \qquad\qquad \text{y} \oplus_{\hookrightarrow} \text{m} = \text{y}{\hookrightarrow}\text{m} \end{array}$$

The structure of boths reductions is identical. This similarity remains when we insert in ordered lists by using the non homomorphic reductor nhl (see example 3.3).

Deletion of all the copies of an element in ordered lists with repetitions is the following homomorphism:

$$\begin{array}{l} \text{delete}_l\ \text{x} = \text{HL}(\oplus_{:}, [\,]) \\ \qquad \textbf{where}\ \text{y} \oplus_{:} \text{m} = \text{m} \quad \textbf{, if x=y} \\ \qquad\qquad\qquad\quad = \text{y:m} \quad \textbf{, otherwise} \end{array}$$

The version for multisets is:

$$\begin{aligned} \text{delete}_m\ x &= \text{HM}(\oplus_{\hookrightarrow}, \{\!|\,|\!\}) \\ &\quad \textbf{where } y \oplus_{\hookrightarrow} m = m \quad \textbf{, if } x{=}y \\ &\qquad\qquad\ = y \hookrightarrow m \textbf{ , otherwise} \end{aligned}$$

The structures of both reductions are again identical. The main difference is that the multiset reduction can be done in any order, taking into account the permutative law that $\hookrightarrow$ satisfies. In forward lists, reduction will always be done from right to left. We obtain the same similarity if we compare *delete* for lists without repetitions with *delete* in a set. However, *delete* the first copy in lists with repetitions is not a homomorphism as it has been shown in lemma 3.1. We can use here the non homomorphic reduction:

$$\begin{aligned} \text{delete } x &= \text{nhl } (=x)\ \oplus_a\ (:)\ [\,] \\ &\quad \textbf{where } y \oplus_a ys = ys \quad \textbf{, if } x{=}y \\ &\qquad\qquad\ = y{:}ys \textbf{ , otherwise} \end{aligned}$$

We observe great similarity with deletion of all the copies. The role of the $(= x)$ predicate consists only of stopping the deletion of copies once the first one has been deleted.

4.2 Binary search trees

We consider trees without repetitions and sets with the special constructors that are homomorphic to the tree constructors.

In example 3.5 it is shown how to insert in a binary search tree. The corresponding version for sets is:

$$\begin{aligned} \text{insert}_s\ x &= \text{HS}(\oplus_{\cup\{\}\cup}, \{\,\}) \\ &\quad \textbf{where } \oplus_{\cup\{\}\cup}\ s1\ y\ s2 = \cup^{\{\}}\cup\ s1\ y\ (\cup^{\{\}}\cup\ \{\,\}\ x\ \{\,\})\ \textbf{, if } s2{=}\{\,\} \\ &\qquad\qquad\qquad\qquad\ = \cup^{\{\}}\cup\ (\cup^{\{\}}\cup\ \{\,\}\ x\ \{\,\})\ y\ s2\ \textbf{, if } s1{=}\{\,\} \\ &\qquad\qquad\qquad\qquad\ = \cup^{\{\}}\cup\ s1\ y\ s2 \qquad\qquad\ \textbf{, otherwise} \end{aligned}$$

It is obvious to see that this is only one of the possibilities. We can freely permute elements in a set. We have chosen the possibility that takes into account the order present in search trees. Then, we can say that the reduction of a search tree can be obtained by adding to the reduction of the corresponding set some hypothesis about the order of the elements (i.e. by taking into account the structural information of search trees).

In example 3.6 it is shown how to delete an element from a binary search tree. The corresponding version for sets is:

$$\begin{aligned} \text{delete}_s\ x &= HS(\oplus_{\cup\{\}\cup}, \{\,\}) \\ &\quad \textbf{where } \oplus_{\cup\{\}\cup}\ s1\ y\ s2 = \cup^{\{\}}\cup\ s1\ y\ s2, \textbf{ if } x{\neq}y \\ &\qquad\qquad\qquad\ \oplus_{\cup\{\}\cup}\ \{\,\}\ y\ s2 = s2 \\ &\qquad\qquad\qquad\ \oplus_{\cup\{\}\cup}\ s1\ y\ s2 = \cup^{\{\}}\cup\ (\text{delone } s1)\ (\text{selone } s1)\ s2 \end{aligned}$$

where *delone s1* deletes the element that *selone s1* selects. The similarity is very clear again. The order hypothesis in search trees forces us to select always the maximum of the left subtree. In sets we are free to choose any element.

5 Conclusion

We have studied in detail two kinds of higher order functions which reduce two common data structures, lists and binary trees, either in a homomorphical or in a non homomorphical way. They encapsulate a powerful recursion scheme so that many usual functions on these structures can be easily expressed in terms of them. Present work is devoted to generalize this idea to other data structures.

Also, we have presented a way of comparing reductions in a hierarchy of data structures differing in the degree of structural information they have. This idea constitutes the basis for transforming algorithms expressed in terms of the structures with less structural information to algorithms for the structures with more structural information.

Acknowledgements

We would like to express our gratitude to Margarita Bradley and Pedro Palao for their help while preparing this paper and to an anonymous referee who pointed out some errors in the draft version.

References

[1] J.W. Backus. Can functional programming be liberated from the Von Neumann style? *Communications of the ACM*, 21:613–641, 1978.

[2] R. S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer Verlag. NATO ASI Series, vol. F36, 1987.

[3] R. S. Bird. Lectures on constructive functional programming. In *Constructive Methods in Computer Science*, pages 151–216. Springer Verlag. NATO ASI Series, vol. F55, 1989.

[4] A. J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, 1992.

[5] M. P. Jones. *GOFER.* Oxford University Computing Laboratory, 1992.

[6] G. Malcolm. Homomorphisms and promotability. In *Mathematics of Program Construction. LNCS 375*, pages 335–347. Springer-Verlag, 1989.

[7] D.A. Turner. Miranda: A non-strict functional language with polymorphic types. In *LNCS 201*, pages 1–16, Berlin, 1985. Springer-Verlag.

[8] A. Wikström. *Functional Programming Using Standard ML.* Prentice-Hall, 1987.

Amalgamating Language and Meta-Language for Composing Logic Programs

Antonio Brogi, Chiara Renso, Franco Turini
Dipartimento di Informatica, Università di Pisa
Corso Italia, 40 - 56125 Pisa, Italy
{brogi,renso,turini}@di.unipi.it

Abstract

Logic programming is extended with expressions of the form *A in Pexp* both in top-level goals and in clause bodies. *A in Pexp* is a meta-level feature that denotes the truth of a formula *A* with respect to a "virtual" set of clauses denoted by the program expression *Pexp*. *Pexp* involves named collections of clauses and composition operations over them. Both the operational and fixpoint semantics of the language are given along with its meta-level definition. A set of examples provides evidence of expressiveness of the language.

1 Introduction

Our long term goal is the re-construction of knowledge representation schemata, reasoning systems, and programming systems on top of the logic programming kernel. There are two reasons for this endeavor. On one hand, bringing together different proposals in a common setting in order to integrate them, and, in the othe and, exploiting the simple semantics of logic programming to set a common semantic foundation for different proposals, that allow a deeper comparison among them.

We claim that a way to get closer to this long-term goal is to provide logic programming with a simple but powerful extension capable of allowing the re-construction of reasoning paradigms on one side, and of preserving the kernel semantics on the other side. The extension consists of the possibility of using goals of the form *A in Pexp* both in top-level queries and in clause bodies, where *Pexp* is an expression involving named collections of clauses (theories) and composition operations over them. Simply stated, we allow the computation of a goal with respect to a *virtual* set of clauses denoted by the expression.

The composition operations provide a rich way of combining the knowledge represented in the theories. The extension we propose consists of amalgamating the language in which the program are written (object language) and the language of program expressions. This way it is possible to express the manipulation of the program (combination of collections of clauses) from within the program itself.

We may claim progress with respect to our long-term goal only as far as our linguistic extension requires only a conservative extension of the semantics of the logic programming kernel. The goal of this paper is exactly the presentation of the semantics of the amalgamated language, which is shown to be a conservative extension of the basic semantics.

The semantics of the amalgamated language is given in the most classical way. We define both a top-down semantics (goal-driven operational semantics) via a set of inference rules, and a bottom-up (fixpoint) semantics via an extension of the standard immediate consequence operator for logic programs. The well founded meaning of the amalgamated language is provided by the equivalence of the two semantics.

The proposal of this paper builds upon a long stream of research on the use of algebras of logic programs as a means for implementing common sense reasoning and program structuring [3, 7, 8, 9]. The novelty here is *amalgamation*: For the first time we are able to allow program expressions within clause bodies (and not only in top-level goals), and to handle the formal semantics of the resulting amalgamated language.

A second contribution of the paper is to show that the proposed language extension can be entirely captured by meta-logical axioms. In practice, we show how the transition rules of the top-down semantics of the language can be directly translated into meta-level axioms of the form $demo(Pexp, Goal) \leftarrow Body$. The translation is useful in many respects, since it provides an executable specification of the amalgamated language and shows the expressiveness of meta-logic itself.

The plan of the paper follows. Section 2 introduces syntax and semantics of the amalgamated language and states the equivalence of operational and fixpoint semantics, which provides confidence in the well foundedness of the extension. A meta-logical implementation of the language is also presented. Section 3 presents a few simple reconstructions of knowledge representation and reasoning systems in order to provide evidence for the expressiveness of the extension. In short, we present a "translation" of a few formalisms proposed in the literature into our amalgamated logic programming. The Conclusions discuss work done towards a more realistic implementation and open issues.

2 The Amalgamated Language

2.1 Syntax

We first present the syntax of the amalgamated language. We consider a set of meta-level operations for composing definite logic programs, originally introduced in [3, 8]: *Union* ($\cup$), *Intersection* ($\cap$), *Encapsulation* ($*$), *Import* ($\triangleleft$) and *Restriction* ($\prec$). These operations lead to the following language of program expressions:

$$Exp ::= Program \mid Exp \cup Exp \mid Exp \cap Exp \mid (Exp)^* \mid Exp \triangleleft Exp \mid Exp \prec Program$$

where *Program* is a name of a collection of clauses.

The language of program expressions has been largely investigated both from the theoretical point of view and from the application point of view (e.g. in [3, 8, 10, 14]). It is worth noting that the language of program expressions has been employed as a meta-language for composing object programs written in a separate language, namely object definite programs.

In this paper, we propose a single language amalgamating the language in which programs are written (object language) and the language of program expressions (meta-language). Namely, programs are extended definite programs that may contain meta-level calls to program expressions in clause bodies. More precisely, a program is a finite set of extended definite clauses of the form $A \leftarrow B_1, \ldots, B_n$ where each B_i is either an atomic formula or a meta-level formula of the form B *in* E where B is an atomic formula and E a program expression.

2.2 Top-down Semantics

We now present a top-down semantics for the amalgamated language by means of a set of inference rules. The operational characterization of the language is expressed by directly extending the standard notion of SLD refutation.

The standard notion of SLD derivation may be defined by means of inference rules of the form

$$\frac{Premise}{Conclusion}$$

with the meaning that *Conclusion* holds whenever *Premise* holds. We write $P \vdash G$ if there exists a refutation for a goal G in a program P. Let us first present four rules that model standard SLD resolution. (For the sake of simplicity substitutions are omitted here.)

$$\frac{}{P \vdash empty}$$

$$\frac{P \vdash G_1 \wedge P \vdash G_2}{P \vdash (G_1, G_2)}$$

$$\frac{P \vdash (A \leftarrow G) \wedge P \vdash G}{P \vdash A}$$

The first rule states that the empty goal is solved in any program P. The second rule states that a conjunction of goals (G_1,G_2) is solved in a program P if both goals are solved in P. The third rule states that, to solve an atomic goal A, choose a clause from P and recursively solve the body in P. Finally, program clauses are represented by means of the following rule:

$$\frac{P \text{ is a program} \wedge A \leftarrow G \in ground(P)}{P \vdash (A \leftarrow G)}$$

The derivation relation $\vdash$ can be generalized to the case of program compositions in a simple way. Namely, each composition operation is modelled by adding new inference rules. Each new rule provides a way of proving that a clause belongs to the virtual program denoted by the expression E, i.e. if $E \vdash (A \leftarrow G)$. From now on we will say $A \leftarrow G$ belongs to E as a shorthand for the clause $A \leftarrow G$ belongs to the virtual program: $\{A \leftarrow G \mid E \vdash (A \leftarrow G)\}$.

$$\frac{P \vdash (A \leftarrow G)}{P \cup Q \vdash (A \leftarrow G)}$$

$$\frac{Q \vdash (A \leftarrow G)}{P \cup Q \vdash (A \leftarrow G)}$$

The two rules above deal with the *union* operation, stating that a clause $A \leftarrow G$ belongs to the program expression $P \cup Q$ if it belongs either to P or to Q.

$$\frac{P \vdash (A \leftarrow G_1) \wedge Q \vdash (A \leftarrow G_2)}{P \cap Q \vdash (A \leftarrow G_1, G_2)}$$

This rule states that a clause belongs to the *intersection* of two program expressions P and Q if there is a clause $A \leftarrow G_1$ in P and a clause $A \leftarrow G_2$ in Q such that $G = (G_1, G_2)$.

$$\frac{P \vdash A}{P^* \vdash (A \leftarrow empty)}$$

This rule states that the *encapsulation* P^* of P contains a unit clause $A \leftarrow empty$ for each atom A which is provable in P.

$$\frac{P \vdash (A \leftarrow G, F) \wedge Q \vdash F}{P \triangleleft Q \vdash (A \leftarrow G)}$$

This rule deals with the *import* operation stating that the clauses in $P \triangleleft Q$ are obtained from the clauses in P by dropping the calls to Q, provided that these are provable in Q.

$$\frac{Q \vdash (A \leftarrow G) \wedge name(A) \notin defs(P)}{Q \prec P \vdash (A \leftarrow G)}$$

This rule describes the operational behavior of the *restriction* operation. Given a program expression Q and a program P, the clauses of $Q \prec P$ (Q restricted by P) are all the clauses $A \leftarrow G$ in Q such that the predicate name of A is not defined in P. Notice that the restriction operation requires that the second operand be a program and not a generic expression. This is due to the need of granting the programmer control over the actual (not virtual) clauses which will be involved in the operation.

The introduction of inheritance features into logic programming has been proposed by other authors (e.g. see [2, 17]). The restriction operator defined here is more primitive and it can be used with the other operations for the re-construction of the other proposals. Indeed, in section 3, we will show the re-construction of the proposal by Monteiro and Porto [17].

$$\frac{Q \vdash A}{P \vdash (A \ in \ Q)}$$

The last rule defines the kind of amalgamation supported by the language. Namely, solving an extended goal of the form $A \ in \ Q$ simply amounts to solving A in the program expression Q.

