Skeletons and Techniques for the Systematic Development of Constraint Logic Programs*

Technical Report OSU-CISRC-6/94-TR30

Spiro Michaylov

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.

June 30, 1994

Abstract

We study the systematic development of Constraint Logic Programs from the viewpoint of Skeletons and Techniques as described by Kirschenbaum and Sterling. We describe a number of fundamental skeleton classes for CLP, and generalize the notion of skeletons to deal with non-structural recursion. Then we describe a range of useful techniques for extending these skeletons. Furthermore, we introduce important classes of techniques that alter the control flow of skeletons in certain well-defined and desirable ways.

This work represents a step towards understanding how to develop complex CLP programs easily, and is expected to contribute to the adoption of CLP for applications projects. It may also lead to the development of semi-automated program development tools. Finally, it helps to justify a substantial body of present work on CLP compiler optimizations that depends on the procedure level structure of programs.

*A preliminary version of this paper appears in the Proceedings of the International Conference on Tools with Artificial Intelligence, New Orleans, November, 1994.

1 Introduction

The utility of Constraint Logic Programming [2] is widely recognized. Many different languages have been defined and implemented, and a wide range of applications have been investigated using those languages. However, the utility of a programming paradigm or language depends greatly on how well its use is understood—that is, whether a useful programming methodology exists for it. The issue is a difficult one, as demonstrated by the continued controversy about the construction of imperative programs. Nevertheless, even partial progress can be tremendously useful. For example, the adoption of Prolog has been aided significantly by the availability of the texts by Sterling and Shapiro [16] and O'Keefe [14], and the collection edited by Sterling [15].

In [17], Sterling and Kirschenbaum describe how to develop Prolog programs by first developing simple programs that traverse data structures in a certain way (skeletons), and then extending them to perform useful computation by the use of program transformations (techniques). For example, the list traversal skeleton of Figure 1 can be extended to the programs in Figure 2 for determining the length and element sum of a list, respectively, by the use of the `Calculate` technique. They develop a catalog of useful skeletons and techniques, and in [7], Kirschenbaum, Sterling and Jain show that the structure of computations is preserved by the application of a class of techniques in an important way.

```
traverse([]).
traverse([X|Xs]) :-
        traverse(Xs).
```

Figure 1: Prolog list traversal skeleton

```
length([], Len) :-                  sum([], Sum) :-
        Len is 0.                           Sum is 0.
length([X|Xs], Len) :-              sum([X|Xs], Sum) :-
        length(Xs, Len1),                   sum(Xs, Sum1),
        Len is Len1 + 1.                    Sum is Sum1 + X.
```

Figure 2: Prolog list length and sum

It has been clear for some time that Prolog programming methodology is quite applicable in the context of CLP, but it has also been quite clear that CLP offers a wider range of programming techniques, as discussed in [3] and elsewhere. Extended discussion of CLP applications can be found in [1, 5, 8, 9, 18] and elsewhere. Our objective in this paper is to develop an understanding of how skeletons and techniques can be applied in general to constraint logic programming languages. We generalize the notion of skeletons to deal with non-structural recursion, and describe a hierarchy of fundamental skeleton classes for CLP, examining a series of corresponding examples. Many of the resulting skeletons turn out to have applications for Prolog programming as well, but the point is that they are not crucial for basic Prolog programming, whereas they are for CLP. We go on to examine a series of techniques for CLP, again considering examples. Furthermore, we describe situations where techniques alter the control flow of skeletons in certain well-defined and desirable ways. Again, special cases of these techniques correspond to Prolog techniques. However, for CLP programming they must be thought of from a different viewpoint—as constraints threaded over a skeleton of computation, rather than as a chain of evaluations and assignments.

The use of skeletons for CLP was introduced by Kirschenbaum, Michaylov and Sterling [6], with a number of simple examples. That paper explored the *common* aspects of the use of skeletons and techniques for logic-based languages. In contrast, this paper explores the *specific* skeletons and techniques needed to make CLP programming accessible to relatively naive programmers. While it is frequently observed that many techniques can be described in terms of the higher-order constructs used in functional programming, this observation misses the point of skeletons and techniques. The primary contribution of this paper for CLP programming methodology is the identification of a core set of specific skeletons for a wide range of applications, and a core set of specific techniques for each skeleton. For simplicity of exposition, the examples in this paper all deal with CLP languages whose domain incorporates both some form of arithmetic and uninterpreted function symbols, but most of the discussion carries over to a wide range of CLP languages based on different domains.

