Schema-Based Top-Down Design of Logic Programs Using Abstract Data Types

E. Marakakis and J.P. Gallagher

Department of Computer Science, University of Bristol
Queen's Building, University Walk, Bristol BS8 1TR, U.K.
e-mail: manolis@compsci.bristol.ac.uk, john@compsci.bristol.ac.uk

Abstract. This paper presents a set of schemata that support stepwise top-down design of logic programs using abstract data types (ADTs). There are thus three main components to this approach: top-down design, program schemata and ADTs, all of which are already well established notions. Our aim is to combine these ideas in a framework, amenable to support by design tools and allowing the use of existing logic program transformation techniques to optimise the final programs.

1 Introduction and Motivation

This paper presents a set of schemata that support stepwise top-down design of logic programs using abstract data types (ADTs). There are thus three main components to this approach: top-down design, program schemata and ADTs, all of which are already well-established notions. Our aim is to combine these ideas in a framework, amenable to support by design tools and allowing the use of existing logic program transformation techniques to optimise the final programs. The choice of the set of available schemata and ADTs is very important, and is based on a thorough survey in the literature of the use of these techniques in logic program construction.

The main aims of this paper are

- to justify the approach,
- to show a small set of schemata and ADTs with which we are currently working,
- to discuss future directions of this work.

The main components of our approach are now discussed.

1.1 Top-Down Design

Top-down design is a process of successive refinement of a design. At each stage in the process there is a number of "undefined" parts of the design. A refinement step replaces some undefined part by a more detailed construct, which itself may contain further undefined subparts as well as constructs from some executable target language L. The process is finished when the design is entirely in L.

The main justification of top-down design is that it gives a goal-oriented structure to the design process, and suggests a "compositional" style of program in which each refinement step can be independently justified or proved correct. Apart from this very little is implied by the phrase "top-down design". The actual refinement steps may take many forms.

Our belief is that, if tools to support top-down design are to be practical, refinement steps should correspond to the set of basic constructs in the target language L. That is, each refinement step should introduce one language construct. For example, if L is a Pascal-like language we could insist that each refinement would introduce only one **while**, **repeat**, **if-then-else** or statement composition construct. The target language L therefore plays a very important role in top-down design since each design decision has to be expressed by a single construct. The constructs available in L should be sufficiently high-level to be regarded as part of the design rather than implementation.

In our approach each refinement step introduces either one instance of a program schema or an ADT operation. In a loose sense, therefore, the choice of schemata and ADTs defines a "design language" whose basic constructs consist of the set of schemata and the set of ADT operations.

The separation of schemata from ADTs is an important aspect of our approach since each schema is independent of any particular types. Design decisions therefore need not confuse control with data. This is unlike most schemata in the logic programming literature, which are associated with specific types (such as lists and trees).

1.2 Program Design Schemata

A program design schema for a language M, or simply schema for M, is an expression that represents a set of programs in M having a common form. Each such program is called an instance of the schema. For logic programs, schemata are defined in Section 4. A schema can be identified with the set of its instances. Schemata are abstractions of program structure, analogous to abstractions of data by ADTs.

Our problem is to define a relatively small set of schemata, each member of which captures a useful form of program. Schemata should contain enough structure to be useful in the design process, but be independent of specific data types. This choice is quite difficult. At one extreme, a single schema could be used to construct all programs [24]. On the other hand the literature contains a large number of possible schemata (variously called schemata, clichés, skeletons, and other things). Too many schemata would be confusing and choosing among them would inevitably obscure the design process. Our current set, shown in this paper, consists of five schemata. Although we do not regard this as final by any means, it is important to stress that we do not envisage a growing library of schemata.

Another requirement on our choice of schemata is that they should have both a clear declarative and a procedural reading. Instantiations of schemata should be executable, but have a clear logical structure.

1.3 Abstract Data Types (ADTs)

The concept of type is used to classify objects with distinct characteristics. In programming, a type or data type is a class of data objects or values, such as integers, reals, sets, graphs, cartesian products, and so on. A number of operations is associated with each type which are applied to the values of the type. Abstract data types give a representation-independent specification of a data type. That is, the operations of the data type are defined in terms of axioms that any representation of the type should satisfy.

We also need to introduce the notion of a parameterised ADT. This is an ADT whose components may be of some other unspecified type. For instance a sequence may consist of integers, sets, other sequences, and so on. Thus we could define a type $sequence(\tau)$ where τ is a type variable indicating the type of elements of the sequence.

In our design approach, abstraction is applied to both the algorithm and the data in a problem. In this way the design is separated from any particular representation of the problem. The use of data abstractions is common, but the use of algorithm abstractions is rarer. Yet we maintain that the experienced program designer makes much use of algorithm abstractions or forms.

The result of the top-down design is an executable design or prototype. To summarise our approach, we might replace the well-known equation [23]

Algorithms + Data Structures = Programs

by

Schemata + ADTs = Executable Designs

As a next step in this approach program transformation techniques can be applied to the executable design. These could have the effect of removing the design layer by intertwining it with implementation, or of introducing other standard optimisations. The intention of this step is to improve the performance of the final program.