2.3 Bottom-up Semantics

We now define the denotational semantics of the amalgamated language in a bottom-up style. The bottom-up semantics is an extension of the standard $T(P)$ semantics for logic programs.

Recall that, for a logic program P, the immediate consequence operator $T(P)$ is a continuous mapping over Herbrand interpretations defined as follows [18]. For any Herbrand interpretation I:

$$A \in T(P)(I) \iff (\exists \bar{B} : \ A \leftarrow \bar{B} \in ground(P) \ \wedge \ \bar{B} \subseteq I)$$

where $\bar{B}$ is a (possibly empty) conjunction of atoms.

The set of immediate consequences of a definite program P from a given interpretation I depends only on the clauses of P and on I itself. In other words, $T(P)(I)$ contains the set of atomic formulae that can be derived from P in a single deduction step by assuming that the formulae in I are true in P.

In order to model the amalgamated language of program expressions, the notion of immediate consequence operator needs to be suitably extended. Indeed the set of immediate consequences of an extended program P given an interpretation I depends also on the truth of formulae of the form $A \ in \ E$. For instance, in the program

P
$r \leftarrow s, \ t \ in \ Q$

the set of immediate consequences of P depends on the assumption of truth we make on the formulae of Q. Namely, $r \in T(P)(I)$ only if $s \in I$ and if t is assumed to be true in Q.

Therefore, the immediate consequence operator for an extended program P should take as argument not only an interpretation for P but also a set of interpretations modelling the other programs that may be *involved* by P. The following definition formalizes the concept of program expressions involved by a program P or a program expression F.

Definition 1 *A program P* directly involves *a program expression F if and only if P contains a goal A in F in one of its clauses. A program expression F* directly involves *a program expression G if and only if there is a program P in F such that P directly involves G. Finally, F* involves *G if and only if either F directly involves G or F directly involves H and H involves G.*
We denote by $\gamma(F)$ and by $\Gamma(F)$ the set of program expressions directly involved and involved by F, respectively.

In the following, we will define the semantics of "well-formed" sets of program expressions. Intuitively speaking, a set $\mathcal{E}$ of program expressions is well-formed if it contains all the sub-expressions of each element of $\mathcal{E}$, and if it contains all the program expressions involved by each element of $\mathcal{E}$. The next definition formalizes these two notions.

Definition 2 *A set $\mathcal{E}$ of program expressions is* closed *if and only if for any $F \in \mathcal{E}$ each sub-expression of F belongs to $\mathcal{E}$.*
A set $\mathcal{E}$ of program expressions is complete *if and only if $\mathcal{E}$ is closed, and $\forall F \in \mathcal{E}$: $\Gamma(F) \subseteq \mathcal{E}$.*

Example 1 *Let us consider the collection of programs*

P	Q	R	S	T
$a \leftarrow b \ in \ Q \cup R^*$	$b \leftarrow d \ in \ (T \cup S) \cup R$	$d \leftarrow$	$e \leftarrow f, g$	$f \leftarrow d \ in \ R$

The set of the program expressions $\gamma(P)$ directly involved by P is $\{Q \cup R^*\}$, while the set of program expressions $\Gamma(P)$ involved by P is $\{Q \cup R^*, \ (T \cup S) \cup R, \ R\}$. Notice that the set $\Gamma(P)$ is not complete since it is not closed w.r.t. sub-expressions. The minimal complete superset of $\Gamma(P)$ is $\{Q \cup R^*, \ Q, \ R^*, \ R, \ (T \cup S) \cup R, \ T \cup S, \ T, \ S\}$.

From now onwards, we will represent a set $\mathcal{E}$ of (names of) program expressions by a vector $\bar{E}$ by assuming a total ordering on (names of) program expressions (e.g. such an ordering can be induced by the lexicographical order on program expressions). More generally we will use overlined symbols of the form $\bar{X}$ to denote vectors, and denote by $\bar{X}_n$ the n-th element of $\bar{X}$

We now define the extended immediate consequence operator T for programs and program expressions written in the amalgamated language. We denote by $\mathcal{B}$ the Herbrand base common to all programs.

Definition 3 *The operator $T : Exp \rightarrow 2^{\mathcal{B}} \rightarrow (2^{\mathcal{B}})^m \rightarrow 2^{\mathcal{B}}$ is inductively defined as follows. Let $\bar{E}$ be a vector representing a complete set of program expressions, and let $\bar{J}$ be a vector of Herbrand interpretations in a one-to-one correspondence with $\bar{E}$. Let $I \in \bar{J}$ and let $P \in \bar{E}$ Then:*

$$
\begin{aligned}
T(P)(I)(\bar{J}) &= \{A \mid \exists \bar{B} : A \leftarrow \bar{B} \in ground(P) \ \wedge \\
&\qquad \forall \bar{B}_i \in \bar{B} : \ ((\bar{B}_i \ atomic\ formula \implies \bar{B}_i \in I) \wedge \\
&\qquad\qquad (\bar{B}_i = C \ in \ \bar{E}_k \implies C \in \bar{J}_k))\}
\end{aligned}
$$

$$
T(\bar{E}_i \cup \bar{E}_h)(I)(\bar{J}) = T(\bar{E}_i)(I)(\bar{J}) \cup T(\bar{E}_h)(I)(\bar{J})
$$

$$
T(\bar{E}_i \cap \bar{E}_h)(I)(\bar{J}) = T(\bar{E}_i)(I)(\bar{J}) \cap T(\bar{E}_h)(I)(\bar{J})
$$

$$
T((\bar{E}_i)^*)(I)(\bar{J}) = T(\bar{E}_i)(\bar{J}_k)(\bar{J}) \qquad if\ \bar{E}_k = (\bar{E}_i)^*
$$

$$
T(\bar{E}_i \lhd \bar{E}_h)(I)(\bar{J}) = T(\bar{E}_i)\ (I \cup T(\bar{E}_h)(\bar{J}_h)(\bar{J}))(\bar{J})
$$

$$
T(\bar{E}_i \prec \bar{E}_h)(I)(\bar{J}) = T(\bar{E}_i)(I)(\bar{J}) \setminus \{q(\bar{t}) \mid q(\bar{t}) \in \mathcal{B} \wedge \exists \bar{t'}.\ q(\bar{t'}) \in T(\bar{E}_h)(\mathcal{B})(\bar{\mathcal{B}})\}
$$

The immediate consequences of a program P refer to an interpretation I that can be seen as the interpretation I in the standard definition of $T(P)(I)$. The novelty is the third argument of T, that is a vector of interpretations, each one associated with a program (or program expression) that may be involved by P. Actually, the n-th element $\bar{J}_n$ of a vector of interpretations $\bar{J}$ denotes the interpretation associated to the program expression $\bar{E}_n$. The immediate consequences of the union of two expressions is the set-theoretic union of the immediate consequences of the two expressions. Notice that symbol $\cup$ is used both as an operation on expressions and as an operation on interpretations. The definition for $\cap$ is analogous. The definition of the immediate consequences of an encapsulated expression $(\bar{E}_i)^*$ states that $(\bar{E}_i)^*$ "disregards" the input interpretation and looks only at its own interpretation. Roughly, this corresponds to a closure with respect to external influence. The immediate consequences of a program expression $\bar{E}_i$ importing a program expression $\bar{E}_h$ are the immediate consequences of $\bar{E}_i$ with respect to the interpretation of $\bar{E}_i$ itself extended with the immediate consequences of $\bar{E}_h$ with respect to its own interpretation. The immediate consequences of a program expression $\bar{E}_i$ restricted by a program $\bar{E}_h$ are the immediate consequences $q(\bar{t})$ of $\bar{E}_i$ such that the predicate q is not defined in $\bar{E}_h$ ($\bar{\mathcal{B}}$ denotes a vector of Herbrand bases).

Notice that the operator T properly extends the standard immediate consequences operator for logic programming. Namely, for any definite logic program the two operators coincide. It is worth observing that, strictly speaking, the above definition of the meaning of program expressions is not formulated in a compositional way. For instance, the meaning of a program P is defined via the T operator in terms of a complete set of program expressions including P, rather than just in terms of the program expressions *directly* involved by P. Notice, however, that the above definition can be easily reformulated in a compositional way in a rather straightforward manner. Roughly speaking, it is necessary to redefine T in such a way that it properly handles subsets of interpretations, instead of dealing with a fixed global vector of interpretations.

The bottom-up semantics of a complete set of expressions is defined by a mapping

over sets of interpretations where each interpretation is computed via T.

Definition 4 *Let $\bar{E} = <\bar{E}_1, \ldots, \bar{E}_m>$ be a complete vector of expressions, and let $\bar{J}$ a vector of interpretations. The operator $\mathbf{T}$ is a mapping over vectors of interpretations, defined as follows:*

$$
\mathbf{T}(<\bar{E}_1, \ldots, \bar{E}_m>)(\bar{J}) = <T(\bar{E}_1)(\bar{J}_1))(\bar{J}), \ldots, T(\bar{E}_m)(\bar{J}_m))(\bar{J})>.
$$

The powers of $\mathbf{T}$ are defined as usual [1]:

$$
\begin{aligned}
\mathbf{T}^0(\bar{E})(\bar{J}) &= \bar{J} \\
\mathbf{T}^{n+1}(\bar{E})(\bar{J}) &= \mathbf{T}(\bar{E})(\mathbf{T}^n(\bar{E})(\bar{J})) \\
\mathbf{T}^{\omega}(\bar{E})(\bar{J}) &= \bigcup_{n<\omega} \mathbf{T}^n(\bar{E})(\bar{J})
\end{aligned}
$$

The $\mathbf{T}$ operator is continuous and the least fixpoint of $\mathbf{T}(\bar{E})$ therefore coincides with $\mathbf{T}^{\omega}(\bar{E})(\bar{\emptyset})$, where $\bar{\emptyset}$ is a vector of empty interpretations. Notice that, in order to calculate the least fixpoint of an expression E, it is sufficient to consider any complete set $\bar{E}$ containing E. The minimal complete set containing E can be always obtained by closing the set $\Gamma(E)$ w.r.t. sub-expressions.

Example 2 *Consider the programs*

P	Q	R
$a \leftarrow b \ in \ Q \cup R$	$b \leftarrow c$	$c \leftarrow d$
		$d \leftarrow$

Suppose we are interested in the least fixpoint of the program P.
Let us consider $\bar{E} = \{P, Q, R, Q \cup R\}$. The powers of $\mathbf{T}(\bar{E})(\bar{\emptyset})$:

	P	Q	R	$Q \cup R$
$\mathbf{T}^0$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
$\mathbf{T}^1$	$\emptyset$	$\emptyset$	d	d
$\mathbf{T}^2$	$\emptyset$	$\emptyset$	d, c	d, c
$\mathbf{T}^3$	$\emptyset$	$\emptyset$	d, c	b, d, c
$\mathbf{T}^4$	a	$\emptyset$	d, c	b, d, c
$\mathbf{T}^5$	a	$\emptyset$	d, c	b, d, c

We have that $\mathbf{T}^4(<P, Q, R, Q \cup R>) = <\{a\}, \emptyset, \{d, c\}, \{b, d, c\}>$ is the least fixpoint of $\mathbf{T}$ and the first component, associated to P, is the set of all the atoms provable in P.

It can be proved that the top-down and bottom-up semantics coincide, as stated in the following theorem.

Theorem 1 *Let $\bar{E} = <\bar{E}_1, \ldots, \bar{E}_m>$ be a complete vector of program expressions. Let $\mathbf{T}^{\omega}(\bar{E})(\bar{\emptyset}) = <M_1, \ldots, M_m>$. Then, $\forall \bar{E}_i \in \bar{E}$:*

$$
A \in M_i \iff \bar{E}_i \vdash A.
$$

2.4 Meta-logical definition

To complete the view of the operational semantics, we turn the inference rules of Section 2.1 into meta-level axioms. The meta-logic preserves the simple and concise description of the inference rules, and it provides an "executable specification" of the amalgamated language. We give the axioms in the form of an extension of the vanilla metainterpreter, employing a two argument proof predicate *demo*. The predicate *demo* is used in a uniform way both to represent provable formulae ($demo(x, y)$) and clauses belonging to program expressions (e.g. $demo(x, y \leftarrow z)$).

Definition 5 *The extended vanilla metainterpreter is defined by the following clauses*

$$
\begin{array}{rcll}
demo(x, empty) & \leftarrow & & (1)\\
demo(x, (y, z)) & \leftarrow & demo(x, y), demo(x, z) & (2)\\
demo(x, y) & \leftarrow & demo(x, y \leftarrow z), demo(x, z) & (3)\\
demo(x \cup y, z \leftarrow w) & \leftarrow & demo(x, z \leftarrow w) & (4)\\
demo((x \cup y, z \leftarrow w) & \leftarrow & demo(y, z \leftarrow w) & (5)\\
demo(x \cap y, (z \leftarrow u, v)) & \leftarrow & demo(x, z \leftarrow u), demo(y, z \leftarrow v) & (6)\\
demo(x^*, y \leftarrow) & \leftarrow & demo(x, y) & (7)\\
demo(x \lhd y, z \leftarrow w) & \leftarrow & demo(x, z \leftarrow w, v), demo(y, v) & (8)\\
demo(x \prec y, z \leftarrow w) & \leftarrow & undefined(z, y), demo(x, z \leftarrow w) & (9)\\
demo(x, z \ in \ y) & \leftarrow & demo(y, z) & (10)
\end{array}
$$

The clauses (1)-(3) define the standard vanilla metainterpreter for logic programs extended with an extra argument. The first clause states that the empty goal is solved in every program expression x. Clause (2) deals with conjunctive goals, and states that a conjunction (y, z) is solved in the program expression x, if y is solved in x and z is solved in x. The third clause states that an atomic goal y is provable in a program expression x if a clause $y \leftarrow z$ belongs to x and z is recursively solved in x. The clauses (4)-(9) are the metalogical definition of the program composition operations previously defined. Clauses (4) and (5) state that a clause $z \leftarrow w$ belongs to the union of two program expressions x and y, if it belongs to x or to y. A clause $(z \leftarrow u, w)$ belongs to the intersection of two program expressions x and y if $z \leftarrow u$ belongs to x and $z \leftarrow w$ belongs to y. A unit clause $y \leftarrow empty$ belongs to the encapsulated program x^* if y is provable in x (clause (7)). Clause (8) deals with the import operation and states that a clause $z \leftarrow w$ belongs to x importing y if a clause $z \leftarrow w, v$ belongs to the "visible" part x and v is provable in the "hidden" part y. Clause (9) introduces a new predicate $undefined(z, y)$ that is true if the predicate name of z is not defined in the program y. The predicate $undefined$ can be implemented, for instance, as a membership test over the (finite) set of predicate names defined in y. So, a clause $z \leftarrow w$ belongs to x restricted by y if z is not defined in y, but the clause belongs to x. The clause (10) states that a goal of the form $z \ in \ y$ is provable in x if the goal z is provable in y.