In addition to providing a more systematic programming methodology for CLP, and thus contributing to the applicability of the languages, it is hoped that this work can also make a major contribution to the area of global program optimization, by demonstrating that global optimization methods as described in [4, 11, 10, 12, 13] correspond to fundamental skeletons and techniques for CLP.

The remainder of this paper is organized as follows. In Section 2 we explore the types of skeletons that are useful for CLP programming, including non-structural skeletons. Then in Section 3 we describe a range of techniques based on iteration of constraints that can be used to extend the skeletons. Finally, in Section 4 we consider classes of techniques that modify the control behavior of skeletons.

2 Skeletons in CLP

Most of the traditional Prolog skeletons are applicable for CLP languages, as the traversal of data structures continues to be of central importance. However, CLP languages allow other kinds of skeletons as well. We consider the traditional and non-traditional kinds of skeletons in order.

2.1 Structural Skeletons

A form common to many CLP programs is the recursive traversal of the description of some system being modeled, while establishing a network of potentially non-trivial constraints. As a convenient example of this paradigm we use the modeling of resistive electrical circuits, described by a term using the constructors `res/1` for an individual resistor of given value, and `par/2` and `ser/2` for the parallel and series composition of two circuits respectively. A

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

Figure 3: Traversal skeleton for resistive circuit description

```
slide([_,_]).
slide([E1, E2, E3 | T]) :-
        slide([E2, E3 | T]).
```

Figure 4: Three-at-a-time sliding window skeleton for a list

```
slide([_]).
slide([_,_]).
slide([E1, E2 | T]) :- slide([E2 | T]).
slide([E1, E2, E3 | T]) :- slide([E3 | T]).
slide([E1, E2, E3 | T]) :- slide([E2, E3 | T]).
```

Figure 5: Flexible sliding window skeleton for a list

```
adj_tree_trav(leaf).
adj_tree_trav(inner(leaf,K,leaf)).
adj_tree_trav(inner(inner(LL,LK,LR),K,leaf)) :-
        adj_tree_trav(inner(LL,LK,LR)).
adj_tree_trav(inner(leaf,K,inner(RL,RK,RR))) :-
        adj_tree_trav(inner(RL,RK,RR)).
adj_tree_trav(inner(inner(LL,LK,LR),K,inner(RL,RK,RR))) :-
        adj_tree_trav(inner(LL,LK,LR)),
        adj_tree_trav(inner(RL,RK,RR)).
```

Figure 6: Binary tree traversal viewing node and two children

```
adj_tree_search(leaf).
adj_tree_search(inner(leaf,K,leaf)).
adj_tree_search(inner(inner(LL,LK,LR),K,R)) :-
        adj_tree_search(inner(LL,LK,LR)).
adj_tree_search(inner(L,K,inner(RL,RK,RR))) :-
        adj_tree_search(inner(RL,RK,RR)).
```

Figure 7: Binary tree search viewing node and child

full traversal skeleton is shown in Figure 3. Skeletons like this one have of course already arisen in the context of Prolog. We include this here because of the techniques that we will discuss in the next section.

Certain kinds of CLP programs are based on skeletons that may well have been written for Prolog programs, but would probably not have been very useful. One such class is that of *sliding window* skeletons, epitomized by the list traversal skeleton in Figure 4. These skeletons allow us to establish relationships between adjacent elements of a data structure. They will be useful when we wish to assert relationships between adjacent elements in a data structure. It is easy to see how similar skeletons can be devised with windows of different size: the example in Figure 5 allows us to both skip certain elements, and to vary the window between two and three elements. This is one of many examples where a simpler skeleton would have provided the same power, but would have resulted in a less efficient program. In fact, in principle the sliding window skeletons are superfluous. However, we also wish to provide programmers with skeletons for reasonably efficient programs. Also, arguably, such skeletons result in more easily understandable programs.