2 Related Work

2.1 Program Abstractions

Approaches for constructing logic programs based on some kind of abstraction have been followed by many researchers [2], [5], [6], [9], [12], [19], [22]. Their difference apart from technical details is on how they have conceived and used the concept of abstraction. These approaches are classified and discussed based on the kind of abstraction that they use.

Program Skeletons and Clichés. Skeletons are the primary control flow of programs [12], [19], [21]. They are used as the starting point for program development. Techniques are standard Prolog programming practices, e.g. passing back results from computations. Logic programs are constructed by successively enhancing skeletons by applying techniques to them.

Clichés in [2] are defined as commonly occurring program forms that can be reused. Formally clichés are defined to be second-order sentences. The construction process involves instantiation of a cliché with argument values supplied from the programmer.

Abstraction in these approaches is based on control flow similarities between logic programs. These tend to be constructed from commonly occurring program idioms such as processing all elements of a list, and in fact these skeletons tend to be closely related to certain data structures. Experience seems to be the main requirement needed to select such schemata.

Program Schemata. In [9] and [22] a schema is formally defined to be the most specific generalisation [18] of a class of programs. The program construction process in [22] is seen as searching a well-founded ordering of schemata up to certain depth guided by a set of positive and negative examples. Schema instantiation is not discussed in [9]. Schemata for list processing only are shown. Schema classification is based on syntactic similarities.

This approach depends very much on the set of programs that are used for finding the most specific generalisation. This kind of abstraction may result in program schemata with very little structure in them to support the design process. The schemata that can be constructed by that approach are as many as there are programs, that is, infinite. Only small programs have been constructed by this approach. Verification is not discussed in [9] while in [22] program correctness is proved with respect to a set of positive and negative instances that have to be covered and not covered respectively by the predicate under construction.

Logic Description Schemata. A method for constructing logic programs based on structural induction and the generalization paradigm is proposed in [5], [6]. A logic description (LD) is a formula in typeless first-order logic. Two kinds of generalisations are used for the construction of LDs, structural and computational generalisation. Structural induction is the basis of the construction method. The constructed LD is in turn transformed into a logic program. Logic description skeletons are produced automatically from logic description schemata. Undefined predicates in logic description skeletons are defined either automatically using object knowledge or interactively by the user. It is shown that logic descriptions are correct by construction. A logic description schema, i.e. Divide-and-Conquer, is used in [8] to guide the synthesis of logic programs from specifications which consist of examples and properties.

The structural and computational generalisations in [6] correspond to ADTs and to design schemata respectively, from the point of view that both are abstractions over the data and the procedural aspects of a program but they are

different kinds of abstractions. The construction of LD and therefore the whole method is data representation dependent. The Divide-and-Conquer schema in [8] is hardwired in the synthesis mechanism while in our approach the schemata are input to the system. Specifications based on examples have the advantage of simplicity but on the other hand the selection of appropriate examples is not always simple. The synthesis of programs which require multiple applications of the Divide-and-Conquer schema are not discussed in [8]. Construction of small programs is demonstrated in [6] and [8]. The construction of large programs does not seem to be a straightforward task. The use of typeless logic is one reason for their limitation.

2.2 Types

The extension of Prolog programs with mode and type declarations is suggested in [3] in order to enhance their readability and obtain more reliable programs. Types are used to detect errors in cases like mistyping arguments or misplaced arguments. Gödel [11] is a strongly typed logic programming language. Its type system is based on many sorted-logic with parametric polymorphism. A polymorphic type checking system for Prolog is proposed in [17]. The type system in [17] is extended with subtype relations in [7] in order to improve its software engineering properies.

Types in [7], [11] and [17] as in our approach are treated as terms over a special alphabet disjoint from the program's alphabet. Unification and type substitution are over the extended set of type symbols which are treated as ordinary terms. Types in our approach are both inferred during program development and for type checking of calls to ADT predicates. The type systems in [7], [11], [17] perform type checking in constructed programs. They use type declarations to check if the programs are type correct.

2.3 Modes

Modes in [3] are used to validate the flow of data through a clause. The problem of automatically inferring modes for predicates in Prolog programs is studied in [4] for optimization purposes. Modes in [16] are used to optimize Prolog programs. The mode analysis in the literature [4], [16] is mainly used for optimization purposes.

In our approach intended modes are declared for the schemata, the ADT operations and the initial predicate specification. During the design, modes are inferred and can be checked for compatibility with the intended modes.

3 Data Types and Modes

3.1 Data Types

The data types that we consider in this approach are polymorphic types. The type variable τ ranges over types. The alphabet of variables V, function symbols

F and predicate symbols P of the underlying logic language has been extended by the type variables V^τ and the type constructors F^τ such that $V \cap V^\tau = \{\}$ and $F \cap F^\tau = \{\}$. The theoretical basis of the types in this approach is *many sorted logic with parametric polymorphism* [10], [11].