3 Examples

This section is devoted to illustrate the expressive power of the amalgamated language. We show how the amalgamated language supports the re-construction of some extensions of logic programming that have been proposed in the past. The examples discussed include applications to contextual logic programming, hypothetical and hierarchical reasoning.

3.1 Hypothetical Reasoning

There are several proposals for extending logic programming to support forms of hypothetical reasoning. Suppose that a given chunk of knowledge is represented by a logic program and that different hypotheses (viewpoints) are represented by a collection of logic programs. Then the process of determining whether the assumption of some hypotheses allows one to derive certain conclusions may be interpreted in terms of dynamic compositions of programs. For example, Miller proposes in [16] an extension of Horn clause logic permitting implications both in goals and in clause bodies. An *implication goal* of the form $Q \Rightarrow G$, where Q is a set of clauses, is provable in a program P if G is provable in the program P extended with Q. Operationally, this corresponds to loading the code of Q before trying to prove G, and then discarding it after the proof of G succeeds or fails. An implication goal $Q \Rightarrow G$ may be employed to determine whether or not the assumption of a viewpoint Q allows the program P to derive the conclusion G.

We consider a restricted form of implication goals, which cannot be universally quantified (as in [15]). Under this assumption, an implication goal $Q \Rightarrow G$ occurring in a program P corresponds to a meta-level call of the form G in $P \cup Q$. According to the definition of meta-level goals, the operational reading is the following. To prove an implication $Q \Rightarrow G$, in the program P, prove G in the program $P \cup Q$. Consider the following example.

```
Travel                                                          Map
reach(X,Y,C,T) ← dist(X,Y,D)  in Map,                           dist(rome,florence,250)
                 time_req(X,Y,D,C,T)  in  Travel ∪ H            dist(rome,pisa,360)
weather(rome,sun)                                               dist(pisa,florence,80)
weather(pisa, rain)
weather(florence, rain)
speed(fiat_uno, 120)          H
speed(ford_fiesta, 130)       time_req(X,Y,D,C,T) ← speed(C,S),
speed(fiat_panda, 90)                               weather(X, W1),
traffic(pisa, rome, heavy)                          weather(Y, W2),
traffic(florence, rome, light)                      traffic(X,Y,L),
traffic(pisa, florence, light)                      average_time(D,S,W1,W2,L,T)
```

The program *Travel* contains information about travels and data concerning three possible cars, weather and traffic conditions. The predicate *reach*, given two cities and a car, returns the estimated time for the travel. Since distances between cities are

represented in the program *Map*, the goal $distance(X, Y, D)$ is evaluated in program *Map*. The goal *time_req* is evaluated in the union of the program *Travel* with a set of hypothesis (program H). H states that the estimated time depends from the speed of the car, the weather in the cities and the traffic on the road. We could change hypothesis and suppose that the estimated time depends only on the speed of the car and the traffic. That is, we can use as hypothesis a new program $H1$ and evaluate *time_req* in $Travel \cup H1$.

3.2 Contextual Logic Programming

Monteiro and Porto introduced in [17] the paradigm of *contextual logic programming*. The basic idea is to partition a logic program into a collection of named programs (called *units*) which are viewed as sets of context-dependent definitions of predicates. Programs may be dynamically combined together to form *contexts*. A context of programs denotes a hierarchical composition of programs, which may be represented by a stack structure. The dynamic formation of contexts is supported by *extension formulae* which may occur both in the goals and in the body of program clauses.

Contexts are represented by sequences of program names such as $u_n \ldots u_1$, where u_n is the element on top of the context. An extension formula $u \gg g$, where u is a program name and g a goal, extends the current context c by pushing u on top of c so that $u \gg g$ can be proved in the context c if g can be proved in the context uc, as stated by the rule

$$\frac{uc \vdash g}{c \vdash u \gg g} \tag{1}$$

The derivation of an atomic formula a is defined by two rules. The first rule

$$\frac{a \leftarrow b \in ground(u) \ \wedge \ uc \vdash b}{uc \vdash a} \tag{2}$$

states that a can be derived by applying a clause for a in the program u on top of the context and by deriving the resulting goal in the same context.

The second rule

$$\frac{name(a) \notin defs(u) \ \wedge \ c \vdash a}{uc \vdash a} \tag{3}$$

states that if the program on top of the context does not contain a definition for a, then a may be derived in the remaining part of the context. In (3), $name(a)$ denotes the name of the predicate of the atomic formula a and $defs(u)$ the set of predicate names defined in u. Notice that the condition $name(a) \notin defs(u)$ defines the same restriction mechanism which has been studied in Section 2.

The form of program composition supported by contextual logic programming may be viewed as an instance of the general approach supported by our amalgamated

language. Indeed the composition operation $\prec$ models the restriction mechanism and the operations $\cup$ and $*$ can be used to represent the hierarchical composition of programs specified by a context of programs. Actually, proving an extension formula $u \gg g$ in a program c corresponds to proving g in the composition $u \cup (c^* \prec u)$. Intuitively, the operation $\cup$ permits to access to the clauses of u according to rule (2) and, combined with $*$ and $\prec$, reconstructs the behavior specified by rule (3).

The correspondence between the above program expression and rules (1), (2) and (3) is illustrated by a simple example drawn from [17].

Consider the following programs *Authors* and *Books*.

Books
$wrote(plato, republic) \leftarrow$
$wrote(homer, iliad) \leftarrow$
$author(x) \leftarrow Authors \gg author(x)$

Authors
$author(x) \leftarrow wrote(x, y)$

The last rule is translated in

$$author(x) \leftarrow author(x) \ in \ Authors \cup (Books^* \prec Authors).$$

For instance, the top-down derivation of the goal $author(plato)$ in the program *Books* is illustrated by the following proof tree, which is constructed according to the inference rules presented in section 2. (In the tree each name is shortened into its initial symbol, e.g. B stands for *Books*, p for *plato* and e for *empty*).

$$\frac{B \vdash a(p) \leftarrow a(p) \ in \ A \cup (B^* \prec A) \qquad B \vdash a(p) \ in \ A \cup (B^* \prec A)}{B \vdash a(p)}$$

The left-hand hypothesis is justified by the presence of the clause in program *Books*. The right-hand hypothesis $B \vdash a(p) \ in \ A \cup (B^* \prec A)$ is in turn justified by the tree:

$$\frac{\dfrac{A \vdash a(p) \leftarrow w(p,r)}{A \cup (B^* \prec A) \vdash a(p) \leftarrow w(p,r)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{B \vdash w(p,r)}{B^* \vdash w(p,r) \leftarrow e}}{B^* \prec A \vdash w(p,r) \leftarrow e}}{A \cup (B^* \prec A) \vdash w(p,r) \leftarrow e} \qquad A \cup (B^* \prec A) \vdash e}{A \cup (B^* \prec A) \vdash w(p,r)}}{\dfrac{A \cup (B^* \prec A) \vdash a(p)}{B \vdash a(p) \ in \ A \cup (B^* \prec A)}}$$

3.3 Blocks

We now consider a form of contextual logic programming which does not take into account the restriction mechanism. This form of contextual logic programming can be defined by replacing rule (3) with

$$\frac{c \vdash a}{uc \vdash a} \tag{4}$$

where the condition $name(a) \notin defs(u)$ is withdrawn.

As pointed out in [6], this form of contextual logic programming defines scoping rules for predicate names resembling those defined by Giordano, Martelli and Rossi in [12] for a notion of *blocks* inspired by conventional programming languages.

This form of contextual logic programming can be reconstructed in terms of the operations $\cup$ and $*$. In fact, to prove an extension formula $u \gg g$ in a program c corresponds to proving g in the composition $u \cup c^*$. Formally, this corresponds to turning rule (1) into the metalevel goal $demo(u \cup c^*, x)$.

Other forms of contextual logic programming can be obtained by by defining the context handling mechanism and the restriction mechanism in different ways, as shown in [3, 6, 13].

3.4 Hierarchical Reasoning

Now, we show how the amalgamated language can support forms of hierarchical reasoning, that is, the definitions of inheritance relations between programs. An inheritance relation defines how the knowledge incorporated into a *super*-program is inherited by a *sub*-program.

We consider, in the following, a sort of a *isa* relationship. The meaning of the relation P *isa* Q where P and Q are logic programs, is that P inherits all the predicate definitions from Q, except for the predicates defined in P. To model the behavior of such hierarchical relation, we use the union, and restriction operations. In particular, proving a goal in a hierarchy P *isa* Q can be modelled by the composition $P \cup (Q \prec P)$. Let us explain the *isa* hierarchical relation with a simple example.

Mammal	*Human*	*Whale*
$eats(meat) \leftarrow$	$mode(walk) \leftarrow$	$habitat(sea) \leftarrow$
$eats(vegetables) \leftarrow$	$lives(X) \leftarrow works\_in(X)$	$mode(swim) \leftarrow$
$mode(gallop) \leftarrow$		
$habitat(ground) \leftarrow$		

John	*Moby*
$works(london) \leftarrow$	$lives(North\_sea) \leftarrow$
$age(25) \leftarrow$	

Suppose we want to know in which city *John* lives. The top-level goal will be

$$\leftarrow lives(X) \ in \ (John \ isa \ Human \ isa \ Mammal)$$

and the answer will be *London*. Indeed, John inherits the definitions of the predicates *lives* and *mode* from *Human*. Notice that the predicate *mode* in *Mammal* has been overriden by the definition in *Human*. Analogously, the goal
$\leftarrow mode(X)$ *in* $(Moby \ isa \ Whale \ isa \ Mammal)$ will answer *swim* because *mode* has no definition in *Moby* and the program *Whale* overrides the program *Mammal*.

4 Conclusions

The amalgamated language of program expressions has been implemented in Goedel by means of meta-programmings techniques ([5]). Although the meta-logical implementation of an amalgamated language provides us with a tool for experimenting with the re-constructions of different knowledge representation and reasoning systems, its low efficiency disallows large experiments. As a step towards a real system, we have developed an implementation based on a compiler plus an extended WAM. Essentially the extended WAM is capable of handling collection of programs [4], the composition operations and the "context switching", necessary to implement the *G in Prog* feature. In [4] we show some experimental results comparing the interpretation-oriented, compilation-oriented and the WAM-based implementation. The result is a considerable improvement of performance when comparing the WAM-based approach w.r.t. a interpretation oriented approach and an acceptable degradation of performance when it is compared with a compilation oriented approach.

The "in" feature can be interpreted as a means of sending messages from a virtual program to another virtual program. This feature along with the reconstruction of inheritance provided by the operations seems to offer opportunities for incorporating object-oriented features into logic programming. The advantage of our approach with respect to many others which have been proposed [11] is in the firm rooting of the semantics in a conservative extension of the semantics of pure logic programming.

Acknowledgements

The initial idea of an amalgamated language for composing logic programs originated from joint discussions and previous work with P. Mancarella and D. Pedreschi. We would also like to thank D. Aquilino for his valuable suggestions. This work has been partly supported by Esprit BRA 6810 *Compulog 2* and by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R. under grant n. 92.01564.PF69.

References

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

[2] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M.C. Meo. Differential logic programming. In *Proc. of Twentieth annual ACM POPL*, pages 359–370. ACM Press, 1993.

[3] A. Brogi. *Program Construction in Computational Logic*. PhD thesis, University of Pisa, March 1993.

[4] A. Brogi, A. Chiarelli, V. Mazzotta, P. Mancarella, D. Pedreschi, C. Renso, and F. Turini. Implementations of program composition operations. In *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, 1994. To appear.

[5] A. Brogi and S. Contiero. Goedel as a meta language for composing logic programs. In Franco Turini, editor, *Proceedings of META94, Fourth International Workshop on Meta Programming in Logic*. University of Pisa, 1994.

[6] A. Brogi, E. Lamma, and P. Mello. A general framework for structuring logic programs. Technical report, 4/1, CNR Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, 1990.

[7] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition Operators for Logic Theories. In J.W. Lloyd, editor, *Computational Logic, Symposium Proceedings*, pages 117–134. Springer-Verlag, 1990.

[8] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(3), 1994.

[9] A. Brogi and F. Turini. Metalogic for Knowledge Representation. In J.A. Allen, R. Fikes, and E. Sandewall, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, pages 100–106. Morgan Kaufmann, 1990.

[10] A. Brogi and F. Turini. Fully abstract compositional semantics for an algebra of logic programs. Technical Report TR 20/93, Dipartimento di Informatica, Università di Pisa, 1993.

[11] A. Davison. A survey of logic programming-based object oriented languages. In P. Wegner, A. Yonezawa, and G. Agha, editors, *Research directions in concurrent object-oriented programming*, pages 42–106. MIT Press, 1994.

[12] L. Giordano, A. Martelli, and G.F. Rossi. Extending Horn Clause Logic with Modules Constructs. *Theoretical Computer Science*, 1992.

[13] E. Lamma, P. Mello, and A. Natali. The design of an abstract machine for efficient implementation of contexts in logic programming. In G. Levi and M. Martelli, editors, *Proceedings Sixth International Conference on Logic Programming*, pages 303–317. The MIT Press, 1989.

[14] P. Mancarella and D. Pedreschi. An algebra of logic programs. In R. A. Kowalski and K. A. Bowen, editors, *Proceedings Fifth International Conference on Logic Programming*, pages 1006–1023. The MIT Press, 1988.

[15] L.T. McCarthy. Clausal intuitionistic logic 1: fixed point semantics. *Journal of Logic Programming*, 5:1–31, 1988.

[16] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[17] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proceedings Sixth International Conference on Logic Programming*, pages 284–302. The MIT Press, 1989.

[18] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

Generic Classes Parameterized by Data Structures

Silvia Clérici
Dept. Leng. y Sist. Informáticos
Univ. Politécnica de Cataluña
e-mail: silvia@lsi.upc.es

Ricardo Peña
Dept. Informática y Automática
Univ. Complutense de Madrid
e-mail: ricardo@dia.ucm.es

Abstract

We present a *class* construction as a possible extension to be included in a functional language. It establishes a mechanism for generic definition which integrates the concepts of overloading, inheritance, polymorphism and higher-order in a coherent way. By means of this mechanism, high order functions can be parameterized by data structures described as algebraic types. Given the definition of a particular type as a parameter, the class supplies instances of the functions for the given type in an automatic way.