Sliding window skeletons can also be defined for more complicated data structures like trees. In particular, we could impose the "binary search tree" property on the nodes of a tree by extending an appropriate version of such a skeleton. Figure 6 shows how to traverse a binary tree in this way, and Figure 7 shows how to search it.

2.2 Non-structural Skeletons

While many CLP programs can be seen to be based on such "structural" skeletons as seen above, it should not be surprising that many are not. A "non-structural" subdomain of a CLP language can just as well be used to develop a recursion. For example, consider the skeleton in Figure 8. This is a common way to recurse through, say, a time interval. This particular skeleton is, of course, isomorphic to a structural one, but in general it is preferable to be able to reason about programs based on such a skeleton directly, without considering a mapping to a structural equivalent. As a more complicated example of a non-structural skeleton, consider Figure 9.

The two examples shown may be thought of as traversal skeletons. It is also possible, and sometimes useful, to define searching skeletons that are non-structural. These skeletons do not traverse the entire conceptual space, but nondeterministically choose parts of that space to explore. This concept is illustrated using the two "divide and conquer" skeletons in Figure 10 and 11. The first of these subdivides an arithmetic interval recursively, exploring both subdivisions. The second similarly subdivides the interval recursively, but nondeterministically chooses a sub-interval to explore.

3 Techniques in CLP

Our primary motivation in developing new classes of skeletons for CLP programs was to enable a network of constraints to be threaded over structures more easily, concisely and efficiently. We now explore the various kinds of techniques that can be used to thread constraints by extending both classical and new skeletons. When we study these techniques

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

Figure 8: Skeleton for "time" stepping

```
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).
```

Figure 9: More complicated non-structural skeleton

```
split(L, H, D) :- H - L < D.
split(L, H, D) :-
        Mid = (L + H) / 2,
        split(L, Mid, D),
        split(Mid, H, D).
```

Figure 10: Non-structural "divide and conquer" traversal skeleton

```
split(L, H, D) :- H - L < D.
split(L, H, D) :-
        Mid = (L + H) / 2,
        split(L, Mid, D).
split(L, H, D) :-
        Mid = (L + H) / 2,
        split(Mid, H, D).
```

Figure 11: Non-structural "divide and conquer" search skeleton

it becomes obvious that sometimes the success or failure of a computation should rely not on the skeleton, but on the network of constraints threaded over it. This means that techniques will effect the control flow of a skeleton in various ways. However, for skeletons and techniques to provide a useful program development method, we require that we understand how various techniques affect the flow of control. Different techniques have different properties in this respect, so we will defer the issue until the next section.

The following taxonomy of techniques serves a number of purposes. It can help make programmers more conscious of the choices they are making in designing programs, it helps us study the computational effects or control properties of various techniques, and by placing programs into syntactically-defined classes we make it easier to define high-level program optimizations and proof rules.

3.1 The Class of Iteration Techniques

To illustrate the fundamental class of techniques needed for CLP programming, we will consider first the structural traversal skeletons, as exemplified by the circuit description traversal skeleton in Figure 3. The `Iterate` techniques associate with each function symbol in a term a (possibly disjunctive) constraint, and define a way for the constraints to be related by variables reflecting the term structure.

The `Map` techniques

We first illustrate this by the somewhat degenerate example of the program `check_circuit`. This associates an inequality constraint with each resistor node, and the constraint *true* with each of the other nodes, but does not relate the constraints to each other at all. Thus, the technique does not add any variables to the relation. We call this the `Map` subclass of the `Iterate` techniques, borrowing from the name of a related class of higher-order functions in functional programming. We note that these techniques could also apply constraints to internal nodes of a tree if they contained data.

```
check_circuit(res(R)) :-
        R > 100.
check_circuit(ser(C1, C2)) :-
        check_circuit(C1), check_circuit(C2).
check_circuit(par(C1, C2)) :-
        check_circuit(C1), check_circuit(C2).
```

In general, these techniques are most useful for checking or imposing local constraints on the contents of a structure.

The `Thread` techniques

Next we consider techniques that do connect the constraints by adding parameters to the skeleton relation, but in a very restricted way. At each internal node of a term structure, these techniques only connect the subtrees pairwise, through a variable sharing (or constraining) pattern often referred to as multilinearity in the deductive databases literature, and closely related to the concept of a "bucket brigade" in the literature on attribute grammars.