A data type term may consist of *type variables*, *basic types* and *type constructors*. The syntax of data types in BNF production rules is the following

Type ::= τ |
B |
C(Type,..,Type)

τ stands for *type variables*. B stands for the *basic types*, or *constructors of arity 0*. C($Type, .., Type$) stands for constructors of arity ≥ 1. Type terms are defined similarly to first-order terms. That is, *type variables* correspond to *individual variables*, *basic types* correspond to *constants* and *type constructors* to *functions*. A *variable typing* [10], [11], is defined to be a set of the form $\{X_1/\tau_1, .., X_n/\tau_n\}$ where X_i $(1 \leq i \leq n)$ are distinct variables and τ_i are types. Each schema individual variable has been assigned a type variable.

3.2 Abstract Data Types

Unfortunately, programming languages do not have facilities for direct modelling of all real world problems. The problem of expressing in a programming language a situation of the real world can be split in two subproblems:

1. Construct a model of the real world situation.
2. Implement that model using the facilities of a programming language.

The relationships of the real world objects are complex and their modelling requires a powerful language. Mathematics seems to be the most appropriate. Once the modelling has been completed, it can be implemented using the facilities of a programming language. An *abstract data type or data type specification* is a representation-independent specification of data types. Each data type is associated with a set of operations which are defined in terms of axioms that any representation of the type should satisfy. A mathematical model of a real world problem can be constructed by using abstract data types. They provide a valuable way to express real world entities and to organize their relationships.

We have implemented several abstract data types such as *sets, ordered sets, stacks, graphs, cartesian product etc.* The operations have been implemented as predicates.

3.3 Modes

The *mode* of a predicate in a logic program indicates how its arguments will be instantiated when that predicate is called. The mode of arguments of a predicate can be seen in two possible states, that is before the call and after the success of the predicate. The mode of an argument in these two states is called *call-mode*

and *success-mode* respectively. Call modes are meaningful in a logic program only when the control strategy has been specified. The program schemata in this approach have been constructed with the left-to-right depth-first computation rule of Prolog in mind. The modes `i` and `d` are used. They stand for "input" and "don't know" respectively. `i` identifies the arguments which are expected to be ground. These terms are characterized as "input". An argument with any instantiation can be assigned mode `d`. Each schema individual variable has been assigned an "intended" mode of use. Modes in this approach are used to guide the application of schemata to undefined predicates and in the refinement of undefined predicates by ADT operations or equality.

Given a partial design and a moded initial goal, a mode inference procedure computes call and success modes as far as possible. These inferred modes are used to check the modes of calls to ADT predicates for compatibility with the intended modes. Our mode inference procedure is an abstract interpretation similar in many respects to other mode inferrence procedures such that in [4]. It is beyond the scope of this paper to describe it in detail.

4 Program Design Schemata

Algorithm design techniques are important in constructing programs [1], [13]. It has been observed that different problems are solved by using the same algorithm. If we take a collection of solution programs for such problems and factor out the differences relating to the problem representation, we obtain the algorithm common to all of them. We call such a problem-independent algorithm a *program design schema*, or *schema*. Such design schemata contain a fixed set of subparts or subgoals, that have different realisations/instantiations for each problem that they are applied to.

4.1 The Design Schema Language

In this section we consider a small set of program design schemata that we find useful for constructing logic programs. Their classification is based on the design strategy that each schema represents.

The schemata are formally defined as expressions in a meta-language of first-order logic. The meaning of the symbols of the meta-language is the following.

- the capital letters X, Y, Z, W, U, V, T, S are schema individual variables. They range over object-language variables or tuples of such variables.
- Other capitalized words are predicate variables. They range over predicate symbols.
- The meta-language logical symbols $\wedge$, $\neg$ and $\leftarrow$ have their usual meaning.

For each schema individual variable its type and mode are shown as well. For example, $(X^{\tau}, \texttt{i})$ stands for the schema individual variable X whose type is τ and mode `i`. In the following, τ_1, τ_2, τ_3 stand for type variables.

4.2 Five Design Schemata

We consider schemata named as follows, and discuss each in turn.

1. Incremental
2. Subgoal
3. Divide-and-conquer
4. Case
5. Backtracking

The predicate variables are given names that indicate their intended meaning.