In our opinion, this concept implies a natural evolution of the classes currently defined in the Haskell [3] and Gofer [6] languages. The mechanisms used by these languages are carefully reviewed in this paper. Examples are given for the generic definition of classes, such as *map*, *fold* and others, and several instances of them are shown for different algebraic types.

1 Introduction

Generecity is a term that, in a wide sense, has been used in programming languages as schema of abstraction. The concept is associated with that of parameterization, in the sense of the identification of a variable part and of an instantiation mechanism of a generic schema. Therefore it is also close to the idea of class, because the schema represents the class of all its valid instances. Moreover, the class formed by a subset of instances which match to a common pattern is conceptually a subclass. Therefore, the concept of inheritance is also related to that of generecity.

In this sense, it can be said that the evolution of programming languages in different paradigms shows a common tendency towards more generic mechanisms and constructions. Related concepts have been adopted in each paradigm to reach this objective. Another result of this evolution is that programming and specification languages are progressively closer. For example, functional languages now include constructions supporting concepts and methods comming from the area of algebraic specification (e.g. equational definition by patterns, constructor operations of algebraic types, abstract types, etc.).

With regard to genericity in the functional case, until the appearance of Haskell [3], the main abstraction mechanism was the combination of *higher-order* and *polimorphism* , which already provides a high level of abstraction: highe-order functions are

indeed generic algorithms and polymorphism is a special form of genericity. The generic schema is, in the case of polymorphism, that of an algorithm or a data structure "invariant" in all its possible concretions. This invariability of the schema is possible thanks to its independence of the form which the parameter adopts. That is, polymorphism is a particular case of genericity where the parameter is unrestricted.

But there are cases in which the algorithm changes from one instance to another(e.g. the addition of integer and real numbers)and, although a certain common pattern exists, the language mechanisms are insufficient to reflect this fact. In such cases we talk of *overloading*, usually meaning" the same name for different algoritms". But the concept implies a particular form of genericity, since the convenience of using the same name comes from the fact that there are common characteristics which are significant enough to do so. In other words, the overloaded operations (and the types over which they are defined) belong conceptually to the same class.

In functional programming the concept of class appears precisely as a consecuence of the attempt to resolve the problem of overloading in a uniform manner. The committee which developed the language Haskell had the aim of designing of a lazy functional language which would sythesize the state of the art. The most serious problem was the lack of a uniform criterion to overload several operations , such as aritmetic operations, equality or the conversion to strings. In previous languages such as Standard ML [11] or Miranda [9], the solution differs from one language to another, and even from one operation to another in the same language. The solution adopted in Haskell was based on a proposal by Wadler [10] to extend the familiar Hindley-Milner system [8] with what he called *type classes*. A type class is a set of types which have one or more overloaded operations defined over them.

In our opinion, the power of this new approach greatly exceded the initial aim, since the resulting construction provides a mechanism of genericity-inheritance complementary to and much wider than polymorphism. The class definition provides a greater level of abstraction and reusability, the definitions (of types, functions, equations, etc.) encapsulate characteristics which are common to a set of individuals, and all that remains is the definition of specifics for each instance. On the other hand the same mechanism can be used to define conditions to be required to the argumentss, and the hierarchy associated to the class inclusion or belonging is used to infer the validity of a specific parameter. Moreover, the language offers possibilities which up to now were present only in specification languages: to define "non-executable" abstract objects (theories in OBJ [4]), such as the class of types with equality or with a partial order relation, and organize the conceptual dependencies among them analogously to what happens in these languages [1].

In the language Gofer [6, 7], which arose as an experimental language related to the definition of Haskell, various extentions have been proposed which improve the mechanism and further emphasize the ideas of genericity and inheritance mentioned above.

In this work we propose a further extension to the class definition by means of a more powerful mechanism of parameterization, which allows the definition of generic algorithms suitable for a wide variety of data structures. The aim is to give complete generic definitions parameterized by algebraic types. Given the definition of a particular type as a parameter, the corresponding instance is automatically obtained. We

believe that this proposal integrates the ideas of overloading, inheritance, polymorphism and higher order in a generic mechanism of a very high level of abstraction.

The organizaton of the paper is as follows: In the following section we review the evolution of the class definition mechanism in Haskell and in later versions of Gofer. In section 3 we describe the extension mentioned above and give examples of generic classes. These represent generalizations of known higher order functions for lists, such as *map* or *fold*. In section 4 these definitions are instantiated for various data structures, explaining the mechanism of expansion and the resulting equations in each case. Finally, section 5 contains the conclussions and outlines possible continuations of this work.

2 Review of the concept of class in Haskell and Gofer

2.1 Type Classes in Haskell

A class in Haskell is defined as the grouping of a set of functions whose name and type must be given. The definition may also contain some equations which are considered default definitions for the involved functions. These definitions may be either preserved or replaced by others in each particular instance of the class. A class represents the set of all the types for which the functions are defined. For instance, the equality types class is defined:

$$\begin{array}{l}\textbf{class Eq a where}\\ \quad (==) :: a- > a- > Bool\\ \quad (/=) :: a- > a- > Bool\\ \quad x /= y = not\ (x == y)\end{array}$$

For a type T to be considered a member of the class, an instance declaration must be provided:

$$\begin{array}{l}\textbf{instance Eq T where}\\ \quad (==) = \ldots \text{definition} \ldots\end{array}$$

If two types T_1 and T_2 have been declared in this way, (==) is said to be an *overloaded* operation. Depending on the context in which (==) appears, the compiler will decide which instance of the operation is applicable. In some ambiguous cases the programmer must provide additional type information.

A function f defined in terms of an overloaded function g will also result overloaded. For instance, the function *member* of a list requires the elements of the list to have defined an equality operation. Its type is expressed in Haskell as follows:

$$member :: (Eq\ a) => a- > [a]- > Bool$$

This declaration restricts the instantiation of the type variable a: only those types belonging to class Eq will be acceptable. The expression to the left of the => symbol is called the *context*. Also, both a **class** and an **instance** declarations may have a

context as condition. An **instance** declaration context could even make reference to another instance of the same class. For instance, we can instantiate the class *Eq* for lists under the condition that the list element type belongs already to the class *Eq*:

```
instance (Eq a) => Eq [a] where
    (==) = ...the definition may use (==) for a ...
```

The mechanism for defining a class as a subclass of another class has a similar syntax. For instance, the following class definition

```
class (Eq a) => Ord a where
    (<=) :: a- > a- > Bool
```

defines the class *Ord* as a subclass of the class *Eq*. This means that *Ord inherits* all the operations defined in *Eq*. In Haskell, a class may be a subclass of more than one class, i.e., *multiple inheritance* is allowed. It is important to emphasize the difference between the two definitions above. In the first case, the context (*Eq a*) represents a requirement on the parameter *a*. In the second case, the context (*Eq a*) makes reference to the class *Eq* itself and defines *Ord* as a subclass of it.

2.2 Constructor Classes in Gofer

The language Gofer represents a step further in the direction initiated by Haskell. The first version [5] eliminates the Haskell restriction that to the right of the => symbol only one type constructor was allowed. But more significant changes came with the last version [7], in which the concept of *constructor classes* is introduced. Here the idea is that the class parameter is no longer a type variable *a* but a polymorphic type constructor *T* such as *Tree* or *List*. This allows the definition of *map* as an overloaded function (that was not possible in Haskell):

```
class Functor T where
    map :: (a- > b)- > Ta- > Tb
```

The underlying idea is that *map* represents an abstraction independent of the structural part of the data type: it converts type *a* elements into type *b* ones while keeping invariant the rest of the structure. So, there are two levels of parameters: the polymorphic ones *a* and *b* which are parameters of each *map* instance, and the type constructor *T* which is a parameter of the class.

It is also possible to define a subclass of a constructor class. The mechanism is the same as that of type classes. For instance,

```
class Functor T => Monad T where
    result :: a- > T a
    join :: T (T a)- > T a
    bind :: T a- > (a- > T b)- > T b
    join x = bind x id
    x 'bind' f = join (map f x)
```


expresses that any type belonging to *Monad* must belong previously to *Functor*, i.e., must have defined an instance of *map*. In addition, it must have defined the functions *result*, *join* and *bind*.

Constructor classes can be instantiated in the same way as type classes. For instance, for lists and binary trees we would write:

```
instance Functor List where
    map f [] = []
    map f (a:x) = f a : map f x

instance Functor Tree where
    map f Emptytree = Emptytree
    map f (left-: root :-right) = (map f left)-: (f root) :-(map f right)
```

By looking at these definitions we can observe that there exists a common pattern in the way the equations are written. The question comes up whether it would be possible to express this pattern and give a generic definition for all the instances of *map*. For doing that, since the number and shape of the equations depend on the type *T* data constructors, it would be needed to consider these data constructors as a parameter. The current language mechanisms do not allow this.

In [7], Jones makes some considerations about the parameterization of classes by monads. If types are considered monads, then generic functions might be defined in terms of *map*, *result*, *join* and *bind*. This approach would result in an indepencence of the particular data constructors. For instance, in the case of lists, instead of reasoning in terms of [ ] and (:), we would do in terms of *result* (unit list) and *join* (*concat*). The authors have explored somewhat this approach and have found two drawbacks in it: in one hand, it is not obvious that any type could be defined as a monad without loosing important characteristics. To this respect we can say that the list is an exceptionally regular type and an attempt to generalize its behaviour to other types leads to little intuitive results. In the other hand, not all interesting higher order functions can be expressed in monadic terms (the reader may wish trying the definition of *fold* for lists in terms of the four monadic functions above).

Summarizing this section, in Haskell the class parameter is a type variable α, in Gofer we can "open" α and use $T\ \beta$ as the class parameter. The next step would be to "open" $T\ \beta$ and use its internal structure as a parameter. What follows is a proposal to extend Gofer in this direction.

3 Language for defining parameterized classes

3.1 Structure Classes

In this section we propose the language constructions needed in Gofer to have classes parameterized by algebraic type definitions. By analogy with the terms *type classes* and *constructor classes* already explained, we will use for them the term *structure classes*.

As a first step in a full implementation of this feature, we can think of it as implemented by a preprocessor taking as input the classes definition written in the language here proposed, along with the program text in which instances of these classes are used. The output produced would be Gofer text consisting of constructor classes definitions and as many instance classes definitions as overloaded instances appear in the text.

To convert algebraic type definitions into a formal parameter we need to express them in a generic way. The starting point is the Gofer grammar for the **data** construction. To simplify the presentation we will assume prefix data constructors and a single polymorphic type variable a. Under these assumptions, the definition of an algebraic data type has the following aspect:

$$\textbf{data}\ T\ a = k_1\ (T_1^1\ a) \ldots (T_1^{m_1}\ a) \mid \ldots \mid k_n\ (T_n^1\ a) \ldots (T_n^{m_n}\ a)$$

where the T_i^j are type expressions that we call *secondary type constructors*, and which can contain T as a subexpression. For instance, in the declaration of an n-ary tree:

$$Ntree\ \alpha = Node\ \alpha\ (List\ Ntree\ \alpha)$$

the matching with the above definition would be: $n = 1$, $k_1 = Node$, $m_1 = 2$, $T_1^1 = Id$, $T_1^2 = List\ Ntree$, where Id denotes the identity type constructor explained in definition 3.3.

The generic definition of an algebraic type will consist of a pattern for this grammar, in some respects similar to those used for list comprehension definitions. In this pattern an index will range over the number of data constructors (k_i), and patterns for the secondary type costructors will be also given. In definition 3.2 the secondary constructors associated to each k_i are grouped to form two type constructor expressions. The first one corresponds to the non recursive part (those T_i^j which does'nt contain T), and the second expression corresponds to the recursive ocurrences of type $T\ a$.

Regarding function defining equations, their number depends on the number of type T data constructors and their shape depends on the secondary type constructors. So a generic way of expressing these equations will be given.

The type of the functions defined in the class will be also given in a generic way. In some cases, for instance in the definition of the *fold* function (see section 3.3), the number and shape of their parameters depend on the data constructors of the type. In this and in other examples it is not possible to give a single polymorphic type to the set of all instance functions of the class, even if this type is parameterized by the type constructor T. This indicates that the overloading concept derivated from our structure classes is more general than the one derivated from the constructor classes.

Finally, it may happen that a particular class definition is not applicable to every algebraic type definition. So, certain restrictions on the number or shape of the data constructors can be expressed in the language.

3.2 Class definition

To ease the language presentation, in what follows we shall use some mathematical notation such as subindexes in some constructor or function names. In an actual

programming language, a concrete syntax would be used for expressing these concepts.

Definition 3.1 A *class* definition has the following syntax:

class *context* ⇒*name* T **over**
$\quad T\ \alpha = \{k_i\ (M_i\ \alpha)\ (M_i'\ (T\ \alpha)) \mid i \leftarrow [1..m]\}$
restrictions ...restrictions on m, k_i, M_i y M_i'...
given f_i :: type of f_i...
defines hof_j :: type of higher order function hof_j
$\quad hof_j\ p_1 \ldots p_n$ = ... equations defining hof_j ...

where:

- *context:* has the same syntax and meaning as in Gofer.
- *name:* is the name of the class.
- T is the name of the polymorphic type constructor being the formal parameter of the class. The actual parameter must satisfy the **restrictions** section.
- The second line is a generic algebraic type declaration for T. This pattern and its interpretation is detailed in definition 3.2.
- The optional **restrictions** section may impose some restrictions on the parameter T. Some examples of restrictions are given in section 4.
- The **given** section declares the run time parameters of the higher order functions defined by the class. The number and type of these parameters depend in general on the shape of T.
- The **defines** section introduces the member functions defined by the class. It consists of the generic type and equations defining these functions. The equations may be conditional, being the conditions questions on the shape of T.

□

Definition 3.2 A *type definition* $T\ \alpha$ consists of the following pattern:

$$T\ \alpha = \{k_i\ (M_i\ \alpha)\ (M_i'\ (T\ \alpha)) \mid i \leftarrow [1..m]\}$$

where:

- $\{k_i \mid i \leftarrow [1..m]\}$, being $m \geq 1$, is a generic set of data constructors.
- $(M_i\ \alpha)$ is a generic type expression possibly including the polymorphic type variable α. It represents the non recursive part of the type definition.
- $(M_i'\ T\ \alpha)$ is a generic type expression including the subexpression $T\ \alpha$. It represents the recursive part of the type definition.