The parameters added by this technique are partitioned into two classes – we call the head instances $\tilde{X}$ and $\tilde{Y}$, and the instances in the ith recursive call (out of n recursive calls) $\tilde{X}_i$ and $\tilde{Y}_i$ respectively. For a given recursive rule, we distinguish two recursive calls a and b such that either

- $n = a = b = 1$, or
- $n > 1$, $1 \leq a \leq n$, $1 \leq b \leq n$ and $a \neq b$.

The constraints in that rule are then organized as follows:

1. A set of constraints relates $\tilde{X}$ and $\tilde{X}_a$.
2. A set of constraints relates $\tilde{Y}$ and $\tilde{Y}_b$.
3. If $n \neq 1$, let the sequence $\langle c_1, \cdots, c_n \rangle$ be a permutation of the sequence $\langle 1, \cdots, n \rangle$ such that $c_1 = a$ and $c_n = b$. Then for each $i : 1 \leq i < n$ there is a set of constraints relating $\tilde{Y}_{c_i}$ and $\tilde{X}_{c_{i+1}}$.
4. No other constraints relate these parameters.

Ideally, all parameters should be related in this way. As a simple example with only one parameter in each class, consider the rule:

```
three_way(tree(T1, T1, T2), X, Y) :-
        X = X1, Y = 2*Y3, X2 - Y1 = 4, X3 - 2*Y2 = 1,
        three_way(T1, X1, Y1),
        three_way(T2, X2, Y2),
        three_way(T3, X3, Y3).
```

We illustrate with the circuit traversal skeleton for simplicity and uniformity, although such techniques are mostly useful for threading non-arithmetic constraints, as they may result in more efficient programs than other techniques. They correspond closely to the `Accumulate` techniques for Prolog lists. The example here simply counts the number of resistors and parallel and series compositions in a circuit, by adding one to the count of those preceding it in the traversal sequence. Note that we do not have single parameters for each count, but rather one for the total before traversal and one for the total after. Of course, a goal like `?- circuit_count(C, 0, NR, 0, NP, 0, NS)` would produce the more conventional result.

```
circuit_count(res(R), RI, RO, P, P, S, S) :-
        RO - RI = 1.
circuit_count(ser(C1, C2), RI, RO, PI, PO, SI, SO) :-
        RO - RI = (RO1 - RI1) + (RO2 - RI2),
        PO - PI = (PO1 - PI1) + (PO2 - PI2),
```

```
        SO - SI = (SO1 - SI1) + (SO2 - SI2) + 1,
        circuit_count(C1, RI1, RO1, PI1, PO1, SI1, SO1),
        circuit_count(C2, RI2, RO2, PI2, PO2, SI1, SO2).
circuit_count(par(C1, C2), RI, RO, PI, PO, SI, SO) :-
        RO - RI = (RO1 - RI1) + (RO2 - RI2),
        PO - PI = (PO1 - PI1) + (PO2 - PI2) + 1,
        SO - SI = (SO1 - SI1) + (SO2 - SI2),
        circuit_count(C1, RI1, RO1, PI1, PO1, SI1, SO1),
        circuit_count(C2, RI2, RO2, PI2, PO2, SI2, SO2).
```

Notice that while the constraints seem rather complicated, they are connected together in a rather simple and uniform pattern.

The `Copy-Combine` techniques

The next more complicated way to connect constraints is to add two classes of parameters to the skeleton relation:

1. the `Copy` parameters, which are used for sharing information among the subtrees of a node, and
2. the `Combine` parameters, which are used for bringing together and summarizing, or combining, information from the subtrees of a node.

Each recursive rule from the skeleton will then obtain constraints as follows:

1. for each `Copy` parameter, for each recursive call, one constraint between the head instance of the parameter and the corresponding instance in the recursive call, and
2. for each `Combine` parameter, one constraint among the head instance and the instances in all of the recursive calls.

The base case rules will have constraints potentially relating all of the `Copy` and `Combine` parameters.