Incremental. The *Incremental* schema processes one by one each piece of the input data and composes the results for each one to construct the solution. The schema is as follows.

$$\begin{aligned} &Incr((X^{\tau_1},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \leftarrow \\ &\qquad Terminating((X^{\tau_1},\mathtt{i})) \wedge \\ &\qquad Initial_result((X^{\tau_1},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \end{aligned}$$

$$\begin{aligned} &Incr((X^{\tau_1},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \leftarrow \\ &\qquad \neg Terminating((X^{\tau_1},\mathtt{i})) \wedge \\ &\qquad Elem_by_elem((X^{\tau_1},\mathtt{i}),(V^{\tau_3},\mathtt{d}),(W^{\tau_1},\mathtt{d})) \wedge \\ &\qquad Incr((W^{\tau_1},\mathtt{i}),(Z^{\tau_2},\mathtt{d})) \wedge \\ &\qquad Non_initial_result((X^{\tau_1},\mathtt{i}),(V^{\tau_3},\mathtt{i}),(Z^{\tau_2},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \end{aligned}$$

Subgoal. The *Subgoal* schema reduces the problem into a sequence of two subproblems. The solutions of the simpler problems are composed into a solution of the original problem. This is, an AND reduction. Note that V could be empty as well. The schema is as follows.

$$\begin{aligned} &SubGoal((X^{\tau_1},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \leftarrow \\ &\qquad SubGoal1((X^{\tau_1},\mathtt{i}),(V^{\tau_3},\mathtt{d}),(Y^{\tau_2},\mathtt{d})) \\ &\qquad SubGoal2((X^{\tau_1},\mathtt{i}),(V^{\tau_3},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \end{aligned}$$

Divide-and-conquer. The *Divide-and-conquer* schema decomposes the problem in two subproblems. The solution of the problem is constructed by composing the solutions of the subproblems together. The schema is of the following form.

$$\begin{aligned} &DivConq((X^{\tau_1},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \leftarrow \\ &\qquad Terminating((X^{\tau_1},\mathtt{i})) \wedge Initial_result((X^{\tau_1},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \\ &DivConq((X^{\tau_1},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \leftarrow \\ &\qquad \neg Terminating((X^{\tau_1},\mathtt{i})) \wedge \\ &\qquad Decomposition((X^{\tau_1},\mathtt{i}),(W^{\tau_1},\mathtt{d}),(Z^{\tau_1},\mathtt{d})) \wedge \\ &\qquad DivConq((W^{\tau_1},\mathtt{i}),(U^{\tau_2},\mathtt{d})) \wedge DivConq((Z^{\tau_1},\mathtt{i}),(V^{\tau_2},\mathtt{d})) \wedge \\ &\qquad Composition((U^{\tau_2},\mathtt{i}),(V^{\tau_2},\mathtt{i}),(Y^{\tau_2},\mathtt{d})) \end{aligned}$$

Case. The *Case* schema reduces the problem in two independent subproblems. Each subproblem corresponds to a different case of the original problem. This is, an *OR* reduction. The schema is as follows.

$$
\begin{aligned}
&Case((X^{\tau_1}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d})) \leftarrow \\
&\qquad Case1((X^{\tau_1}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d})) \\
&Case((X^{\tau_1}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d})) \leftarrow \\
&\qquad Case2((X^{\tau_1}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d}))
\end{aligned}
$$

Backtracking. The *Backtracking* schema does an organized exhaustive search in the solution space. Solutions are constructed in a stepwise fashion. Note that the solution returned may be composed from the complete search path. The schema is as follows.

$$
\begin{aligned}
&Backtr((X^{\tau_1}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d})) \leftarrow \\
&\qquad Initial_search_node((X^{\tau_1}, \mathtt{i}), (W^{stack(\tau_3)}, \mathtt{d}), (U^{\tau_2}, \mathtt{d})) \wedge \\
&\qquad Backtr1((X^{\tau_1}, \mathtt{i}), (W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d})) \\
&\\
&Backtr1((X^{\tau_1}, \mathtt{i}), (W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d})) \leftarrow \\
&\qquad Solution_complete((W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i})) \wedge \\
&\qquad Solution_assignment((U^{\tau_2}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d}) \\
&Backtr1((X^{\tau_1}, \mathtt{i}), (W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d})) \leftarrow \\
&\qquad \neg Solution_complete((W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i})) \wedge \\
&\qquad Forward_move((X^{\tau_1}, \mathtt{i}), (W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i}), \\
&\qquad\qquad (Z^{stack(\tau_3)}, \mathtt{d}), (V^{\tau_2}, \mathtt{d})) \wedge \\
&\qquad Backtr1((X^{\tau_1}, \mathtt{i}), (Z^{stack(\tau_3)}, \mathtt{i}), (V^{\tau_2}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d})) \\
&Backtr1((X^{\tau_1}, \mathtt{i}), (W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d})) \leftarrow \\
&\qquad \neg Solution_complete((W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i})) \wedge \\
&\qquad \neg Forward_move((X^{\tau_1}, \mathtt{i}), (W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i}), \\
&\qquad\qquad (Z^{stack(\tau_3)}, \mathtt{d}), (V^{\tau_2}, \mathtt{d})) \wedge \\
&\qquad Backtracking((W^{stack(\tau_3)}, \mathtt{i}), (U^{\tau_2}, \mathtt{i}), \\
&\qquad\qquad (T^{stack(\tau_3)}, \mathtt{d}), (S^{\tau_2}, \mathtt{d})) \wedge \\
&\qquad Backtr1((X^{\tau_1}, \mathtt{i}), (T^{stack(\tau_3)}, \mathtt{i}), (S^{\tau_2}, \mathtt{i}), (Y^{\tau_2}, \mathtt{d}))
\end{aligned}
$$

5 Schema Instantiation

A *program design schema* is instantiated with respect to an undefined predicate.