M_i and M_i' are generic type constructors defined in definition 3.3. □

Definition 3.3 A *generic type constructor* M is any expression generated by the following abstract grammar:

$$M ::= () \mid Id \mid M_1 \sim M_2 \mid M_1\ M_2 \mid Mon \mid Pol$$

where:

- $()$ is the null constructor. When applied to a polymorphic variable β it deletes β from the expression.
- Id is the identity constructor. When applied to a polymorphic variable β it produces β.
- The *expansion operator*, denoted $\sim$, defines the currying of constructors M_1 and M_2. A variable x with type $M_1 \sim M_2\ \alpha$ denotes a pair of variables x_1, x_2 whose types are respectively $M_1\ \alpha$ and $M_2\ \alpha$. We shall write this pair as $x_1 \sim x_2$. The expression $f\ x_1 \sim \cdots \sim x_n$ denotes applying the curried function f to the sequence of parameters $x_1 \cdots x_n$, i.e., it is equivalent to $f\ x_1 \cdots x_n$.
- $M_1\ M_2$ denotes the functional composition of type constructors. The standard interpretation for $M_1\ M_2\ \beta$ is $M_1\ (M_2\ \beta)$.
- Mon denotes a monomorphic type constructor such as $Bool$.
- Pol denotes a polymorphic type constructor such as $List$. Its arity must be greater or equal than one.

□

As an example of this definition, the polymorphic type expression $(Bool, \alpha, [\alpha])$ is parsed by the above grammar as follows:

$(,,)\ (Bool \sim Id \sim [])\ \alpha$ or, more abstractly, by $Pol\ (Mon \sim Id \sim Pol)\ \alpha$

It represents the tupling constructor $(,,)$, applied in a curried way to three type expressions:

$$(,,) :: bool \to \alpha \to [\alpha] \to (bool, \alpha, [\alpha])$$

A variable $x :: Bool \sim \alpha \sim [\alpha]$ denotes the triple $b \sim a \sim l$, where $b :: bool, a :: \alpha, l :: [\alpha]$. Also, $(,,)\ b \sim a \sim l$ is equivalent to $(,,)\ b\ a\ l$.

Frequently we shall use function names overloaded for different types. For instance, f_M, being M a generic type constructor, denotes the instance of function f corresponding to the type expression $M\ \alpha$. In the generic definitions below it will be frequent to define an instance f_{M_1} of f in terms of another instance f_{M_2} for a type constructor M_2 simpler than M_1. To ensure the termination of this recursion, we need to define the instances of f corresponding to the simplest type constructors. So, we give the following default definition.

Definition 3.4 For any function f, we define:

$$f_{Id}\ x = x, \qquad f_{()}\ x = (), \text{ and } \qquad ()\ x = ()$$

□

3.3 Some examples of classes

The *map* function

As a first example, we define the class af all algebraic types admitting the higher order function *map*. Given as parameters a function $f :: \alpha \to \beta$ and a data structure of type $T\ \alpha$, the *map* function produces a data structure of type $T\ \beta$ by applying the funtion f to every value of type α found in the original structure. As it has been said, in Gofer it is possible to give a generic type for this overloaded *map* function, but the programmer must define every instance of *map* he/she needs. Below we give a generic definition of *map*. Since the **restrictions** section is empty, in fact we are defining *map* for any data type.

class *with_map* T **over**
  $T\ \alpha = \{k_i\ (M_i\ \alpha)\ (M_i'\ (T\ \alpha)) \mid i \leftarrow [1..m]\}$
given $f :: \alpha \to \beta$
defines $map_T :: (\alpha \to \beta) \to T\ \alpha \to T\ \beta$
  $\{map_T\ f\ (k_i\ a_i\ x_i) = k_i\ (map_{M_i}\ f\ a_i)\ (map_{M_i'}\ (map_T\ f)\ x_i)$
  where $map_{M_1 \sim M_2}\ f\ x_1 \sim x_2 = (map_{M_1}\ f\ x_1) \sim (map_{M_2}\ f\ x_2)$
    $map_{Mon}\ f\ x = x$
  $\mid i \leftarrow [1..m]\}$

This definition of *map* involves two types of recursion: in one hand, the instance function map_T is recursively applied to the substructures of type $T\ \alpha$ of the original structure; in the other hand, the definition recurs to the instances map_{M_i} and $map_{M_i'}$ for other type constructors M_i and M_i' of the function being defined. The first recursion is the usual one. It terminates when the function is applied to a non recursive data constructor k_i (for instance, the empty list). For the second type of recursion we need instances of *map* for simpler and simpler type constructors. The same definition applies except in the cases covered by the **where** clause or by definition 3.4.

We give the details of the instantiation of *map* for the non recursive polymorphic type $(bool, \alpha, \alpha)$. In section 4 more significant examples are presented.

instance *with_map Trip*
where $Trip\ \alpha = triple\ Bool\ \alpha\ \alpha$

The substitution of formal parameters by actuals and the equation generated by the instantiation mechanism would be as follows:

$$k_1 = triple \;\Big|\; \begin{array}{l} M_1 = Bool \sim Id \sim Id \\ a_1 = Bool \sim Id \sim Id\ a = b\ a_1\ a_2 \end{array} \;\Big|\; \begin{array}{l} M_1' = () \\ x_1 = () \end{array}$$

$$\begin{aligned} &map_{Trip}\ f\ (triple\ b \sim a_1 \sim a_2) \\ &\quad = triple\ (map_{Bool \sim Id \sim Id}\ f\ b \sim a_1 \sim a_2)\ (map_{()}\ (map_{Trip}\ f)\ ()\ ) \\ &\quad = triple\ (map_{Bool}\ f\ b) \sim (map_{Id}\ f\ a_1) \sim (map_{Id}\ f\ a_2)\ (()\ ()) \\ &\quad = triple\ b \sim (f\ a_1) \sim (f\ a_2) \end{aligned}$$

$$map_{Trip}\ f\ (triple\ b\ a_1\ a_2) = triple\ b\ (f\ a_1)\ (f\ a_2)$$

The *fold* function

The intuition behind *fold* is to "reduce" a structure of type $T\ \alpha$ to a value of type β. For this purpose it uses a set $\{r_i\}_{i\in\{1..m\}}$ of reductor functions homomorphical to the type $T\ \alpha$ data constructors. The following class defines the higher order function *fold* for any algebraic data type $T\ \alpha$.

$$
\begin{array}{ll}
\textbf{class } \mathit{with\_fold}\ T \textbf{ over} & \\
\qquad T\ \alpha = \{k_i\ (M_i\ \alpha)\ (M_i'\ (T\ \alpha)) \mid i \leftarrow [1..m]\} & \\
\textbf{given} & \{r_i :: M_i\ \alpha \rightarrow M_i'\ \beta \rightarrow \beta \mid i \leftarrow [1..m]\} \\
\textbf{restrictions} & \{M_i' \textbf{ subclass of } \mathit{with\_map} \mid i \leftarrow [1..m]\} \\
\textbf{defines} & \mathit{fold}_T :: \{M_i\ \alpha \rightarrow M_i'\ \beta \rightarrow \beta \mid i \leftarrow [1..m]\} \rightarrow T\ \alpha \rightarrow \beta \\
& \{\mathit{fold}_T\ \{r_i \mid i \leftarrow [1..m]\}\ (k_j\ a_j\ x_j) = \\
& \qquad r_j\ a_j\ (\mathit{map}_{M_j'}\ (\mathit{fold}_T\ \{r_i \mid i \leftarrow [1..m]\})\ x_j) \\
& \mid j \leftarrow [1..m]\}
\end{array}
$$

In this example only one level of recursion is present: the application of fold_T to the recursive substructures of type $T\ \alpha$. This recursion terminates when fold_T is applied to a non recursive data constructor k_j. The class requires the existence of a *map* instance for the generic type constructor M_j'. The *map* function distributes the application of fold_T to all the substructures of type $T\ \alpha$ present in the original structure. Since any type belongs to the *with_map* class, then any type will also belong to the *with_fold* class.

We present here, omitting the details, the instance of *fold* for the monomorphic type *Bool*. In section 4 more complex examples are presented.

$$\mathit{fold}_{Bool} :: (bool \rightarrow \beta) \rightarrow bool \rightarrow \beta \qquad \mathit{fold}_{Bool}\ r\ b = r\ b$$

It simply applies the reductor function r received as a parameter to the boolean value b, i.e. it is the identity function $Id_{Bool \rightarrow \beta}$.

As it has been said at the begining of this section, the overloading introduced by this example can not be expressed with the current facilities of Gofer. Clearly, it is not possible to assign a single polymorphic type to all the instances of *fold*, as its type heavily depends on the shape of the parameter $T\ \alpha$.

The *map2* function

The *map2* function "intersects" two structures of types $T\ \alpha$ and $T\ \beta$ having corresponding matching portions. It produces a structure of type $T\ \delta$ by locating a value of type δ in those places where in the original structures there exist respectively values of types α and β. These δ values are obtained by applying a function $f :: \alpha \rightarrow \beta \rightarrow \delta$ to the values located in homolog places in the original structures. The instance for lists is well known and its utility have been stressed in many contexts [2]. The instance for trees could be useful as the basis for an unification algorithm.

$$
\begin{array}{ll}
\textbf{class } \mathit{with\_map2}\ T \textbf{ over} & \\
\qquad T\ \alpha = \{k_i\ (M_i\ \alpha)\ (M_i'\ (T\ \alpha)) \mid i \leftarrow [1..n]\} & \\
\textbf{restrictions} & n > 1 \Rightarrow \mathit{arity}\ (k_1) = 0 \\
& \{n > 1 \wedge i > 1 \Rightarrow \mathit{arity}\ (k_i) \neq 0 \\
& \mid i \leftarrow \{1..n\}\} \\
\textbf{given} & f :: \alpha \rightarrow \beta \rightarrow \delta \\
\textbf{defines} & \mathit{map2}_T :: (\alpha \rightarrow \beta \rightarrow \delta) \rightarrow T\ \alpha \rightarrow T\ \beta \rightarrow T\ \delta \\
& \{[n = 1 \vee (n > 1 \wedge \mathit{arity}\ (k_i) \neq 0)] \Rightarrow \mathit{map2}_T\ f\ (k_i\ a_i\ x_i)\ (k_i\ b_i\ y_i) \\
& \qquad\qquad = k_i\ (\mathit{map2}_{M_i}\ f\ a_i\ b_i)\ (\mathit{map2}_{M_i'}\ (\mathit{map2}_T\ f)\ x_i\ y_i) \\
& \qquad\qquad \textbf{where } \mathit{map2}_{M_1 \sim M_2}\ f\ (x_1 \sim x_2)\ (y_1 \sim y_2) = \\
& \qquad\qquad\qquad (\mathit{map2}_{M_1}\ f\ x_1\ y_1) \sim (\mathit{map}_{M_2}\ f\ x_2\ y_2) \\
& \qquad\qquad\qquad \mathit{map2}_{Mon}\ f\ x\ y = x \sim y \\
& \mid i \leftarrow \{1..n\}\} \\
& [n > 1] \Rightarrow \mathit{map2}_T\ f\ x\ y = k_1
\end{array}
$$

In this example we emphasize the importance of the **restrictions** clause. Although we have not made a special effort to define a restrictions language, we indicate that it would have to express any restriction on the parameter $T\ \alpha$ that could be checked at compile time. In this example we are asking that, if $T\ \alpha$ has more than one data constructor, then exactly one of the data constructors must have arity zero. This constructor is used in the second (conditional) equation to deliver a "default" structure for those parts where the original structures do not match. In the instance for lists this covers the case of different length lists, and in the instance for binary trees, this covers not matching subtrees. The default constructors are respectively the empty list and the empty tree.

The *traversal* function

The *traversable* class defines the *traversal* function doing the traversal of a data structure of type $T\ \alpha$ and producing the list $[\alpha]$ of all type α elements.

$$
\begin{array}{ll}
\textbf{class } \mathit{with\_map}\ T \Rightarrow \mathit{traversable}\ T \textbf{ over} & \\
\qquad T\ \alpha = \{k_i\ (M_i\ \alpha)\ (M_i'\ (T\ \alpha)) \mid i \leftarrow [1..m]\} & \\
\textbf{defines} & \mathit{traversal}_T :: T\ \alpha \rightarrow [\alpha] \\
& \{\mathit{traversal}_T\ (k_i\ a_i\ x_i) = \\
& \quad (\mathit{traversal}_{M_i}\ a_i) +\!\!+ \mathit{traversal}_{M_i'\ List}\ (\mathit{map}_{M_i'}\ \mathit{traversal}_T\ x_i) \\
& \quad \textbf{where } \mathit{traversal}_{M_1 \sim M_2}\ (x_1 \sim x_2) = (\mathit{traversal}_{M_1}\ x_1) +\!\!+ (\mathit{traversal}_{M_2}\ x_2) \\
& \qquad\qquad \mathit{traversal}_{Mon}\ x = [\,] \\
& \qquad\qquad \mathit{traversal}_{Id}\ x = [x] \\
& \qquad\qquad \mathit{traversal}_{()}\ x = [\,] \\
& \qquad\qquad \mathit{traversal}_{List}\ x = x \\
& \mid i \leftarrow \{1..m\}\}
\end{array}
$$

For the traversal to make sense, the structure must only contain type α elements, being α any type. However, the definition above does not assume this property. If

there were values of other types, they simply would not appear in the resulting list. Since the only parameter of the *traversal* function is the structure itself, let us note in this case that the **given** section is empty.

4 Instantiation of paramerized classes

In this section, some instances of the classes defined in the previous section are presented. We explain to a certain level of detail the tasks the preprocessor must accomplish. The first step consists of an "clever" pattern matching between the formal parameter $T\ \alpha$ of the class and the definition, as an algebraic type, of the actual parameter. By clever we mean that in such a matching the actual data constructors k_i must be numbered and, for each of them, the secondary constructors of their parameters must be grouped and reordered, so determining the actual expressions for M_i and M_i'. The initial ordering of the parameters must however be remembered so that the format of the generated equations be the expected one.

Once k_i, M_i and M_i' are determined, the next step (we can call it the expansion step) consists essentially of a rewriting process applying the definitions contained in the class, or the default definitions, from left to right. The examples clarify the details.

