# Skeletons and Techniques as a Normative Approach to Program Development in Logic-based Languages

*Technical Report OSU-CISRC-5/94-TR25*

Marc Kirschenbaum* Spiro Michaylov† Leon Sterling‡

May 13, 1994

**Abstract**

To help encourage the wide use of a variety of logic-based languages, we advocate the identification and use of systematic program construction methods. Our approach is to capture programming idioms in the form of skeletons, or basic control flows, and standard programming techniques for construction and manipulation of data structures. We argue that the approach, coming initially from Prolog and logic grammars, is equally applicable in the context of constraints, committed-choice and higher order constructs, and indeed logic-based languages in general. This paper introduces our generalized view of skeletons and techniques, gives examples of the breadth of their applicability, and explores positive consequences for program development.

## 1 The Need for a Normative View

Most proponents of logic programming, after having been excited about the advantages of logic programming, have puzzled why the technology has not become more widely adopted. Despite obvious advantages, logic programming has not greatly impacted software development. Why is this so?

One explanation is the lack of a proven development method. Because of significant differences between programming with logic languages and programming with imperative languages, it has not been obvious how to adapt conventional development methods to logic programming. People not immediately taken with logic programming dismiss it, remarking that programming is too hard or it is too inefficient or impractical in some undocumented way. There has been a lack of teaching material that stresses how to incorporate logic programming components. Furthermore, there has been a lack of ready-made software for system components such as user interfaces, and the software that is available is not widely accessible. We hear anecdotal evidence from industry that managers are fearful of adopting new programming approaches where specialized skills are needed and may not be available, especially in the U.S.

*John Carroll University, Cleveland, OH 44118, U.S.A., kirsch@jcvaxa.jcu.edu

†Department of Computer and Information Science, The Ohio State University, 228 Bolz Hall, 2036 Neil Avenue Mall, Columbus, OH 43210-1277, U.S.A., spiro@cis.ohio-state.edu

‡Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH 44106, U.S.A., leon@ces.cwru.edu

This paper is focussed on program development using languages based on logic programming. We claim that standard programming methods can be, and indeed have been identified, and can be incorporated into a standard practice. We discuss basic programming idioms that have emerged from programming experience, and which we have previously categorized for Prolog into skeletons or basic control flows and techniques, which implement programming segments for construction and manipulation of data structures [16, 5].

In this paper, we point out that the language of skeletons and techniques, developed for Prolog and implicit in discussion of logic grammars [13], is applicable to logic-based languages in general. Sections 2 and 3 explain what we mean by skeletons and techniques and give examples from a range of languages including Prolog, CLP($\mathcal{R}$), Flat Concurrent Prolog and $\lambda$Prolog. More generally, in these sections we enumerate and describe the major issues that need to be addressed in order to establish a core set of skeletons and techniques for a particular logic-based language. In so doing, we establish the normative role of this methodology for logic-based languages: it is a uniform strategy that can be applied to a newly designed language to more quickly enable it to be ready for use by programmers at large.

There are positive consequences from structuring a logic program in terms of skeletons and techniques. The most obvious is ease of explanation of (possibly) complicated programs and the documentation that such a structuring entails. Another advantage is the potential for detailed program analysis leading, for example, to proofs of correctness and guidelines for optimization. Still another advantage is ease of program maintenance. These advantages are discussed in Section 4.

Section 4 also addresses support tools. We have implemented a prototype tool for constructing Prolog programs using skeletons and techniques. The tool suggests a process of program development where a consistent programming style is encouraged. Section 5 concludes the paper with some observations about skeletons and techniques.

Skeletons and especially techniques can be viewed as higher order programming constructs. This is not a view we encourage, and more importantly, is not the point of the paper. We find that students intuitively grasp programming techniques and are able to use them quickly and effectively. We do not believe that would be the case if techniques were explained as an instance of a second-order programming schema: much of the benefit comes from providing programmers with a set of specific, easy to follow skeletons and techniques. There is a role for investigating the theoretical underpinnings of skeletons and techniques. One approach we advocate is program maps as introduced in [5] and continued in [6]. This paper is concerned with the development of clear, effective programming practice for logic programming languages.