Let $p(X_1, .., X_n)$ be an undefined predicate with variable typing

$$\Omega = \{X_1/\tau_1, ..., X_n/\tau_n\}.$$

In the following, *mode(X)* stands for the mode of argument X. Let

$$\overline{I} = (Y_1, .., Y_{n1}) : mode(Y_j) = i,\ Y_j \in \{X_1, .., X_n\}$$

$$\overline{D} = (Z_1, .., Z_{n2}) : mode(Z_j) = d,\ Z_j \in \{X_1, .., X_n\}$$

$\{Y_1, .., Y_{n1}\} \cup \{Z_1, .., Z_{n2}\} = \{X_1, .., X_n\}$. The variables in the tuples $\overline{I}$ and $\overline{D}$ respectively have the same order that they have in $p(X_1, .., X_n)$.

Tupling is an operation that is applied on the undefined predicate p/n as follows:

$$tupling(p(X_1, .., X_n)) = p(\overline{I}, \overline{D})$$

Let $\{C_1, .., C_r\}$ be a variant of schema Σ. Each clause C_i $(1 \leq i \leq r)$ has the form $H_i \leftarrow B_i$, where H_i is an atom and B_i is a conjunction of literals.

A *schema substitution* Ψ_i for clause C_i of Σ with respect to $p(X_1, .., X_n)$ is defined as the composition of the following substitutions.

$$\Psi_i = \Theta_i \circ \Delta_i$$

where

- $\Theta_i = mgu(p(\overline{I}, \overline{D}),\ H_i)$. That is, Θ_i is a substitution of schema predicate variables and individual variables that occur in H_i.
- Δ_i is a substitution of the schema predicate variables in B_i that do not occur in H_i.

Schema predicate variables are substituted by fresh predicate symbols that have not been used before. The instance $I^{'}$ of Σ which defines $p(\overline{I}, \overline{D})$ is defined to be

$$I^{'} = \{C_1\Psi_1, .., C_r\Psi_r\} = \{C_1^{'}, ..., C_r^{'}\}$$

Let A be an atom and Ω a variable typing of A. $A\Omega$ denotes the application of variable typing Ω on the atom A. Similarly let C be a clause and Ξ a variable typing of C. $C\Xi$ denotes the application of variable typing Ξ on the clause C.

Let Ξ_i be the variable typing of clause C_i in schema Σ. Let

$$\phi_i^{\tau} = mgu(p(\overline{I}, \overline{D})\ \Omega, H_i\ \Xi_i)$$

Then the *derived variable typing* for $C_i^{'}$ $(1 \leq i \leq r)$ is defined to be

$$\Pi_i = \{X/\sigma\phi_i^{\tau} : X/\sigma \in \Omega \cup \Xi_i\}$$

Any variable X in $C_i^{'}$ of $I^{'}$ with variable typing of the form $(\tau_1, .., \tau_k)$ is replaced by the tuple of variables $(X_1, .., X_k)$. The variable typing Π_i of $C_i^{'}$ is extended by such variable typings, i.e. $\{X_1/\tau_1, .., X_k/\tau_k\}$. The new variable typing $\Pi_i^{'}$ for $C_i^{'}$ consists from the union of Π_i and such variable typings.

detupling(A) where A is an atom is defined to be detupling(A) = p$(X_1, .., X_r)$ where $\{X_1, .., X_r\}$ = vars(A) and $X_1, .., X_r$ have the same order as in A.

A detupling operation on clause C of the form $H \leftarrow L_1, .., L_l$ is defined to be

detupling$(H \leftarrow L_1, .., L_l)$ = detupling(H) $\leftarrow$ detupling(L_1),..,detupling(L_l)
detupling$(L_i) = \neg$ detupling(A_i), $(1 \leq i \leq l)$

where A_i is the atom of L_i.

The instance I of schema Σ that defines the predicate $p(X_1, .., X_n)$ is defined to be

$I = \{\text{detupling}(C_1^{'}),..,\text{detupling}(C_r^{'})\}$

Example:

Suppose that the *Incremental* schema Σ is applied to the undefined predicate p(X1, X2, X3). The variable typing of p(X1, X2, X3) is Ω = {X1/set(str), X2/τ, X3/orderedSet(str)} where τ is a type variable, and the "intended" modes of its arguments are mode(X1) = i, mode(X2) = i, mode(X3) = d.

tupling(p(X1, X2, X3) = p((X1,X2), (X3)) = p($\overline{I}$,$\overline{D}$)