Example 4.1 Instantiation of the class *with_map* with the type *binary tree of* α.

instance *with_map Bintree*
where $Bintree\ \alpha = \triangle \mid \text{⩓}\ (Bintree\ \alpha)\ \alpha\ (Bintree\ \alpha)$

It follows the subtitution of formal variables by actual ones and the resulting equations for the corresponding instance of *map*:

$$k_1 = \triangle \;\Big|\; \begin{matrix} M_1 = () \\ a_1 = () \end{matrix} \;\Big|\; \begin{matrix} M_1' = () \\ x_1 = () \end{matrix} \;\Big\|\; k_2 = \text{⩓} \;\Big|\; \begin{matrix} M_2 = Id \\ a_2 = a \end{matrix} \;\Big|\; \begin{matrix} M_2' = Id \sim Id \\ x_2 = y_1 \sim\ y_2 \end{matrix}$$

$$\begin{aligned}
&map_{Bintree} :: (\alpha\to\beta)\to Bintree\ \alpha\to Bintree\ \beta \\
&map_{Bintree}\ f\ \triangle = map_{Bintree}\ f\ (\triangle\ (map_{()}\ f\ a_1)\ (map_{()}\ f\ x_1) = \\
&\quad = \triangle\ (()\ f\ ())\ ((()\ f\ ()) = \triangle \\
&map_{Bintree}\ f\ (\text{⩓}\ y_1\ a\ y_2) = map_{Bintree}\ f\ (\text{⩓}\ a\ y_1 \sim y_2) = \\
&\quad = \text{⩓}\ (map_{Id}\ f\ a)\ (map_{Id\sim Id}\ (map_{Bintree}\ f)\ y_1 \sim y_2) \\
&\quad = \text{⩓}\ (map_{Id}\ (map_{Bintree}\ f)\ y_1)\ (map_{Id}\ f\ a)\ (map_{Id}\ (map_{Bintree}\ f)\ y_2) \\
&\quad = \text{⩓}\ (map_{Bintree}\ f\ y_1)\ (f\ a)\ (map_{Bintree}\ f\ y_2)
\end{aligned}$$

Let us note in the second and fourth terms of the second equation that the parameters of the constructor ⩓ suffer first a grouping and then they recover the original order, corresponding to the appearing and disappearing of the expansion operator $\sim$. The reason for this grouping and reordering is the already commented matching between the formal parameter $T\ \alpha$ and the definition of the actual parameter $Bintree\ \alpha$.

Example 4.2 Instantiation of the class *with_fold* with the type *binary tree of* α *with elements only in the leaves.*

instance *with_fold Tree*
where $Tree\ \alpha = leaf\ \alpha \mid (Tree\ \alpha) :\wedge: (Tree\ \alpha)$

It follows the subtitution of formal variables by actual ones and the resulting equations for the corresponding instance of *fold*:

$$k_1 = leaf \;\Big|\; \begin{matrix} M_1 = Id \\ a_1 = a \end{matrix} \;\Big|\; \begin{matrix} M_1' = () \\ x_1 = () \end{matrix} \;\Big\|\; k_2 = :\wedge: \;\Big|\; \begin{matrix} M_2 = () \\ a_2 = () \end{matrix} \;\Big|\; \begin{matrix} M_2' = Id \sim Id \\ x_2 = y_1 \sim\ y_2 \end{matrix}$$

$$\begin{aligned}
&r_1 :: Id\ \alpha\to()\ \beta\to\beta \qquad r_1 :: \alpha\to\beta \\
&r_2 :: ()\ \alpha\to Id \sim Id\ \beta\to\beta \qquad r_2 :: \beta\to\beta\to\beta \\
&fold_{Tree} :: (\alpha\to\beta)\to(\beta\to\beta\to\beta) \\
&fold_{Tree}\ r_1\ r_2\ (leaf\ a) = fold_{Tree}\ r_1\ r_2\ (leaf\ a_1\ x_1) = \\
&\quad = r_1\ a\ (map_{()}\ (fold_{Tree}\ r_1\ r_2)\ ()\ ) = r_1\ a \\
&fold_{Tree}\ r_1\ r_2\ (y_1 :\wedge: y_2) = fold_{Tree}\ r_1\ r_2\ (:\wedge: a_2\ x_2) = \\
&\quad = r_2\ ()\ (map_{Id\sim Id}\ (fold\ r_1\ r_2)\ y_1 \sim y_2) = r_2\ (fold\ r_1\ r_2\ y_1)\ (fold\ r_1\ r_2\ y_2)
\end{aligned}$$

Example 4.3 Instantiation of the class *with_map2* with the type *n-ary tree of* α.

instance *with_map2 Ntree*
where $Ntree\ \alpha = \text{⩓}\ \alpha\ (List\ Ntree\ \alpha)$

In a more usual notation we can write instead $Ntree\ \alpha = \text{⩓}\ \alpha\ [Ntree\ \alpha]$. Let us note that the number of constructors of $Ntree\ \alpha$ is $n = 1$. The instantiation of formal parameters is as follows:

$$k_1 = \text{⩓} \;\Big|\; \begin{matrix} M_1 = Id \\ a_1 = Id\ a = a \\ b_1 = Id\ b = b \end{matrix} \;\Big|\; \begin{matrix} M_1' = List \\ x_1 = List\ x = x \\ y_1 = List\ y = y \end{matrix}$$

Only the first of the two conditional equations of *with_map2* meets its condition and, as there is one constructor, only the following equation will be generated:

$$\begin{aligned}
&map2_{Ntree} :: (\alpha\to\beta\to\delta)\to Ntree\ \alpha\to Ntree\ \beta\to Ntree\ \delta \\
&map2_{Ntree}\ f\ (\text{⩓}\ a\ x)\ (\text{⩓}\ b\ y) = \text{⩓}\ (f\ a\ b)\ (map2_{List}\ (map2_{Ntree}\ f)\ x\ \ y)
\end{aligned}$$

Let us note that the detection of the possible no matching between the two *n*-ary tree parameters of $map2_{Ntree}$ is delegated to the instance $map2_{List}$ for lists. If the number of subtrees of the first tree is different from that of the second one, this instance generates a list whose length is the shortest of the two.

Example 4.4 Instantiation of the class *with_map2* with the type *binary tree of α with elements only in the leaves* defined in example 4.2.

In this case, the instantiation mechanism would produce an error as this type does not meet the **restrictions** section requirements. In effect, the type $Tree\ \alpha$ has more than one constructor and none of them is of arity zero. May be it is now more evident the need for this requirement: in this case the corresponding instance of *map2* would not be able to deliver a default tree value when the original trees do not match. We might modify the type definition by adding a zero-ary constructor, obtaining:

$$\textbf{instance } with\_map\ Tree'$$
$$\textbf{where } Tree'\ \alpha = \Delta \mid leaf\ \alpha \mid (Tree\ \alpha) :\wedge: (Tree\ \alpha)$$

With this modification the instantiation of *map2* will produce the following equations:

$$map2_{Tree'} :: (\alpha \rightarrow \beta \rightarrow \delta) \rightarrow Tree'\ \alpha \rightarrow Tree'\ \beta \rightarrow Tree'\ \delta$$
$$map2_{Tree'}\ f\ (leaf\ a)\ (leaf\ b) = leaf\ (f\ a\ b)$$
$$map2_{Tree'}\ f\ (x_1 :\wedge: x_2)\ (y_1 :\wedge: y_2) = (map2_{Tree'}\ f\ x_1\ y_1) :\wedge: (map2_{Tree'}\ f\ x_2\ y_2)$$
$$map2_{Tree'}\ f\ x\ y = \Delta$$

Example 4.5 Instantiation of the class *traversable* with the type *binary tree of α* defined in example 4.1. The equations generated in this case are the following:

$$traversal_{Bintree} :: Bintree\ \alpha \rightarrow [\alpha]$$
$$traversal_{Bintree}\ \Delta = [\,]$$
$$traversal_{Bintree}\ (x_1\ a\ x_2) = (traversal_{Bintree}\ x_1) +\!\!+ [a] +\!\!+ (traversal_{Bintree}\ x_2)$$

Due to the pattern matching between the formal and the actual parameters, the order in which the elements appear in the list depends on the order of the data constructor parameters. The generic traversal chooses the substructures from left to right following the order in which they are declared. In this example, the *inorder* traversal is generated. The programmer may be willing to alter its type declaration to produce other traversals.

5 Conclusion

We have presented a **class** construction as a proposal to be incorporated in a functional language supporting overloading. It represents a natural evolution of the overloading concepts already present in Haskell and Gofer. In Gofer it is possible to overload a polymorphic function whose type depends on a type constructor T. The set of functions with this overloaded type and name constitutes a *class*. The programmer must construct explicitly all the instance functions for different types T he/she may need. In our proposal, these instances are generated automatically from a single generic definition, and the types of the functions may be different (in the sense of non unifiable types) from one instance to another.

We believe that this proposal pushes up the level of abstraction of the language since the programmer can construct higher order functions for a large number of data types with a single definition.

As a continuation of this work we are currently trying to enlarge the number of higher order functions for which makes sense to have a generic definition. To this respect, most of the well known algorithmic schemes (for instance, the greedy method or the dynamic programming scheme) can be expressed as generic functions over a (restricted) generic data type. In the other hand, we are studying the possibilities opened by this mechanism for the definition of higher order functions for abstract data types.

With respect to the implementation of the ideas here proposed, the first step will consist of the construction of a preprocessor for Gofer. This approach, although useful, would have some drawbacks since the overloading that can be supported by the Gofer type inference system is somewhat limited for our purposes. The final approach would be the full incorporation of the facility into the language.

References

1. S. Clérici and F. Orejas. GSBL: An algebraic specification language based on inheritance. In *Proc. ECOOP'88 Oslo*, pages 78–92. Springer-Verlag, LNCS 322, August 1988.
2. Silvia Clérici, Ricardo Peña, and Nicos Milonakis. *Redes estáticas para procesos continuos en Miranda*, pages 95–110. ISBN:84-00-07383-5, 1993. PRODE'93. Programación Declarativa.
3. A. J. T. Davie. *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, 1992.
4. K. Futatsugi, J. A. Goguen, J. P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. 12th ACM Symp. on Principles of Programming Languages, New Orleans*, 1985.
5. M. P. Jones. *GOFER 2.21 release notes.* Oxford University Computing Laboratory, 1991.
6. M. P. Jones. *GOFER manual.* Oxford University Computing Laboratory, 1992.
7. M. P. Jones. *GOFER 2.28 release notes.* Oxford University Computing Laboratory, 1993.
8. R. Milner. A theory of type polimorphism in programming. *Journ Comput. Syst. Sci*, 17:348–375, 1978.
9. D.A. Turner. *Miranda: A Non-Strict Functional Language with Polymorphic Types*, pages 1–16. Number 201 in LNCS. Springer-Verlag, Berlin, 1985.
10. P.L Wadler and S. Blot. How to make ad-hoc polymorphism less ad-hoc. In *Proc 16th ACM Symposium on Principles of Programming Language, Austin Texas*, pages 60–76, 1989.
11. A. Wikström. *Functional Programming Using Standard ML.* Prentice-Hall, 1987.

TAS-D^{++} vs Tablas Semánticas

G. Aguilera J. L. Galán I. P. de Guzmán M. Ojeda *

Departamento de Matemática Aplicada.
Universidad de Málaga.
E-mail: gimac@ctima.uma.es

Resumen

TAS-D^{++} [1], [2] es un demostrador automático de teoremas vía refutación basado en el estudio de la estructura sintáctica de la fórmula para detectar subfórmulas válidas, insatisfacibles y equivalentes.

Las siglas TAS corresponden a *Transformaciones de Árboles Sintácticos*, la filosofía de TAS-D^{++} está basada en el tipo de transformaciones introducidas en TAS-NC [3]. TAS-D^{++} es el resultado de un refinamiento de TAS-ND (el algoritmo dual de TAS-NC). Este refinamiento consiste en incorporar a las transformaciones semántica-preservantes de TAS-ND, transformaciones que conservan la insatisfacibilidad para, de este modo, obtener una mayor eficiencia.

TAS-D^{++} toma como entrada una fórmula (o inferencia) de la Lógica Proposicional realizando el análisis sintáctico de la negación de la fórmula y obteniendo así el árbol de entrada al algoritmo TAS-D^{++} (una lista de árboles para el caso de una inferencia).

En el estudio de la validez de una inferencia, TAS-D^{++} utiliza estrategias adicionales de eficiencia para tratar el nuevo parámetro del problema (el número de hipótesis), que evitan la necesidad de mantener demasiadas fórmulas en memoria . Además, TAS-D^{++} genera un contramodelo en el caso de que la fórmula (o inferencia) de entrada no sea válida.

En este trabajo presentamos una comparación entre el demostrador automático de teoremas, para la Lógica Proposicional, TAS-D^{++} frente al conocido método de las tablas semánticas [4]. La elección para la comparación del método de las tablas semánticas se debe a que ambos están basados en la forma normal disyuntiva y son métodos de construción de modelos.

Para comparar, en tiempo y espacio, la implementación de ambos ATP's se ha utilizado un módulo adicional encargado de generar aleatoriamente fórmulas bien formadas para la Lógica Proposicional. Este módulo requiere dos parámetros, uno para indicar el tamaño de la fórmula que se va a generar y otro para determinar el número máximo de literales distintos que intervendrán en dicha fórmula.

*Los autores son miembros del grupo de investigación GIMAC.

Además de los módulos encargados de las implementaciones de los ATP's por comparar y del módulo de generación de fórmulas aleatorias, se ha utilizado un cuarto módulo estadístico, encargado de medir y comparar tiempos y espacios empleados por ambos demostradores dependiendo de los dos parámetros utilizados por el generador aleatorio. Todos los módulos desarrollados han sido implementados utilizando el lenguaje de programación C++.

Es notable observar el menor consumo tanto en tiempo como en espacio de TAS-D^{++} frente al empleado por el método de las tablas semánticas. Esta diferencia es especialmente importante cuando crece el número de símbolos en la fórmula. El mejor comportamiento de TAS-D^{++} en este caso se puede explicar atendiendo al hecho de que este ATP incorpora diversas estrategias de eficiencia para evitar distribuciones innecesarias, extrayendo informaci'on de la propia estructura de la fórmula, lo que puede llevar incluso a no realizar ninguna distribución.

References

[1] G. Aguilera, I.P. de Guzmán and M. Ojeda. TAS-D^{++}: Syntactic Trees Transformations for Automated Theorem Proving. Proceedings of JELIA'94, York (England), Sept. 1994. Also in Springer's Lect. Notes in Artif. Intelligence series (to appear).

[2] G. Aguilera, I.P. de Guzmán and M. Ojeda. Automated Model Building via Syntactic Trees Transformations. Proceedings of CADE-12, Nancy (France), June 1994.

[3] Gabriel Aguilera, Inma P. de Guzmán, Manuel Ojeda. *Un algoritmo eficiente y paralelo para la transformación a forma normal conjuntiva.* ProDe'93.

[4] R. Jeffrey. *Formal Logic: its scope and limits.* Mc. Graw-Hill.1981.