## 2 Skeletons

In a logic-based language, a skeleton is a piece of code that describes how a term from the language is to be "traversed". The skeleton provides:

- an inductive definition of the term(s) to be traversed,
- a strategy for traversing the term(s),
- templates for extracting data from the term(s), and
- conditions under which term traversal may terminate before being completed.

While the above aspects of skeletons are invariant with respect to language, their exact form, and even the purpose, is largely language dependent. It is influenced by the syntax of the language, its computational primitives (unification, constraint satisfaction, etc.) and its control features. In the following discussion, we will frequently make a distinction (introduced in [10]) between terms of a language on the basis of their being "structural" or "non-structural". This is based on the idea that inductively defined terms such as lists and trees can be constructed and destructed my manipulating individual components (structural) while terms such as numbers relate to others of the same type by computation, but have no clear syntactic relationship.

To develop a core set of skeletons for a language, we need to consider the following viewpoints, based on some experience with developing applications in the language and teaching the language.

1. *Important data structures from terms in the language.*
   Since skeletons primarily specify how terms will be traversed, the dominant factor in the design of skeletons is the set of useful data structures that can be constructed from these terms, or alternately how terms can be considered as control parameters. The latter is the more appropriate view for non-structural skeletons. For example, in Prolog we wish to identify the most important tree/list structures, in a committed choice languages we have processes and communication channels, and in higher-order LP languages we consider the kinds of object languages we wish to manipulate.

2. *Appropriate views of structures being traversed*
   As a structure is being traversed step by step, each such computational step has a certain limited view of the structure. How this view is organized will have a significant effect on the kinds of computation that can easily be threaded along the structure by techniques for this skeleton. For example, the typical view of a list being traversed in Prolog is one list element, but in some cases being able to "view" two or more adjacent list elements at once is more appropriate. Sometimes it is useful to split a list for 'divide-and-conquer'. A closely related question is that of how the view at the next computational step relates to that at the present step. For example, a Prolog skeleton that views two adjacent list elements at one step may view the next two at the next step, or the second of the original two together with the next adjacent element. For non-structural skeletons, the corresponding approach may be to have parameters for prior values of a given control parameter.

3. *Major data structure traversal strategies*
   Different applications require structures to be traversed in different ways. Traversals differ primarily in how much of the structure is traversed in a successful computation, how the structure is searched for that traversal, and the order of the traversal. To understand this precisely, we need the notion of traversal sequence corresponding to that of a derivation sequence. The traversal sequence is the sequence of elements of a structure that are visited by the derivation steps in a derivation sequence. Then the search for a successful derivation corresponds directly to a desirable traversal of the data structure. Different skeletons then result in different sequences of attempted traversal sequences depending on their search strategy, and also in traversals of different amounts of the structure in different orders. For example, a Prolog skeleton traversing a binary tree may deterministically find an inorder (or pre or post order) traversal, or it may search for a path from the root to a leaf. The corresponding notion for non-structural skeletons lies in the different ways a control parameter can be computed from its previous value.

4. *Useful traversal early termination conditions*
   Structural traversals may involve early termination, either with success or failure. A search may terminate when a particular datum is found in a structure, or when it is determined that

the datum will not be present. Examples in Prolog include cases where a fact matches on something other than a leaf node in a tree.