Assuming that a renamed substitution for *Incremental* schema is

$$
\begin{array}{l}
\{\ C_1, C_2\ \} = \\
\{\quad Incr(X^{\tau_1}, Y^{\tau_2}) \ \leftarrow \\
\qquad\qquad Terminating(X^{\tau_1}) \ \wedge \ Initial_result(X^{\tau_1}, Y^{\tau_2}) \\
\\
\qquad Incr(X^{\tau_1}, Y^{\tau_2}) \leftarrow \\
\qquad\qquad \neg Terminating(X^{\tau_1}) \ \wedge \ Elem_by_elem(X^{\tau_1}, V^{\tau_3}, W^{\tau_1}) \wedge \\
\qquad\qquad Incr(W^{\tau_1}, Z^{\tau_2}) \ \wedge \ Non_initial_result(X^{\tau_1}, V^{\tau_3}, Z^{\tau_2}, Y^{\tau_2})\ \}
\end{array}
$$

where, τ_1, τ_2, τ_3 are the types corresponding to each schema individual variable. The variable typing for C_1 is $\Xi_1 = \{X/\tau_1, Y/\tau_2\}$. The variable typing for C_2 is $\Xi_2 = \{X/\tau_1, Y/\tau_2, V/\tau_3, W/\tau_1, Z/\tau_2\}$

The *schema substitution* $\Psi_i = \Theta_i \circ \Delta_i$ $i \leftarrow \{1,2\}$ with respect to p($\overline{I}$,$\overline{D}$) is the following

$\Theta_1 = \Theta_2 = \text{mgu}(\text{p}(\overline{I},\overline{D}), \text{Incr(X,Y)}) = \{$ Incr/p, X/(X1,X2), Y/(X3) $\}$
$\Delta_1 = \{$ Terminating/q1, Initial_result/q2 $\}$
$\Delta_2 = \{$ Terminating/q1, Elem_by_elem/q3, Non_initial_result/q4 $\}$

$$
\begin{array}{l}
I^{'} = \{C_1\Theta_1\Delta_1, C_2\Theta_2\Delta_2\} = \{C_1^{'}, C_2^{'}\} \\
\quad = \{\text{p((X1,X2), (X3))} \leftarrow \\
\qquad\qquad \text{q1((X1,X2))} \ \wedge \\
\qquad\qquad \text{q2((X1,X2), (X3))} \\
\\
\qquad \text{p((X1,X2), (X3))} \leftarrow \\
\qquad\qquad \neg\ \text{q1((X1,X2))} \ \wedge \\
\qquad\qquad \text{q3((X1,X2), V, W)} \ \wedge \\
\qquad\qquad \text{p(W, Z)} \ \wedge \\
\qquad\qquad \text{q4((X1,X2), V, Z, (X3))}\ \}
\end{array}
$$

Let Φ_1^{τ} and Φ_2^{τ} be two type substitutions such that

$$\Phi_1^\tau = \text{mgu}((\text{p}(\overline{I},\overline{D})\ \Omega), (\text{Incr(X,Y)}\ \Xi_1)) =$$
$$\{\ \tau_1/(\text{set(str)},\tau),\ \tau_2/(\text{orderedSet(str)})\ \}$$
$$\Phi_1^\tau = \Phi_2^\tau = \text{mgu}((\text{p}(\overline{I},\overline{D})\ \Omega), (\text{Incr(X,Y)}\ \Xi_2)) =$$
$$\{\ \tau_1/(\text{set(str)},\tau),\ \tau_2/(\text{orderedSet(str)})\ \}$$

The variable typings of the clauses $\{C_1\Theta_1\Delta_1, C_2\Theta_2\Delta_2\}$ are the following

$$\Pi_1 = \{\Omega\Phi_1^\tau\} \cup \{\Xi_1\Phi_1^\tau\}$$
$$\Pi_2 = \{\Omega\Phi_2^\tau\} \cup \{\Xi_2\Phi_2^\tau\}$$

$$\Pi_1 = \{\ \text{X1/set(str)},\ \text{X2}/\tau,\ \text{X3/orderedSet(str)},$$
$$\text{X}/(\text{set(str)},\tau),\ \text{Y}/(\text{orderedSet(str)})\}$$
$$\Pi_2 = \{\ \text{X1/set(str)},\ \text{X2}/\tau,\ \text{X3/orderedSet(str)},$$
$$\text{X}/(\text{set(str)},\tau),\ \text{Y}/(\text{orderedSet(str)}),$$
$$\text{V}/\tau_3,\ \text{W}/(\text{set(str)},\tau),\ \text{Z}/(\text{orderedSet(str)})\}$$

I = { detupling($C_1^{'}$), detupling($C_2^{'}$) } =
= {p(X1, X2, X3) ←
q1(X1, X2) ∧
q2(X1, X2, X3),

p(X1, X2, X3) ←
¬ q1(X1, X2) ∧
q3(X1, X2, V, W1, W2) ∧
p(W1, W2, Z1) ∧
q4(X1, X2, V, Z1, X3) }

$$\Pi_1^{'} = \Pi_1 \cup \{\}$$
$$\Pi_2^{'} = \Pi_2 \cup \{\text{W1/set(str)},\ \text{W2}/\tau,\ \text{Z1/orderedSet(str)}\}$$