As an example we show a program `circuit_cost` that models the relationship between the costs of resistors and capacitors in a circuit, and the total cost of the circuit. This example has two `Copy` parameters (`RC` and `CC`) and one `Combine` parameter (`TC`). Note that the equality constraints on `Copy` parameters are implicit.

```
circuit_cost(res(R), RC, CC, TC) :-
        TC = RC.
circuit_cost(cap(R), RC, CC, TC) :-
        TC = CC.
circuit_cost(ser(C1, C2), RC, CC, TC) :-
        TC = TC1 + TC2,
        circuit_cost(C1, RC, CC, TC1),
        circuit_cost(C2, RC, CC, TC2).
```

```
circuit_cost(par(C1, C2), RC, CC, TC) :-
        TC = TC1 + TC2,
        circuit_cost(C1, RC, CC, TC1),
        circuit_cost(C2, RC, CC, TC2).
```

A number of interesting variations on these techniques are possible:

- There may be either no `Copy` parameters or no `Combine` parameters, resulting in either a `Combine` or `Copy` technique, respectively.
- The constraints on either `Copy` or `Combine` parameters may omit some of the parameter instances.
- Individual parameters may be replaced by a group of parameters, so that a number of constraints may relate the same set of parameters in a rule.
- The `Copy` parameters may be involved in the constraints on the `Combine` parameters. For example, we may have added a parameter for the cost of solder points, and added the cost of one solder point to each series circuit composition and two solder points for each parallel circuit composition.
- In different recursive rules, the `Copy` and `Combine` parameters may swap roles.

This last may seem somewhat bizarre initially, but in fact it has useful applications. Consider the following `analyse_circuit` program. The first two rules follow the above pattern, with one `Copy` parameter (`I`) and one `Combine` parameter (`V`). However, this is reversed in the third rule. It can be imagined that arbitrary repartitioning of parameters among recursive rules could also take place. However, we have not observed this in any useful programs, and it is quite difficult to develop intuition for.

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

3.2 Techniques for Sliding Window Skeletons

The `Iterate` techniques on basic structural skeletons only allow different parts of a structure to be related via parameters added to the skeleton relation. However, in some kinds of computations we wish to relate neighboring elements in a structure. This can always be done solely through the use of additional parameters, but the resulting programs will frequently be obscure and inefficient.

As mentioned earlier, sliding window skeletons allow us to assert relationships between adjacent components of a data structure. The `heat` program below is obtained by augmenting the sliding window skeleton of Figure 4 using a variant of a `Map` technique, which we may call a `Slide-Map` technique. Each adjacent triple of elements in a list is required to have the property that the middle element is the average of its two neighbors.

```
heat([_,_]).
heat([E1, E2, E3 | T]) :-
        E1 + E3 = 2 * E2,
        heat([E2, E3 | T]).
```

As should be expected for a techniques derived from `Map`, this program does not have any additional parameters: the constraints being threaded across the structure are still somewhat local in nature. Of course, `Slide` variants of both the `Thread` and `Copy-Combine` techniques can be defined. Whether they are useful in practice, however, has yet to be determined. This is because overlapping local constraints achieve a form of global constraining. For example, the expected use of the `heat` program is with a list where the first and last elements (boundary values) are ground and the remaining elements are free. The established network of constraints will result in all elements of the list being grounded by the end of executing such a call. For a somewhat contrived example of a degenerate `Slide-Copy-Combine` technique, consider the `list_delta` program below, which ensures that the difference between successive pairs of elements in a list is non-decreasing. A 2-variable version of the `slide` skeleton has been extended.

```
list_delta([_], Delta).
list_delta([E1, E2 | T], Delta) :-
        Delta = abs(E2 - E1),
        Delta1 >= Delta,
        list_delta([E2 | T], Delta1).
```

We say that this uses a degenerate form of `Slide-Copy-Combine` because there is no meaningful distinction between `Copy` and `Combine` parameters when there is only one recursive call.