Let us now consider some concrete examples. We begin with a Prolog skeleton for inorder traversal of a binary tree. Notice that this skeleton both defines the binary tree structure in terms of the functors `leaf/0` and `node/3`, and also describes the actual order of traversal. The logical consequences of this program contain exactly the ground binary trees, but many skeletons that are not useful to us from this viewpoint have the same consequences. This skeleton is desired because of the way it interacts with the operational model.

```
tree_io(leaf).
tree_io(node(K,L,R)) :-
        tree_io(L),
        tree_io(R).
```

The following skeleton also defines binary trees inductively, but instead of an inorder traversal, it specifies a depth-first, left-right search to individual leaves. This distinction is not reflected by examining its logical consequences.

```
tree_df(leaf).
tree_df(node(K,L,R)) :- tree_df(L).
tree_df(node(K,L,R)) :- tree_df(R).
```

To see how broadly applicable such skeletons are, we show another full traversal skeleton, this time for trees representing resistive circuits in terms of resistors and parallel and series combinations of sub-circuits.

```
circuit(res(R)).
circuit(ser(C1,C2)) :-
        circuit(C1),
        circuit(C2).
circuit(par(C1,C2)) :-
        circuit(C1),
        circuit(C2).
```

The three skeletons above are structural. Good examples of non-structural skeletons can be taken from languages performing arithmetic. A paradigmatic example is using an integer value to control iteration. Here we may consider the term being traversed, i.e. the changing integer value, as a control parameter, which determines the next step of a computation but is also recomputed by each computation step.

The following CLP($\mathcal{R}$) example shows a parameter being used to control a computation in much the same way that a structural term is traversed by destructing it at each step.

```
step(T) :-
        T = 1.
step(T) :-
        T > 1,
        step(T - 1).
```

Non-structural terms give us considerable flexibility, because of the wide variety of ways we can compute their next values and test for termination. The example below of `divide_step` is a different skeleton from `step` precisely because next values are computed differently and the termination test is different. A distinction can be made between structural terms and non-structural terms according to the ways that they terminate. Structural terms can terminate because the 'base structure' has been reached or because early termination is dictated by the skeleton. Non-structural terms have no analogue of the former type, reaching a 'null' element.

```
divide_step(T) :-
        T < 1.
divide_step(T) :-
        T > 1000, T1 = T / 2, divide_step(T1).
divide_step(T) :-
        T <= 1000, T >= 1, T1 = T / 3, divide_step(T1).
```

The skeletons presented so far are useful for Prolog and CLP($\mathcal{R}$). but are also useful for parallel logic programming languages. The skeleton `tree_io` searches both branches of a tree concurrently in a logic language that exploits AND-parallelism. The skeleton `tree_df` searches both branches of a tree concurrently in a logic language that exploits OR-parallelism.

Another example of a skeleton that is useful both for Prolog and parallel languages is a meta-interpreter. The terms that a meta-interpreter traverses, conjunctive goals, are dynamic. The following meta-interpreter for Flat Concurrent Prolog, taken from [15], essentially spawns a number of goal processes. A similar meta-interpreter for Strand is given in [4].

```
reduce(Program,true).                       % halt
reduce(Program,(A,B)) :-                    % fork
        reduce(Program?,A?),
        reduce(Program?,B?).
reduce(Program,Goal) :-                     % reduce
        Goal \= true,Goal \= (A,B),
        clause(Goal?,Program?,Body) |
        reduce(Program?,Body?).

clause(Goal,[C|Cs],B) :-                    % a clause
        new_copy(C?,(H,G,B)),
        Goal = H,G | true.
clause(Goal,[C|Cs],B) :-                    % the rest
        clause(Goal,Cs?,B) | true.
```

As we consider a wide variety of logic-based languages, we see that the distinction between structural and non-structural terms is often not clear. Conjunctive goals traversed by the meta-interpreter are not as clearly structural as a binary tree. For a further example, in $\lambda$Prolog [12] we have the usual terms of Prolog (although with a strong type system and unusual applicative syntax) but in addition we have $\lambda$ abstraction over a special kind of variable. Traversing such terms introduces a complication when the $\lambda$ abstraction case is encountered in the inductive definition of a data structure. However, the idiom for traversing such structures is well-known to $\lambda$Prolog programmers.