6 Refinement by ADT or Equality Predicates

The definition of an undefined predicate $p(X_1, ..X_n)$ in terms of an ADT or equality predicate will create a new clause of either form

$$p(X_1, ..X_n) \leftarrow R(Y_1, .., Y_m)\ m \leq n$$

$$p(X_1, ..X_n) \leftarrow \neg R(Y_1, .., Y_m)\ m \leq n$$

where R is an ADT operation predicate symbol or the equality symbol and $\{Y_1, ..Y_m\} \subseteq \{X_1, ..X_n\}$. The variables in the clause are typed according to the declared types of ADTs.

The call-success modes generated by the mode inference procedure are used to validate the call modes of $R(Y_1, .., Y_m)$. That is, the call mode of $R(Y_1, .., Y_m)$ is subsumed by its intended mode.

7 The Design Process

Initially, the programmer gives a specification of the predicate that he wants to construct. That specification consists of the *data type* and the *mode* of each argument. Goals are expected to have ground the arguments with mode `i`, arguments with mode `d` can be either ground or nonground or partially ground. The initial refinement is applied to that predicate. The next refinements are applied to undefined subgoals of the clauses created. Each refinement step involves either of the following actions.

- The application of a design schema.
- The definition of the predicate being refined in terms of an ADT or equality predicate or its negation.

The construction process is a successive top-down application of refinements until the complete construction of an executable design of the system. A design is considered to be complete when all of its predicates are defined.

The above discussion is illustrated by applying an example with five refinements. That is, three applications of design schemata and two predicate definitions in terms of ADT predicates. The effect of the application of each schema will be shown, not the instantiation details of the schema.

1. *Predicate specification*: p((I,i,set(τ)), (D,d,orderedSet(τ)))
2. Refinement 1 applied on p/2: *Incremental schema.*

```
p(I, D) :-
        q1(I),
        q2(I, D).
p(I, D) :-
        not q1(I),
        q3(I, X, RI),
        p(RI, RD),
        q4(I, X, RD, D).
```

3. Refinement 2 applied on q2/2: *Subgoal schema.*

```
q2(I, D) :-
        t1(I, Y, D),
        t2(I, Y, D).
```

4. Refinement 3 applied on q3/3: *Subgoal schema.*

```
q3(I, X, RI) :-
        r1(I, X, RI),
        r2(I, X, RI).
```

5. Refinement 4 applied on r1/3: *Abstract data type operation get_elem/2.*

```
r1(I, X, RI) :- get_elem(I, X).
```

6. Refinement 5 applied on r2/3: *Abstract data type operation delete/3*

```
r2(I, X, RI) :- delete(I, X, RI).
```

7.1 Examples

Insertion sort. insert_sort(S1, S2) is true if S2 is the set S1 with its elements sorted.

- *Predicate specification:* insert_sort((S1,i,set(τ)), (S2,d,orderedSet(τ)))

Refinements.
Schema: Incremental

```
insert_sort(S1, S2) :-
   terminating(S1),
   construct_initial_result(S1, S2).
insert_sort(S1, S2) :-
   \+ terminating(S1),
   get_elem_incr(S1, Elem, RestS1),
   insert_sort(RestS1, RestS2),
   construct_non_initial_result(S1, Elem, RestS2, S2).

terminating(S1) :- empty_set(S1).

construct_initial_result(S1, S2) :- empty_ordSet(S2).
```

Schema: Subgoal

```
get_elem_incr(S1, Elem, RestS1) :-
    get_element(S1, Elem, RestS1),
    delete_element(S1, Elem, RestS1).

construct_non_initial_result(S1, Elem, RestS2, S2) :-
    set_elem_ordSet(Elem, RestS2, S2).

get_element(S1, Elem, RestS1) :- get_elem(S1, Elem).

delete_element(S1, Elem, RestS1) :- delete(Elem, S1, RestS1).
```

Abstract Data Type Operations.
ADT operations for unordered sets

```
- empty_set(S)
- delete(X, S1, S2)
- get_elem(S, Elem)
```

ADT operations for ordered sets

```
- empty_ordSet(S)
- set_elem_ordSet(X, S1, S2)
```

8 Conclusions and Further Work

Five novel features in the development of logic programs are introduced by this appoach. First, the problem domain representation, i.e. the abstract data types, are separated from the structural or control part of the algorithm, i.e. the program schemata. Second, the set of program schemata consists of only five schemata, making them manageable by humans. Third, the application of a program schema to an undefined predicate is directed by modes. Fourth, the constructed program designs are type correct. Fifth, static analysis is performed dynamically during program construction.

So far, we find this small set of schemata surprisingly expressive. Larger examples of constructed programs, e.g. depth-first traversal of a graph, can be found in [15]. Related work in functional programming languages has been followed in [20]. One strategy, i.e. divide and conquer, is used in [20] for automatically constructing programs from formal specifications.