Gedblog: a Multi-Theories Deductive Environment to Specify Graphical Interfaces*

Domenico Aquilino Patrizia Asirelli Paola Inverardi

Istituto Elaborazione Informazione del C.N.R.
Via S. Maria 43, Pisa, Italy

Gedblog Aims

The Gedblog project starts from the idea of building a uniform environment, based on logic databases theory, that could be specialized to support fast prototyping of applications that take benefit from a declarative specification style. In this perspective, we propose its extension to support declarative specification of tools that are mainly based on graphical user interaction. The poster gives an overview of the system, as it is implemented now, by introducing the extended logic theories that Gedblog manages. It gives also a taste of the graphical specification language we have defined and of the applications it has been tested on.

1 Gedblog Theories

Gedblog manages logical theories composed of different kinds of clauses:

- *Horn Clauses* (rules and facts)
- *Integrity Constraints* (IC), in the form $A >> B1, ..., Bs$

These can be informally interpretated as follows: each time A is true, then $B1$ and ... and Bs must be true. IC are managed following the method of *modified program*.

- *Checks*, in the form $A1, ..., Am ==> B1, ..., Bn$

These formula are checked only on user request and do not modify the knowledge expressed by the theory, as integrity constraints do.

- *Transactions*, in the form T := Precondition # B1, ..., Bn # Postcondition

The goals in the body can modify the extensional component of the theory (facts). Modifications are recorded only if Postcondition succeeds.

*This work has been supported by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R.

By means of the system-defined predicate *theory/1*, it is possible to perform inclusion among theories. In this way, given a *starting* theory Th, the associated gedblog theory can be defined as the set-theoretic *union* of all the theories in the inclusion tree rooted in Th.

2 The graphical language

The graphic specification language[1] integrates the features of Motif and X11 in the Gedblog theories by means of *prototype* definitions and instantiations. The graphical input is managed, throught the *callback* mechanism, by transactions. A *prototype* enables to describe structured objects with parameters, and to associate them a graphical representation. It is possible to instantiate a prototype by fixing the values of its parameters, in order to define a visualizable graphical object (denoted by predicate *object*). An important point to stress is that in Gedblog the *frame* theory (the space of visualized objects) is managed in a deductive way. Only instances of the *object* predicate that can be deduced time by time are visualized.

3 Applications

A number of graphical interfaces has been prototyped by using Gedblog: an SADT browser, a software modules configuration interface, and some interface tools for the Oikos project (a process-centered environment for supporting the software development process developed at Department of Computer science of Pisa). The simple mechanism of inclusion among theories resulted very useful in the modularization of specifications we have realized until now.

Acknowledgements

We thank all people that worked to the implementation of Gedblog, as well as Domenico Apuzzo of Intecs Sistemi S.p.A. for his suggestions.

References

[1] D. Aquilino, P. Asirelli and P. Inverardi. Prototyping in the Gedblog System. In *Proc. 4th Conf. on Software Engineering and Knowledge Engineering*, IEEE, 1992.

LogicSQL: Augmenting SQL with Logic *

Ugo Manfredi, Mirko Sancassani (um,mirko@dslogics.it)
DS Logics s.r.l. Viale Silvani 1, 40122 Bologna, Italy

Abstract

LogicSQL is a query language which allows the exchange of data between a Prolog system and a relational DBMS. The language design goal was to achieve good expressivity with a clean syntax. In our experience this meant keeping the syntax of the language as close as possible to standard SQL. However, thanks to the logic paradigm, the expressive power of LogicSQL considerably increases that of standard SQL.

The LogicSQL language was designed to solve the Prolog/DBMS interfacing problems faced by the IDEA project [1], a decision support system in the epidemiological field. The IDEA Data Retrieval Sub-System is an *intelligent interface* between a health administrator and several DBMS where data are stored.

The main task of the IDEA Prolog/DBMS interface is to efficiently fill the gap between the logical view of data offered to the epidemiological user and coded in the IDEA knowledge bases and the physical organization of data. The query language we need in IDEA must conveniently solve syntactical as well as semantical problems.

In the past years many researchers have dealt with the problem of interfacing Prolog with relational DBMS. We reviewed some of the available interfaces, but we found all the reviewed systems unsatisfactory for our needs. Most of them provide a language which maps relations into predicates, trying to be completely transparent to the Prolog programmers. This approach, though elegant and clean, suffers from great limitations if used in real life applications.

The first problem arises from the mapping declaration (db tables to Prolog predicates) which is *static*. Unfortunately, in IDEA we must *dynamically* compose complex database queries, involving aggregate functions and joins between several tables. Therefore, the query language must be able to express a sort of *query template* which corresponds to different SQL queries, depending on which parts of the Prolog query have been instantiated. Without this capability we would loose most of the expressive power of a logical programming language like Prolog.

The second major problem is the expressiveness and simplicity of the Prolog/DBMS interface: starting a priori from a Prolog-like syntax and adding to it the missing features, it is easy to obtain languages that do not cover SQL entirely, or are much more complex and counterintuitive than SQL itself. On the other hand, supporting only

*Work partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant n. 92.01662.69 and by the Health Ministry of Emilia Romagna, Dr. Barbolini.

partially the potentialities of SQL (for example excluding group-by clauses, aggregate functions, having clauses, ...) was definitely out of question, given the strong requirements posed by the IDEA project.

When designing the syntax of our query language, we found SQL syntax simple and declarative enough to advise against the creation of a completely new one. In SQL relation's fields can be referred to by name, not by position as in Prolog, and this is a desirable feature.

Another problem was deciding how to return values (tuples) from the DBMS. SQL is a *set-oriented* language and to interface it with Prolog one has to choose whether to maintain this feature or not. In standard terminology, this means choosing between the *one-tuple-at-a-time* and the *all-the-tuples-at-once* approaches. We chose the former approach for the following reasons:

- It does not make sense to return all the tuples if the program needs only one or few: doing so will simply mean to waste time and storage space;
- If the program effectively needs all the tuples, the overhead of getting them one at a time is negligible. This is because most of the overhead of a query is connected to the *setup* phase (translating the query to SQL, compiling it, allocating and setting up the buffer area, etc.) while the fetching phase is very efficient.
- Commercial DBMS can be interfaced with some generality only through their embedded SQL interfaces: these interfaces return the tuples one at a time, and getting all the tuples means necessarily fetching them one at a time through the *cursor* mechanism.

Built on top of this base mechanism, we also offer the possibility of getting all the tuples: in this case we avoid using the Prolog set-oriented built-ins, in favour of directly asserting the tuples in memory in the form of Prolog *facts*. The user has complete control over the format of these assertions.

In LogicSQL, a query is expressed as the argument of the Prolog predicate `lsql/1` and, as the SQL `select` statement, it is made of several parts or clauses. The *LogicSQL* clauses maintain the same meaning of the SQL ones, but they are expressed by Prolog terms which include logical variables. Logical variables can be used both as input or output parameters depending on whether they are instantiated or not at run-time. In *LogicSQL*, logical input variables are used to specialize the query on the basis of information which is known only at query execution time, while logical output variables can be placed only in the `select` clause and are used to retrieve values of relation fields or results of computation performed by the underlying DBMS.

The following example gives a flavour of the language. The LogicSQL query:

```
lsql(select [sum(pop.number):Number, pop.city:City] from [pop] groupby [pop.city]) .
```

produces, if evaluated with `Number` and `City` unbound or with `Number` unbound and `City` bound to bologna, respectively the following SQL queries:

```
select sum(pop.number), pop.city from pop group by pop.city ;
select sum(pop.number), pop.city from pop where pop.city = 'bologna' group by pop.city ;
```

References

[1] C. Ruggieri and M. Sancassani. IDEA: Intelligent Data Retrieval in Prolog. In *Joint Conference on Declarative Programming GULP-PRODE'94*, Sept 1994.

A type checking tool for a formal specification language *

Nicos Mylonakis, Javier Pérez Campo
U. P. C., L. S. I. Department
C. Pau Gargallo 5 08028 Barcelona Spain
E-mail: mylonakis@lsi.upc.es, jperez@lsi.upc.es

June 1994

1 Introduction

We present a tool that performs static analysis for the loose algebraic specification language Glider. [2] The tool can help in the detection of errors and ambiguities of unfinished specifications, allowing the user an incremental way to develop and check specifications. The tool binds to every operation call appearing in a formula its defined arity, and if it is possible, it finishes the unfinished formulas of the given specification . This can help the user to verify that what he has written is what he wanted to express.

2 The Specification Language

In Glider [2] one can define new basic and generic types . Since the semantics of Glider is loose, a type will denote in general a class of models. It is possible to make instantiations of generic types, and we call them module expressions. Module expressions can be used as basic sorts, so they can be the type of a parameter of a generic sort or appear in the arity of an operation.

There are four different ways of defining a new type in Glider: just by axioms, by operation constructors, by subtyping or by inheritance. Types defined by subtyping or inheritance generate additionally, a subtype or subclass relation between the defined type and the type used to define it. Furthermore, it is possible to define subclass relation a posteriori between two types (see [3] for details).

Axioms in Glider are formulas of first order logic. Besides, Glider incorporates the "..." operator to make possible the incremental development of specifications . This operator can be inserted in a formula, variable declaration or even in the declaration of the arity of an operation, expressing that there are some elements missing. Finally, in Glider it is possible to have overloading of defined operations (adhoc polymorphism).

*Work done inside ICARUS project, C.E.C. DG XIII ESPRIT II program, contract n. 2537

3 The tool

The tool performs a static analysis of the specification. The two most outstanding points of this analysis are the subclass relation check (necessary to guarantee correctness of instantiation of generic sorts) and the formula analysis.

- Subclass Relation Check. To check the subclass relation between two modules we should check that the class of models denoting the subclass module is included in the class of models denoted by the superclass module. Since this checking is not proved to be decidable, we consider that there is a subclass relation between two modules iff it has been defined explicitly by the specifier.
- Formulae Analysis. The formulae analysis step performs a type checking of the several terms on the formulae contained in the specification to check. It also copes with unfinished formulae and declarations as well as with subtyping and overloaded function declarations. The algorithm can be found in [6] and in [5].

When the user requests a check of a program, the tool shows the different interpretations (because of overloading) with all the completed information inserted in it in a new window. The check tool also displays the errors and/or warnings found during the checking. When the user has found the desired alternative, he/she can copy it to the editor upgrading all the completed information.

The definition of static semantics is given in sets of rules like in [4]. The rules have two parts: a list of premises and a conclusion. Premises can be either boolean functions , judgements or proof obligations whereas the conclusion is a judgement.

This tool has been implemented using CENTAUR [1]. CENTAUR is a tool oriented to the development of programming environments for user defined languages.

References

1. Borras, P.; Clement D.; Despeyroux Th.; Incerpi J.; Kahn G.; Lang B.; Pascual V. CENTAUR: the system. ACM SIGPLAN Notices February 1989.
2. S. Clerici, R. Jiménez, F. Orejas. Semantic Constructions in the Specification Language Glider. Proceedings of WADT 93.
3. S. Clerici, F. Orejas. GSBL: an algebraic specification language based on inheritance. Proc. 1988 Europ. Conf. on Object Oriented Programming. LNCS 322.
4. Robin Milner, Mads Tofte, Robert Harper. The Definition of Standard ML. MIT Press 1990.
5. Javier Pérez Campo. Una modelización de la incompletitud en los programas. Research Report LSI-94-4-R. Department of L. S. I., U. P. C.
6. Javier Pérez Campo, Nicos Mylonakis. A one pass algorithm for type checking with overloading and polymorphism. LSI-92-14-R. Dept. of L. S. I., U. P. C.

OASIS 2.0: An Object Definition Language for Object Oriented Databases

Oscar Pastor López, Isidro Ramos Salavert, José Cuevas Aparici,
Jaume Devesa Llinares

DSIC-Universidad Politécnica de Valencia.
Camino de Vera, S/N , 46071 Valencia.
Tel: 3877350; Fax: 3877357; e-mail: opastor@dsic.upv.es

Abstract

We present OASIS 2.0, a new version of OASIS. OASIS is a formal, OO Specification Language developed in the DSIC ([1] [2] [3] [5]). In its initial versions, the Software Development Environment associated to the Language was formalized using first order theories that evolve in time. This lead us to a Logic Programming environment with the complement of a monitor to deal with dynamics. But it has a natural, sound and complete formal basis within the context of Dynamic Logic. A new formalization of Oasis according to this view has been developed in order to characterize properly objects as observable processes. The language expressiveness has been accordingly enriched, making possible to use it as a class definition language in OO Data Base environments.

1 Introduction

Research on the integration of the OO and the deductive approach for designing Information Systems and Databases has been a major topic of interest in the last years. The OO and deductive environment represented by **Oasis 1.0** as OO Specification Language, and **OO-METHOD** ([4]) as a methodological extension covering the Analysis and Design Software Production phases within an Automated Programming Paradigm, is the result of a research effort tackled during the last two years in the DSIC-UPV.

The transition to the new language version (OASIS 2.0) is accomplished by giving a new characterization to the different kind of formulas that, within a Dynamic Logic environment, are provided by OASIS. Additionally, the previous language version is also enriched with:

- the introduction of actors (or agents) to specify which actors are allowed to activate which events
- the introduction of a temporal logic to allow the specification of dynamic integrity constraints

- the specification of transactions and processes
- the possibility of defining interfaces or virtual classes to establish a protection model, fixing which view of a given object will have objects of other classes

2 Formalization of Oasis

A class in Oasis is made up of a class name, an identification function for instances (objects) of the class and a type or template that all the instances share.

In Oasis, an object is defined as an observable process. The process specification in a class allows us to specify object dinamics and determines the access relationship between the states of instances. Processes are constructed by using **events** as atomic actions and are terms in a well-defined algebra of processes.. But the designer has also the choice of grouping events in execution units, so-called **transactions.**

3 The Application

From an Oasis specification, an environment of industrial software production can be automatically generated by performing a translation process. This constitutes a prototype of the system, which can be validated and animated by the user.

We know how to obtain logical prototypes. The specification is translated to a formal theory, say clausal, functional or clausal with equality, whose operational semantics allows us to animate it. We can also generate OO prototypes that can be run over OO industrial software production tools rather than logic-based tools. In order to achieve a friendly-user interface, they are built over Windows-based systems, by using the graphical environments provided by development tools like MS-Access, Visual-Basic or Power-Builder.