As an example, we consider a skeleton where the data structure defined is the higher-order abstract syntax for $\lambda$ expressions, with $\lambda$Prolog being used in its natural application domain as a meta-language. For an overview, see [14]. Object language variables are represented by $\lambda$-bound variables in the meta language, and the `apply` and `lambda` functors of the meta language represent object language application and abstraction respectively. Traversing applications is trivial in the `traverse_absyn` skeleton, but to traverse an abstraction we need a $\Pi$-bound variable which replaces the $\lambda$-bound variable by the use of meta-language application in the term `(M X)`.

```
type exp.
type apply (exp -> exp -> exp).
type lambda ((exp -> exp) -> exp).

type variable (exp -> o).
type traverse_absyn (exp -> o).

traverse_absyn X :- variable X.
traverse_absyn (apply F A) :-
        traverse_absyn F,
        traverse_absyn A.
traverse_absyn (lambda M) :-
        pi X \ ( variable X => traverse_absyn (M X) ).
```

While this skeleton differs in detail from the structural skeletons above, the "spirit" of its traversal is the same as before.

To summarize this section, we note that a competent programmer in a logic language has a set of skeletons at her fingertips as realizations of basic programming idioms. We advocate that a primary step in "packaging" a logic language for wide use is to present a core set of useful and appropriately diverse skeletons as a starting point for programmers. Furthermore, it is desirable to develop a strategy for designing new skeletons and identifying important ones.

## 3 Techniques

Techniques are program transformations that enhance a skeleton by threading a web of computation over the structure that is being traversed. The *kind* of computation that is threaded, like the kind of skeleton, depends on the kind of computation supported by the language. For example, in Prolog it can be unification constraints or term construction, in a CLP language it can be a network of the appropriate kinds of constraints, and in a committed choice language one can use streams incorporating the appropriate scheduling primitives or a variable structure that needs to be instantiated for synchronization of processes.

Much as we advocate a strategy for developing core skeletons for a logic-based language, we seek a strategy for identifying the important techniques corresponding to each class of skeletons. We wish to identify:

1. *Significant classes of non-skeletal computation supported*
   That is, we need to realize the basic kind of computation supported by the language. This is both at the fundamental level (e.g.: unification, constraint solving, assignment) and at the idiomatic level (list building, chains of constraints, etc.).

2. *Important mappings between traversal path and computation patterns*
   Having identified classes corresponding to computational idioms, we see that for any given

skeleton and given computation, there may be more than one way to map between the data structure traversed and the computation chain. We seek isomorphisms between traversal paths defined above and chains of computation. For example:

- a list being built up in Prolog may correspond elementwise to a list being traversed by the skeleton, or to elements of the skeleton list that have a certain property;
- a chain of arithmetic constraints may map to a path from the root of a tree to a leaf, or to its inorder traversal;

3. *How non-skeletal computation can and should affect skeletal traversal*
   While in many cases the decision as to how much of a data structure should be traversed can be left to a skeleton, sometimes this choice is intrinsically associated with the computation being threaded rather than the skeleton itself. There are two natural ways for this to happen:

   (a) *Direction of traversal*
   Where the skeleton specifies search among various paths through a data structure, the technique may, through success or failure of its computations, eliminate some of the branches. A good example occurs in the modeling of electrical circuits using piecewise models, or in heuristic search.

   (b) *Early termination of traversal*
   Similarly, the computation imposed by a technique may determine that an appropriate traversal as specified by the skeleton is not possible, and will result in failure. A good example of this comes from using the higher-order term traversal skeleton above to develop a type checker. If the program represented is not well-typed, the type constraints will be unsatisfiable. Another example is imposing a depth bound on a computation.

We begin our examples of techniques again with Prolog. We present two techniques for producing the same effect, namely collecting information gained during traversal into a list. The techniques are illustrated with respect to the inorder traversal skeleton for a binary tree `tree_io`. The first, called the *build* technique [16], adds a single argument, and relates the recursive results to the desired result with a call to append.

```
tree_io(leaf, []).
tree_io(node(K,L,R), LL) :-
        tree_io(L, L1),
        tree_io(R, L2),
        append(L1, [K|L2], LL).
```