A set of support tools for program development using these ideas is under development. The question of heuristics for selecting the appropriate schema or ADT operation at each refinement step is also a subject that requires study.

The transformation of the executable design into a more efficient program will also be investigated. The fact that executable logic programs are produced means that state of the art transformation methods can be brought to bear. Techniques such as partial evaluation have already been applied to removing the overhead of ADTs [14]. Other deeper transformations include those based on unfold/fold and coroutining between goals.

Our approach produces type-correct designs, and the role of ADTs is important for this. On the question of verification in general, we have not assumed the presence of a full formal specification of the problem (that is, a detailed formalisation of the input/output relation of the program). We have considered only questions of algorithm design and have assumed that some aspects of specification are subsumed in the design process. However, higher level specification languages that can be mapped to the schemata could be investigated.

References

1. A. Aho, J. Hopcroft and J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.
2. D. Barker-Plummer, *Cliché Programming in Prolog*, Proceedings of the Second Workshop on Meta-programming in Logic, (ed. M. Bruynooghe); April 1990, Belgium, pp.247-256.
3. M. Bruynooghe, *Adding Redundancy to Obtain More Reliable and More Readable Prolog Programs*, Proceedings of the First International Logic Programming Conference, Sept. 1982, Marseille, France, pp.129-133.
4. S. Debray, D. Warren, *Automatic Mode Inference for Prolog Programs*, Proceedings of the Symposium on Logic Programming, Salt Lake City, Utah, Sept. 1986, pp.78-88.

5. Y. Deville, J. Burnay, *Generalization and Program Schemata: A Step Towards Computer-Aided Construction of Logic Programs*, Proc. of the North American Conference on Logic Programming, 1989, (eds. E. Lusk and R. Overbeek); pp.409-425, vol.I,.
6. Y. Deville, *Logic Programming: Systematic Program Development*, Addison-Wesley, 1990.
7. R. Dietrich, F. Hagl, *A Polymorphic Type System with Subtypes for Prolog*, ESOP'88 Proceedings of the 2nd European Symposium on Programming, Nancy, France 1988, Lecture Notes in Comp. Sc. no 300, pp.79-93.
8. Pierre Flener, Yves Deville, *Towards Stepwise, Schema-Guided Synthesis of Logic Programs*, Proceedings of International Workshop on Logic Program Synthesis and Transformation, LOPSTR'91, Manchester, U.K., 1991, pp.46-64.
9. T. Gegg-Harrison, *Basic Prolog Schemata*, Technical Report, CS-1989-20, Dept. of Computer Science, Duke University, Durham, North Carolina, Sept. 1989.
10. P. Hill, and R. Topor, *A Semantics for Typed Logic Programs*, Chapter 1 in *Types in Logic Programming* edited by F. Pfenning, The MIT Press, 1992.
11. P. Hill, J. Lloyd, *The Gödel Programming Language*, The MIT Press, 1994.
12. A. Lakhotia, *A Workbench for Developing Logic Programs by Stepwise Enhancement*, PhD thesis, Case Western Reverse University, Department of Computer Engineering and Science, Aug. 1989.
13. J. Kingston, *Algorithms and Data Structures, Design, Correctness, Analysis*, Addison-Wesley, 1990.
14. J. Komorowski, Towards a Programming Methodology Founded on Partial Deduction, Proc. of the European Conference on Artificial Intelligence, Stockholm, Sweden, August 1990.
15. E. Marakakis, J.P. Gallagher, *Schema-Based Top-Down Design of Logic Programs Using Abstract Data Types*, Technical Report, CSTR-94-02, University of Bristol, Department of Computer Science.
16. C. Mellish, *Some Global Optimizations for a Prolog Compiler*, Journal of Logic Programming, 1, 1985, pp.43-66.
17. A. Mycroft, R. O'Keefe, *A Polymorphic System for Prolog*, Artificial Intelligence, 23, 1984, pp. 295-307.
18. G. Plotkin, *A Note on Inductive Generalization*, Machine Intelligence, 5, 1970, pp.153-163.
19. D. Robertson, *A Simple Prolog Techniques Editor for Novice Users*, Proceedings of the 3^{rd} Annual Conference on Logic Programming, Edinburgh, April 1991, pp.78-85.
20. D. Smith, *Top-Down Synthesis of Divide-and-Conquer Algorithms*, Artificial Intelligence, vol. 27, 1985, pp.43-96.
21. L. Sterling, M. Kirschenbaum, *Applying Techniques to Skeletons*, Chapter 6 in *Constructing Logic Programs* edited by Jean-Marie Jacquet, Wiley, 1993.
22. N. Tinkham, *Induction of Schemata for Program Synthesis*, PhD thesis, Department of Computer Science, Duke University, 1990.
23. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.
24. T. Yokomori, *Logic Program Forms*, New Generation Computing, No.4, 1986, pp.305-319.

This article was processed using the LaTeX macro package with LLNCS style