References

[1] Pastor,O.:Hayes,F.:Bear,S. *OASIS: An Object Oriented Specification Language* In the Proceedings of the CAiSE-92 Conference, Springer-Verlag, pag.348-363; Mayo 1992, Manchester (UK)

[2] Pastor O. *Diseño y Desarrollo de un Entorno de Producción Automática de Sotfware basado en el modelo OO* PhD thesis, DSIC-UPV, May 1992.

[3] Pastor,O;Casamayor,J.C. *A Deductive and Object Oriented Environment for Software Production* In the Proc. of the Workshop in Expert Systems, AI and Software Engineering Application, ILPS-91, San Diego (California).

[4] Pastor,O.;Ramos,I.;Canos,J. *From Analysis to Design in an OO and deductive environment* Accepted for presentation in DEXA-93 Conference, Prague.

[5] Ramos,I. *Logic and OO Databases: a Declarative Approach* Proc. of the DEXA 90, Springer-Verlag 1990

LANM, SRA y CONTRADICCIÓN

GONZALO RAMOS

Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga

Resumen

Nuestro trabajo, de varios años, se centra en la creación de una nueva lógica no estándar, y en base a ella diseñar e implementar un sistema de razonamiento que tenga, tanto él como ella, un marcado enfoque cognitivo, es decir, se aproxime, de manera formal, lo más posible al razonamiento humano. En otras palabras estamos buscando crear lo que podríamos denominar una Lógica Cognitiva.

Esta lógica ya la hemos definido, y una introducción a la misma la podemos encontrar en [1]. Sólo comentar sobre ella que su denominación es LANM (Lógica Autodoxástica No Monótona), que como su nombre indica es autodoxástica, es decir, razona sobre sus propias creencias, que surge bajo la idea fundamental de que un enfoque que equipare conocimiento con creencia en lugar de conocimiento con veracidad es más adecuado para la simulación de procesos cognitivos de creencias humanas, y por último que sus principales características son: No monótona, Multivaluada y continua, con valores de creencia en el intervalo [-1,1], Abarca todos los estados doxásticos (curiosidad, coherencia, ajuste, conflicto), Soporta y trata las contradicciones.

Definida LANM, nuestra siguiente fase de trabajo fue la definición formal de un sistema de razonamiento basado en LANM, que trabajase con creencias con la potencia que nuestra lógica lo hace, que fuera capaz de soportar y resolver las contradicciones, y que tuviera un procedimiento avanzado de aprendizaje. Todo esto lo hemos conseguido con nuestro SRA (Sistema de Razonamiento Autodoxástico). Una introducción a SRA la podemos encontrar en [2]. El núcleo de SRA es un conjunto de reglas, de la forma $A_1/A_2/.../A_n$ --> B, cada una con un *coeficiente de creencia* (CC) asociado. Los $A_1,A_2,...,A_n$ son los hechos *antecedentes* de la regla, y B el hecho *consecuente*. Cada hecho (creencia bien formada) tiene asociado un *grado de creencia* (GC) que varía dentro del intervalo [-1,1]. El CC de las reglas sin embargo varía dentro del intervalo (0,1], ya que solo trabajamos con reglas en las que "creemos" al menos en cierta medida.

La representación de la deducción (o estado del proceso deductivo) la realizaremos por medio de un grafo dirigido acíclico. Los nodos del grafo serán ternas de la forma |R|H|GC|, donde R es el número de la regla aplicada, H el hecho deducido con R y GC el grado de creencia de este hecho. Consideramos dos tipos de arcos, los *positivos* y

los *negativos*, y un tipo especial que explicaremos posteriormente denominado *conflictivo*. Se conectará un arco positivo de un nodo hacia otro (sucesor del primero) cuando un incremento en el GC del primero supusiera también un incremento en el GC del segundo. Por el contrario un arco se marcará como negativo cuando un incremento del GC del antecesor conlleve una disminución del GC del sucesor. Los arcos conflictivos son unos arcos especiales que se crean cuando en el proceso deductivo, que veremos a continuación, aparecen dos hechos iguales pero con distinto signo en su grado de creencia, dando lugar a una contradicción (o conflicto).

El proceso deductivo consiste en, a partir de un conjunto de hechos y con las reglas que dispongamos, ir ampliando el grafo de la deducción hasta que se cumpla la condición de terminación del proceso deductivo o no podamos aplicar más reglas. Nosotros hemos diseñado este proceso deductivo de forma que puede realizarse en dos modos, el modo reflexivo y el modo resolutivo. En el *modo resolutivo* se amplía el grafo solo cuando hay ganancia de seguridad en la creencia o bien un conflicto. En este modo se resuelven las ganancias y las contradicciones que surgen pero nunca se modifican los CC de las reglas. En el *modo reflexivo* en cambio sólo no se amplía el grafo cuando hay identidad del GC en la duplicidad de hechos. Los refuerzos y las contradicciones se resuelven y también afectan, a veces, a los CC de las reglas. Es esta capacidad del sistema en modo reflexivo de modificar los CC de las reglas lo que le da la capacidad de aprender, de utilizar la experiencia que va adquiriendo para depurar y mejorar sus reglas.

En conclusión: Hemos desarrollado un sistema, SRA, que trata y resuelve las contradicciones, y donde la aparición de estas puede producir un debilitamiento de las reglas del sistema. Este debilitamiento junto con el refuerzo visto en [2] se complementan, produciendo una tensión dinámica de las dos tendencias que tiene como resultado un sistema "vivo", que aprende de la experiencia, mejorando sus "conocimientos".

Para más información contactar con:

Gonzalo Ramos Jiménez
Dpto. Lenguajes y Ciencias de la Computación
Universidad de Málaga Plza. El Ejido s/n. 29013 Málaga
Tel. (95) 2131400 FAX: (95) 2131355
(Grupo de Investigación de Ciencias Cognitivas. Código P.A.I.: 1080)

Referencias

[1] ALFREDO BURRIEZA, INMA P. DE GUZMÁN, GONZALO RAMOS. "'43 escalones' hacia una máquina inteligente (Fundamentos de la Lógica Autodoxástica No Monotónica)" *PRODE'92*, UPM, Madrid (1992)

[2] GONZALO RAMOS. "Un Sistema de Razonamiento Autodoxástico 'tras la escalera' (Basado en la Lógica Autodoxástica No Monótona)" *PRODE'93*, J. AGUSTÍ, P. GARCÍA (eds.) IIIA CSIC, Blanes (1993)

Especificación Orientada a Objetos desde un enfoque Algebraico

José A. Troyano Jesús Torres Miguel Toro

Dpto. de Lenguajes y Sistemas Informáticos
Universidad de Sevilla
Avd. Reina Mercedes s/n
41012 Sevilla, Spain

Resumen

Este trabajo es un intento de relacionar las especificaciones algebraicas, de datos y procesos, con el paradigma orientado a objetos, desde el punto de vista de un lenguaje de especificación formal. Mostramos algunas características del lenguaje TESORO y la facilidad que incorpora este lenguaje para la descripción del comportamiento de los objetos utilizando distintos estilos de especificación.

1 Especificación del Modelo OO

Nuestro modelo OO se describe a partir de una especificación de tipos de datos, una especificación de clases y un conjunto de relaciones entre clases. Así, una especificación en TESORO constará de las siguientes secciones:

Biblioteca. En esta sección se enumerarán los tipos abstractos de datos (TAD) que se vayan a utilizar para la definición de los dominios de los atributos de las clases especificadas. El lenguaje que usaremos para la especificación algebraica de los TAD será ACT ONE [1].

Clases. Las clases se utilizan para especificar la estructura y comportamiento que comparten una colección de objetos, y pueden ser simples, si se describen sin apoyarse en otras clases, o compuestas, si se definen a partir de otras clases con los constructores de herencia (que permite describir una nueva clase de objetos como una extensión de clases previamente definidas) y agregación (que da características de clase a una relación). La especificación de las clases se compone de tres secciones: a) sección de **atributos**, donde se describen los aspectos estructurales de una clase de objetos, definiendo atributos de identificación, atributos constantes y atributos variables, además de restricciones estáticas sobre el valor de los atributos variables, b) sección de **eventos**, donde se describen los aspectos de comportamiento de una clase de objetos a través de permisos, disparos y restricciones dinámicas (imponiendo un orden de ocurrencia de eventos mediante una especificación algebraica de procesos [2]) y c) sección de **transiciones**, que define como van a cambiar los valores de los

atributos variables de los objetos de una clase, a través de la ocurrencia de eventos o de los cambios de otros atributos.

Relaciones. En el modelo se pueden definir relaciones entre clases. Estas relaciones se van a basar en la sincronización y comunicación de los objetos de distintas clases, lo que se consigue a través de eventos compartidos. Estas relaciones sirven básicamente para describir las ligaduras entre los distintos componentes del sistema.

2 Estilos de Especificación del Comportamiento

Haremos la siguiente distinción a la hora de definir el estado de un objeto: *estado explícito* que depende de los valores de los atributos en un determinado instante y *estado implícito* que depende de la secuencia de eventos que han ocurrido hasta un determinado momento, en la vida de un objeto. Definimos así dos estilos de especificación del comportamiento de los objetos de una clase:

Especificación Orientada a Estados. Este estilo se caracteriza porque se definen un conjunto de variables adicionales que hacen explícito el estado de un objeto en cada momento, describiéndose el comportamiento a través de un conjunto de permisos de ocurrencia de eventos, desde unos estados determinados y un conjunto de transiciones que determinan los cambios de estados tras la ocurrencia de eventos.

Especificación Orientada a Restricciones. Este estilo se caracteriza porque se utilizan los operadores de álgebra de procesos para especificar el comportamiento de un objeto como el conjunto de las trazas de eventos válidas en la vida del objeto, a través de las restricciones dinámicas y sin apoyarnos en el estado explícito del objeto.

Como trabajo futuro pretendemos definir una semántica operacional de TESORO basada en un sistema de transiciones. A partir de la descripción de la semántica, podremos generar un prototipo desde la especificación. Otra vía para construir un prototipo es el propuesto en [4], donde se relaciona TESORO con LOTOS [3].

Bibliografía

[1] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification, Part 1.* Springer-Verlag. Berlin. 1985.

[2] *Communicating Sequential Processes.* Prentice-Hall International Seires in Computer Science. 1985.

[3] ISO-Information Processing Systems - Open Systems Interconnection. *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour.* ISO 8807. 1988.

[4] Jesús Torres, José A. Troyano, Miguel Toro. *Desde el Lenguaje de Especificación Orientado a Objetos TESORO a LOTOS.* Por aparecer en la Revista de Informática y Automática.

Combining Depth-first and Breadth-first Search in Prolog Execution*

Jordi Tubella Antonio González

Universitat Politècnica de Catalunya
c/ Gran Capità s/n - Mòdul D6
E-08071 Barcelona (Spain)
E-mail: {jordit, antonio}@ac.upc.es

Abstract

Multipath is an execution model that combines a depth-first and breadth-first traversing of the SLD-tree. Performance figures about a sequential and a parallel execution of Multipath are depicted. Results on a sequential environment outperform the standard WAM implementation. Fine-grained parallelism is easily obtained, and it is efficiently exploited by a SIMD-like machine.

1 Multipath Execution Model

The standard execution of a Prolog program follows a depth-first left-to-right traversing of the SLD-tree. Traditionally, a breadth-first search has not been considered due to the complexity of its management. Obviously, a complete breadth-first search can not be considered due to the amount of memory it would need. Nevertheless, a combined breadth-first and depth-first search would have a number of advantages. A novel execution model, *Multipath*, that combines these two search strategies is proposed [2]. The main advantages of this model over a depth-first search are the following:

- The overhead due to the execution of control instructions is considerably reduced. The impact of this is especially important for combinatorial search problems.
- The number of unifications decreases. This happens when the same unification occurs in different paths of the SLD-tree. The standard model repeats the unification operation every time. Multipath may avoid this unnecessary recomputations.
- A new type of parallelism, called *path parallelism*, is exhibit by the model. This type of parallelism is fine-grained and may be exploited by a SIMD-like machine.

* This work has been supported by the Ministry of Education of Spain (CICYT TIC 91/1036)

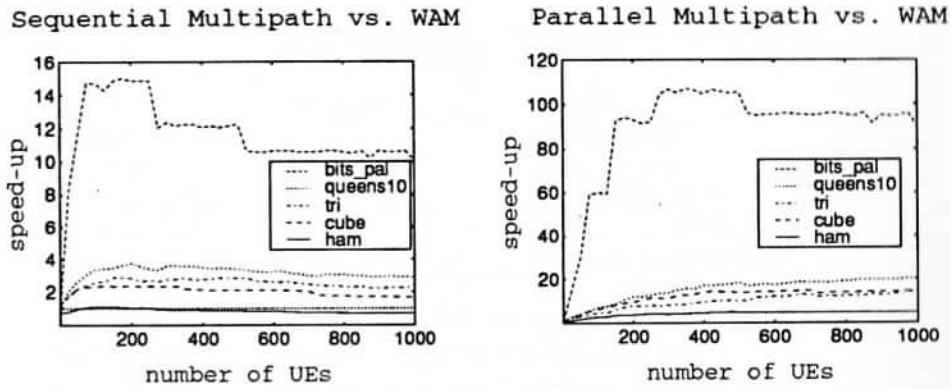

Figure 1: Speed-up of sequential Multipath and parallel Multipath vs. WAM.

2 Performance figures

The implementation of the MEM is done by introducing some extensions to the WAM [1]. This modified WAM is called Multipath Abstract Machine (MAM). The main differences with WAM are:

- The distinction between two types of variables: *single* and *multiple*. The former are the conventional variables used by WAM. The latter are variables that may be bound to multiple values at the same time.
- The introduction of a *Main Engine* (ME), responsible for traversing the SLD-tree, and several *Unification Engines* (UE), performing unification operations on multiple-binding variables.

The ME conducts the search of the SLD-tree. Every time it has to perform an operation on a multiple variable (like unification), it sends a command to all the UEs related to the simultaneously traversed paths. In a sequential execution, the command for each UE is performed one after each other. In a parallel execution, they may be performed concurrently by different *Unification Units* (UU). Several UEs are mapped on each UU.

Figure 1 compares the performance of the Multipath execution model with that of a standard WAM for several benchmarks usually found in the literature. Graphs show the speed-up defined as the execution time of the standard model divided by the execution time of Multipath. In the sequential execution, Multipath outperforms the standard model. In the parallel execution (the number of UUs is 10), we can note that path parallelism is usually found in Prolog programs, and can be efficiently exploited with a small number of UUs.

References

[1] H. Aït Kaci. *Warren's Abstract Machine*. MIT Press, 1991.

[2] J. Tubella and A. González. MEM: A New Execution Model for Prolog. *Microprocessing and Microprogramming, 39*, North-Holland, 1993.