The second technique, *accumulate-build*, threads a list representing a partial inorder traversal through successive branches of the tree, adding elements as it proceeds through nodes. To implement the technique a pair of extra arguments, known as an accumulator pair, are needed.

```
tree_io(leaf, L, L).
tree_io(node(K,L,R), IL, OL) :-
        tree_io(L, IL, TL),
        append(TL, [K], TL1),
        tree_io(R, TL1, OL).
```

Of course, the above program can be made more efficient. By reversing the order of the subgoals of the recursive rule, the element at the node cab be added to the beginning of the list rather than at the end of the list, which obviates the need for the call to append. That requires an understanding of the technique and its relation to the skeleton. Another way of removing the explicit call to append, accomplishing a more efficient program, is to use the difference-list technique [18].

The difference-list technique is also useful for committed choice languages. A related technique is the short circuit technique which can be used to synchronize goals and detect termination. We demonstrate the technique on the `reduce` clauses of the meta-interpreter given earlier for Flat Concurrent Prolog. Again our account follows [15]. Many other techniques, including aborting computations and deadlock detection, are given in Safra and Shapiro's paper.

```
reduce(Program,true,Chain-Chain).                        % halt
reduce(Program,(A,B),Left-Right) :-                      % fork
        reduce(Program?,A?,Left-Middle),
        reduce(Program?,B?,Middle-Right).
reduce(Program,Goal,Left-Right) :-                       % reduce
        Goal \= true,Goal \= (A,B),
        clause(Goal?,Program?,Body) |
        reduce(Program?,Body?,Left-Right).
```

Next we consider a case of threading arithmetic constraints over a structural skeleton. The skeleton `circuit/1` is given in the previous section. By threading arithmetic constraints over the data structure we get a simple program for modeling resistive DC circuits. The threading technique is discussed in greater detail in [10].

```
circuit(res(R), V, I) :-
        V = I * R.
circuit(ser(C1, C2), V, I) :-
        V = V1 + V2, I = I1, I = I2,
        circuit(C1, V1, I1),
        circuit(C2, V2, I2).
circuit(par(C1, C2), V, I) :-
        V = V1, V = V2, I = I1 + I2,
        circuit(C1, V1, I1),
        circuit(C2, V2, I2).
```

Now we give two examples using the abstract syntax term traversal skeleton for $\lambda$Prolog. The first uses a constraint iteration technique for type checking (and type synthesis) for object language terms, and the second uses a build technique for checking linearity of an expression. The `has_tp` predicate extends the skeleton by threading type unification constraints over the term structure. Then the `linear` predicate uses essentially a build technique to keep track of occurrences of object level bound variables occurring in a term to ensure that the object term is strictly linear in the sense that each bound variable occurs exactly once in its scope.

```
type exp.
type tp.
type apply (exp -> exp -> exp).
type lambda ((exp -> exp) -> exp).
```

```
type arrow (tp -> tp).

type variable (exp -> o).

type hs_tp (exp -> tp -> o).
type var_tp (exp -> tp -> o).

has_tp X T :- var X T.
has_tp (apply F A) T :-
        has_tp F (arrow T1 T),
        has_tp A T.
has_tp (lambda M) (arrow T1 T2) :-
        pi X \ ( var_tp X T1 -> has_tp (M X) T2 ).

type linear (exp -> list -> list -> o).
type lintop (exp -> o).

lintop E :- linear E nil nil.

linear X I O :- var X, delete X I O.
linear (apply F A) I O :-
        linear F I M,
        linear A M O.
linear (lambda M) I O :-
        pi X \ ( variable X => linear (M X) (cons x I) O).
```

To summarize, we note that in order for a logic language to be usable, it is necessary to describe specific useful techniques for skeletons. Corresponding to the identification of skeleton strategy and specific skeletons, it is necessary to provide a strategy for identifying techniques for given skeletons. Again, we emphasize the importance of providing a core set of techniques to give programmers a starting point.