3.3 Techniques for Non-Structural Skeletons

In discussing non-structural skeletons, we saw that the same sorts of recursion patterns that could be obtained from traversing or searching structural skeletons could be obtained for the non-structural ones. As an example, let us consider the well-known `mortgage` program, obtained from the non-structural skeleton of Figure 8. This may actually be seen as a case of applying both a `Thread` technique and a `Copy` technique. The threading is through the `P` and `B` parameters, while the `I` and `R` parameters are trivially copied for use in the threading constraint. It is possible to obtain more complicated examples of applying the various

Iterate techniques to a non-structural skeleton by considering a variant of the mortgage program where, say, the interest rate changes with time.

```
mortgage(T, P, I, R, B) :-
        T = 1,
        B = P * I - R.
mortgage(T, P, I, R, B) :-
        T > 1,
        T1 = T - 1,
        P1 = P * I - R,
        mortgage(T1, P1, I, R, B).
```

Thus, in most respects, techniques for non-structural skeletons are just like those for structural ones. However, in general it is more likely that the skeleton parameters will be related to those introduced by techniques through constraints when non-structural skeletons are used.

4 Techniques that Modify Control Behavior

Skeletons are intended broadly to control a computation, while techniques are intended to add work to the computation, without fundamentally changing its structure. In particular, skeleton computations that would have failed should lead to failing computations in the extended program. Furthermore, successful computations should essentially be of the same length. However, by examining a number of existing, well-written CLP programs we can see that it can be useful to allow techniques to modify this traversal in certain well-defined ways. In fact, many of the techniques we discussed in Section 3 could have modified control behavior. Here we justify the use of techniques that modify the control behavior of a skeleton in certain ways, and identify the kinds of control modification that are needed.

4.1 Classes of Control Modification in Techniques

The point of the `Iterate` class of techniques is to connect a network of constraints in a way that reflects the form and content of a syntactic term (or the conceptual structure of a non-syntactic entity). Now, for some terms, such a network of constraints will turn out to be unsatisfiable, and it is natural for this to happen. In fact, the purpose is often to separate the terms for which the network is satisfiable from those for which it is not. We see that this property can cause a technique to modify the control behavior of a skeleton in a number of different ways.

Failure by Local Property

For the `Map` techniques, and others, it may well be the case that an individual constraint is unsatisfiable. This usually means that the data contained at some point in a term is locally undesirable in some way. In fact, the `check_circuit` program in Section 3 behaves in this way: if the term represents a circuit with a resistor of value ≤ 100, the entire computation fails.

Failure by Accumulated Property

Many `Thread` techniques cause a computation to fail in a different way. Instead of checking a local property of the data in a skeleton, these techniques check a property of one of the parameters added. Because of the nature of `Thread` techniques, this means that for queries that ground the corresponding parameter, the constraints introduced check a property of a value "computed so far". Typically, with arithmetic constraints, the programmer's intent is to cause "early bailout" if the value of some running total has left the allowable range.

As an example, consider the following modified recursive rule for the `mortgage` program. It alters the behavior of the `step` skeleton by "bailing out" if the principal parameter `P` falls below zero. Again, this is an innocuous change, since successful computations are not affected.

```
mortgage(T, P, I, R, B) :-
        T > 1,
        P >= 0,
        T1 = T - 1,
        P1 = P * I - R,
        mortgage(T1, P1, I, R, B).
```

Conversely, it is interesting to observe that, in practice, `Thread` techniques that do not include such "bailout" constraints tend not to cause failure in the middle of a computation. They usually allow enough degrees of freedom in constraints at each step of the computation that only when the goal arguments corresponding to the class 1 and class 2 parameters of the program are finally linked by the program there may be an inconsistency. Next we see a class of techniques for which this is not usually the case.

Failure by Global Property

Programs resulting from the application of non-trivial `Copy-Combine` techniques often have the most subtle control properties. Such programs tend to build up rather complicated and tightly connected networks of constraints, which for some skeleton arguments will simply turn out to be unsatisfiable. Furthermore, in many CLP languages, the domain of computation will make it impossible to ascribe the failure to any particular constraint, and hence to any particular part of a structural argument to a skeleton.

A simple example of such global failure comes in the `analyse_circuit` program, which may turn out to fail for certain unexpected queries if a negative resistance value is included in the skeleton argument (for example, not every voltage will be possible even if current is left unspecified). Furthermore, it will not necessarily fail exactly when that resistance value is dealt with. Now, such a circuit is pathological. However, there are numerous examples of CLP programs that fail in such a way for certain inputs that were written precisely to *check* properties of that input. For example, in [6], a λProlog skeleton for traversing abstract syntax tree representations is extended to produce a type checker. The constraints added by the `Combine` technique, on variables added by that technique, cause failure when the abstract syntax representation of a program is not well-typed. That example demonstrates that such control alteration by techniques is not merely tolerable but in fact essential.