# 4 Consequences

Two areas of software engineering where structuring programs via skeletons and techniques is definitely useful are software analysis and software maintenance. By software analysis, we refer to issues such as (formal) correctness of programs and analysis of programs to allow better compiler optimizations, and the ability to exploit parallelism for example as in [1]. By software maintenance, we refer not only to low-level program changes, but also to broader 'software evolution', where some parts of a software systems are re-designed, but most are re-used. Skeletons and techniques constitute reusable components in a literal sense.

## 4.1 Analysis

This section presents two aspects of program analysis: program correctness and program optimization. We sketch an approach for generating correctness proofs for Prolog programs based on skeletons and techniques, and then discuss optimizations for CLP($\mathcal{R}$) programs that are guided by an understanding of specific techniques.

Our strategy for addressing correctness of Prolog programs is to begin with a skeleton whose behavior in traversing terms is well understood. A technique is applied. If the technique which is applied preserves the behavior of the skeleton, proving the correctness of the resulting program amounts to showing that the computations interwoven by the technique 'do the right thing.' This can be done by reproducing the (usually) inductive proof that establishes correctness of the skeleton or by trying to argue relative correctness by formally mapping computations from the final program back to the skeleton program. This topic is addressed in detail in [6].

For example, consider the following program which computes the number of nodes and the depth of a ground binary tree. A suitable program for finding the maximum of two integers, max, is assumed.

```
nodes_depth(leaf,0,0).
nodes_depth(node(K,L,R),N,D) :-
        nodes_depth(L,NL,DL),
        nodes_depth(R,NR,DR),
        N is NL+NR+1,
        max(DL,DR,DMax), D is DMax + 1.
```

Counting the number of nodes in the tree requires a complete traversal suggesting `tree_io` as our skeleton. Clearly `tree_io` traverses the binary tree. We can evaluate the program clause by clause and argue that the correct calculations are being performed, essentially relying on an inductive argument.

The other aspect of program analysis that can be aided significantly by the use of skeletons and techniques is that of efficient execution. A key component in optimizing the performance of virtually any kind of programming language is global flow and type analysis to determine how components of a program fit together. Such information is crucial for optimizing compilation of any language where "procedures" of fine granularity are written, including virtually all logic-based and functional languages. It is also crucial for detecting and exploiting parallelism safely and efficiently.

While it is unlikely to supplant the use of global program analysis, the use of skeletons and techniques for program development can aid such optimization of programs, since the choice of skeleton and technique explicitly provides non-local information about program structure. Each particular skeleton determines the kinds of search sequences that can result from a query, and each individual technique determines the ways in which computation is imposed on the search paths. Thus, the knowledge that a program component was constructed using a particular skeleton and a particular technique can be used to trigger the use of a particular optimization. A number of such optimizations have already been described for constraint logic programming.

In [9], it is shown how it is possible to avoid deciding large numbers of linear arithmetic inequalities in an important class of recursive procedures utilizing a constraint iteration technique that imposes an early termination check. For certain query patterns this termination check is inexpensive but useful, while for others it is expensive but in all but one iteration useless. The optimization can render usable programs that were previously considered totally impractical. In [11], it is shown that some linear constraint iteration techniques for some basic skeletons allow a compiler to replace most linear equation solving with ground computation, again resulting in major performance improvements. In both cases, the optimizations rely heavily on the way the rules of a program fit together to impose sequences of constraints throughout a computation. The idiom of skeletons and techniques helped to develop the optimizations, and is expected to help make them applicable in a globally optimizing compiler being developed for CLP($\mathcal{R}$).

## 4.2 Maintenance

We take a broad view of program maintenance, including a range of operations and activities, from low-level changes and bug fixing through adding new features to substantial revisions which happen through program evolution. While facilitating simple operations is clearly easier, we also intend looking at the larger issues in software maintenance.

Any naive support for skeletons and techniques will allow simple program maintenance. If the only change is to a single technique, all the programmer need do is create a new program embodying the changed technique and replay the sequence of operations and transformations which produced the complicated programs. Ideally, the modifications and replay are moderated through an easy-to-use user interface. Facilitating program modification by replaying operations that have not changed is certainly viable. A prototype system was demonstrated by Komorowski at a logic programming workshop in 1991. His system was based on a poorer version of techniques proposed by Lakhotia [7].