Disjunction and Failure by Missing Case

We already stated that techniques of the `Iterate` class may associate disjunctions of constraints with skeleton rules. In fact, such *disjunctive techniques* produce one variant of the skeleton rule for each disjunct. Operationally, this changes the structure of computations in a number of ways. Of course, some disjunct may lead to failure in any of the ways described above. But then the rule variants produced by the other disjuncts will kick in, perhaps leading to successful computation, perhaps not.

As an example, consider adding the fact `traverse_circuit(diode)` to the circuit traversal skeleton of Figure 1 so that diodes may be modeled as well. We can develop a piecewise linear approximation to diode behavior by associating a disjunction of three conjunctive constraints with a diode instance. This results in three new rules in the extended program as follows.

```
analyse_circuit(diode, V, I) :-
        V < -100, I = (V+100)*10-0.1.
analyse_circuit(diode, V, I) :-
        V >= -100, V < 0.6, I = 0.001*V.
analyse_circuit(diode, V, I) :-
        V > 0.6, I = (V-0.6)*100+0.0006.
```

The extended program will use deep backtracking to find a combination of diode states that is consistent with the rest of the circuit. That is, global failure will cause alternatives to be explored. Because of the nature of the diode model, for any reasonable circuit a consistent state will be found.

Clearly disjunctive techniques will result in broadening the search tree associated with a skeleton and a given input. However, the details of the resulting behavior depend somewhat on the nature of the disjunction:

1. *Overlap*
   If the disjuncts are not mutually exclusive, they may actually lead to an increase in the number of successful or unsuccessful computations.

2. *Partial Cover*
   If the disjuncts do not cover all of the possibilities for the relevant variables, it is possible that the number of successful derivations will actually increase.

3. *Partition*
   If the disjuncts exactly partition the possibilities for the relevant variables, at least one of them will be consistent with the other constraints. Hence, no successful derivations are lost, and some may be added if the query provides enough freedom, but unsuccessful derivations may now have more possibilities to consider before failing.

Finally, we observe that depending on the query used, the backtracking caused by disjunctive constraints may be shallow (essentially local or cumulative failure) or deep (essentially global failure).

4.2 Simple Conditions for Determining Control Effects

We have considered classes of techniques that affect the operational behavior of skeletons in desirable ways. It is also valuable to consider how easily the control effects of these techniques can be understood by the programmer applying them.

Here we briefly list some simple tests that a programmer may consider to check that a technique either does not alter the control behavior or does so in a well-understood way. A full theoretical consideration of the control properties of certain classes of techniques is probably worth-while, but beyond the scope of this paper.

Free Query Behavior

Many techniques, including most of the `Thread` variety, will result in programs that will have unaltered control behavior if only the skeleton variables are instantiated, usually building up a network of constraints. This means that the constraints generated are internally consistent no matter what skeleton arguments are used.

Partially Free Query Behavior

In some cases, it is possible to do better—that is, maintain control properties for a wider range of queries. In particular, there may be one or more subsets of parameters added by the technique that can safely be instantiated without changing the control behavior of the skeleton. A large proportion of `Thread` techniques have this property. For example, if we consider a simple list traversal skeleton, the following program uses just a simple `Thread` technique to count the elements.

```
count([], M, N) :- M = N.
count([H | T], M, N) :-
        M1 = M + 1,
        count([T], M1, N).
```

In this program, the control behavior of the skeleton is preserved if only the query arguments corresponding to the `M` or `N` parameters are instantiated or pre-constrained, but many queries that constrain both will lead to failure.

Disjunctive Partition

As we have already discussed, when a technique adds a disjunction of constraints to a skeleton rule, and the disjuncts form a partition of the possible values of the relevant variables, successful computations are never lost, and some may be added depending on the amount of freedom allowed by the query and rest of the program. In certain cases, a disjunction may partition just some of the variables involved, and even this property can be useful for considering behavior for certain calling patterns.