The broadest issue of program maintenance is the status of skeletons and techniques as reusable software components. Indeed in a literal sense, skeletons are reused. The same simple Prolog meta-interpreter has been the starting point for expert system shells and a program tracer. The grammar from Chapter 24 of [18] was the starting point of software testing programs.

It would be desirable to easily transfer techniques from one skeleton to another. For example, a prototype system for instrumenting data coverage testing for Pascal was developed at Case Western Reserve University. A novice Prolog programmer was assigned the task of adapting the program for instrumenting data coverage testing to work on C programs rather than Pascal. The programmer was successful largely because of the structuring of the program into skeleton, a parser for the language, and techniques, adding the data paths, etc. The skeleton changed from the Pascal version to the C version due to the different syntax of the language, but the techniques did not. This nontrivial programming task would almost certainly not have been accomplished without the structuring.

## 4.3 Programming Tools

An environment supporting Prolog program development using skeletons, techniques and composition has been designed, and a prototype implemented [19]. The environment allows the application of four techniques to skeletons - calculate, build, accumulate-calculate and accumulate-build. Seven skeletons are pre-defined with the environment, and the user can add her own. Limited error checking has been incorporated.

The prototype currently facilitates construction of programs concerned with recursive data structure traversal. Via a mouse-driven interface, a user can apply techniques to skeletons. The user is prompted to fill in values for arguments and supply extra goals - the interface stepping through the code to the appropriate places.

A consequence of using the tool is to enforce a programming style. All the programs constructed with the tool incorporate the techniques in the same way. We claim that an application built by a team of programmers using our tool could be said to be using a standardized programming practice in a way that may be acceptable for process standards.

There is considerable scope and potential for both enhancing the tool and widening its applicability. It is currently tailored to Prolog but can be readily adapted for CLP($\mathcal{R}$). More techniques can be embedded, and support for program maintenance could be made explicit. This is the subject of ongoing research.

Another tool has been built to apply techniques to programs - via partial evaluation of an enhanced meta-interpreter rather than direct program construction. This elegant way to apply techniques is as follows. It is especially useful for large programs.

- Write a simple interpreter;
- Enhance the interpreter with the desired technique;
- Partially evaluate the enhanced interpreter with respect to the program.

The meta-interpreter for Flat Concurrent Prolog given in Section 2 is amenable to partial evaluation, as is advocated in [15]. Partial evaluation as a means of easy incorporation of techniques is also applicable to rule-based interpreters as presented in [17].

# 5 Discussion and Conclusions

We begin with an observation about the maturation of programming languages. When a radically different programming language is first introduced, there is an exciting period of experimentation. Some people find clever ways of programming interesting examples. It takes time to know whether the cleverness is a generally useful programming tactic or just a diverting curiosity. Experience has shown that cleverness is less important than clarity, and a language will only be widely adopted if good programming style can be taught and used. We believe that identifying skeletons and techniques is a sign of maturity of a logic-based language. Conversely, maturity will be speeded up by actively seeking to determine skeletons and techniques.

Explaining nontrivial programs in terms of skeletons and techniques has proven very useful in the classroom. In logic programming classes at our universities we have discovered that students easily comprehend more complicated examples than students had in the same course in prior years. Students claim that their programming ability is improved, though this is hard to document rigorously.

Reactions to this structured approach to programming in logic based languages in seminars has been universally favorable, though the reaction is often that skeletons and techniques are just common sense and reaffirm the validity of the listeners programming style. We take this positively as standard structured programming for imperative languages is essentially only common sense programming. Nonetheless logic languages have lacked an accepted methodology that can be identified which we believe skeletons and techniques can provide.

Skeletons and techniques can serve as useful documentation of a programming project, and we have some experience that thinking in terms of them leads to insight. The two most notable examples are experience of Arun Lakhotia building a graphical debugger for Prolog and the two-layer interpreter for success and failure developed by Ümit Yalçinalp and applied for explanation [22] and uncertainty reasoning [23], which appears in the second edition of "The Art of Prolog" [18].

## Acknowledgements

The authors wish to thank Frank Pfenning for corrections to the λProlog examples and Ashish Jain for useful discussions about skeletons and techniques. This work was partially supported by NSF grant CCR-9303484.

## References


[1] A. Bansal and L. Sterling. Classifying Generate-and-Test Logic Programs. *International Journal of Parallel Processing*, 8:401-446, 1989.

[2] A. Bansal and L. Sterling. An Abstract Interpretation Scheme for Identifying Inherent Parallelism in Logic Programs. *New Generation Computing*, 7: 273-324, 1990.

[3] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inferencing. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pp. 669-683, MIT Press, Cambridge, Mass., 1988.

[4] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1990.

[5] Marc Kirschenbaum, Leon Sterling, and Ashish Jain. Relating logic programs via program maps. *Annals of Mathematics and Artificial Intelligence*, 1993.

[6] M. Kirschenbaum, A. Jain and L. Sterling. Relative Correctness of Prolog Programs. *Proc. Third Int'l. Symp. on Artificial Intelligence and Mathematics*, Florida, Jan. 1994.

[7] A. Lakhotia. Incorporating 'Programming Techniques' into Prolog Programs. *Proc. North American Conference on Logic Programming*, (eds. Lusk and Overbeek), pp. 426-440, MIT Press, 1989.

[8] C.S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming*, 2:43-65, 1985.

[9] Spiro Michaylov. Repeated redundant inequalities in constraint logic programming. Technical Report forthcoming, Department of Computer and Information Science, The Ohio State University, 1994.

[10] Spiro Michaylov. Skeletons and techniques for the systematic development of constraint logic programs. Technical Report forthcoming, Department of Computer and Information Science, The Ohio State University, 1994.

[11] Spiro Michaylov and Bill Pippin. Optimizing compilation of linear constraints in a class of constraint logic programs. Technical Report forthcoming, Department of Computer and Information Science, The Ohio State University, 1994.

[12] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 448–462, London, July 1986. Springer Verlag LNCS 225.

[13] F.C.N. Pereira and S.M. Shieber. Prolog and Natural language Analysis. CSLI Lecture Notes No. 10, CSLI, 1987.

[14] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208. ACM Press, June 1988.

[15] S. Safra and E.Y. Shapiro. Meta Interpreters for Real. in *Concurrent Prolog, Collected Papers* (E. Shapiro, ed.), pp. 166-179, MIT Press, 1987.

[16] Leon Sterling and Marc Kirschenbaum. Applying techniques to skeletons. In J.-M. Jacquet, editor, *Constructing Logic Programs*, pages 127–140. MIT Press, 1993.

[17] Leon Sterling and Randall Beer. Meta-interpreters for expert systems. *Journal of Logic Programming*, 6:163–178, 1989.

[18] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, $2^{nd}$ edition, 1994.

[19] L. Sterling and Chok Sitt Sen. A Tool to Support Stepwise Enhancement in Prolog. *Proc. Workshop on Logic programming Environments*, Vancouver, October, 1993.

[20] P. Van Roy and A.M. Despain. High-Performance Logic programming with the Aquarius Prolog Compiler. *IEEE Computer*, 25:54-68, Jan. 1992.

[21] R. Warren, M. Hermenegildo, and S.K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pp. 684-699, MIT Press, Cambridge, Mass., 1988.

[22] L.Ü. Yalçinalp and L.S. Sterling. Building Embedded Languages and Expert System Shells in Prolog. In *Proc. Tools for AI*, pp. 147-158, Washington, DC, November, 1990.

[23] L.Ü. Yalçinalp and L.S. Sterling. Uncertainty Reasoning in Prolog with Layered Meta-Interpreters. In *Proc. 7th Conference on Artificial Intelligence Applications*, IEEE Computer Society Press, pp. 398-402, February, 1991.