Conclusions and Future Work

We have shown how the programming methodology of using skeletons and techniques can be applied to developing CLP programs systematically. Furthermore, we have described how the CLP paradigm both requires a generalized view of this methodology and provides opportunities for its more systematic use. In particular:

- The kinds of structural skeletons typical in Prolog programming are still of great value in CLP.
- Other kinds of structural skeletons, like the sliding window skeletons, become of interest in the context of constraints. They may also be useful for Prolog, although they were not motivated by Prolog.
- Skeletons relying on built-in non-structural domains such as arithmetic are of tremendous importance. Some of these can be quite complicated.
- The fact that there is a close correspondence between the use of constructors in a skeleton and the building of a network of constraints leads to a range of CLP techniques: the `Iterate` techniques. These can be further subdivided according to the way the constraints are related to each other through program parameters.
- In CLP, it can be particularly important for the new parameters and constraints added by a technique to affect flow of control. However, control tends to be affected in certain easily understandable ways.

In the future, we hope to investigate the construction of larger and more diverse CLP programs using this methodology, and explore the development of software construction and verification tools, as well as compiler optimizations, based upon it. We also hope to undertake a more formal study of both the construction of skeletons and techniques from more fundamental components, and the characterization of techniques that modify control behavior in various restricted ways.

Acknowledgments

The work described here, and its presentation, have both been heavily influenced by valuable discussions with Marc Kirschenbaum and Leon Sterling. Helpful comments were also provided by Nathan Loofbourrow, Bill Pippin and a number of anonymous referees.

References

[1] Nevin Heintze, Spiro Michaylov, and Peter Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. *Journal of Automated Reasoning*, 9:231–260, October 1992.

[2] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany*, pages 111–119. ACM, January 1987.

[3] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, July 1992.

[4] Niels Jørgensen, Kim Marriott, and Spiro Michaylov. Some global compile-time optimizations for CLP($\mathcal{R}$). In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 420–434, San Diego, CA, October 1991. MIT Press.

[5] Jean Jourdan and Raúl E. Valdés Pérez. Constraint logic programming applied to hypothetical reasoning in chemistry. In Saumya Debray and Manuel Hermenegildo, editors, *Logic Programming: Proceedings of the North American Conference, Austin, Texas*, pages 154–172. MIT Press, November 1990.

[6] Marc Kirschenbaum, Spiro Michaylov, and Leon Sterling. Skeletons and techniques as a normative approach to program develeopment in logic-based languages. Technical Report OSU-CISRC-5/94-TR25, Department of Computer and Information Science, The Ohio State University, 1994.

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

[8] Sivand Lakmazaheri and William J. Rasdorf. Constraint logic programming for the analysis and partial synthesis of truss structures. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 3(3):157–173, 1989.

[9] Catherine Lassez, Ken McAloon, and Roland Yap. Constraint logic programming and options trading. *IEEE Expert, Special Issue on Financial Software*, 2(3):42–50, August 1987.

[10] Kimbal G. Marriott and Peter J. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. In *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Charleston, NC, January 1993.

[11] Andrew D. McDonald, Peter J. Stuckey, and Roland H. C. Yap. Redundancy of variables in CLP($\mathcal{R}$). In *Logic Programming: Proceedings of the 1993 International Symposium*, Vancouver, October 1993. MIT Press.

[12] Spiro Michaylov. Repeated redundant inequalities in constraint logic programming. In Jean-Pierre Jouannaud, editor, *Proc. 1st International Conference on Constraints in Computational Logics*, Munich, Germany, September 1994. Longer version appears as Technical Report OSU-CISRC-6/94-TR31, Department of Computer and Information Science, The Ohio State University.

[13] Spiro Michaylov and Bill Pippin. Optimizing compilation of linear constraints in a class of constraint logic programs. In *Logic Programming: Proceedings of the 1994 International Symposium*, Ithaca, New York, November 1994.

[14] R. A. O'Keefe. *The Craft of Prolog*. MIT Press, 1990.

[15] L. Sterling. *The Practice of Prolog*. MIT Press, 1990.

[16] L. Sterling and E. Y. Shapiro. *The Art of Prolog*. MIT Press, 1986.

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

[18] Roland Yap Hoc Chuan. Restriction site mapping in CLP($\mathcal{R}$). In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 521–534, Paris, France, June 1991. MIT Press.