CHAPTER 1

INTRODUCTION

1.1. RECURSION AND PARALLELISM

Recursion is fundamental to logic programming. However, there is a vexatious problem with the traditional method for running recursive programs: it is sequential. As an example, consider the program for checking whether a number z is smaller than each element of a list:

$$\texttt{LessAll}(\emptyset, \texttt{z}).$$
$$\texttt{LessAll}(\texttt{x.y}, \texttt{z}) \leftarrow \texttt{z} < \texttt{x} \land \texttt{LessAll}(\texttt{y}, \texttt{z}).$$

Suppose we invoke this program with a goal containing a list of n elements. It will take n steps to do the n comparisons, since the n recursive calls to the program must be made in sequence.

Yet this problem is parallel in nature. It should, in principle, be possible to make all comparisons simultaneously since they are not interdependent. The sequentiality is imposed by our way of controlling the computation.

Tärnlund (1990) has proposed a solution to this problem: the Reform inference system. The idea is to specialize the program at run-time with respect to a particular goal. For example, specialization of the recursive clause of the `LessAll` program with respect to the goal

$$\leftarrow \texttt{LessAll}(\texttt{1.2...n.}\emptyset, \texttt{0})$$

adds the following clause to the program:

$$\texttt{LessAll}(\texttt{x}_1\texttt{...x}_\texttt{n}\texttt{.y}, \texttt{z}) \leftarrow \texttt{z} < \texttt{x}_1 \land \cdots \land \texttt{z} < \texttt{x}_\texttt{n} \land \texttt{LessAll}(\texttt{y}, \texttt{z}).$$

The program now represents a parallel algorithm and the n comparisons can be made simultaneously. Furthermore, there is an additional source of parallelism in the specialized program: unification of the list $\texttt{1.2...n.}\emptyset$ in the goal with the list $\texttt{x}_1\texttt{...x}_\texttt{n}\texttt{.y}$ in the clause head reduces to n independent subunifications that can proceed in parallel.

We shall take up this idea and consider the question: how do we uncover the parallel algorithm *at compile-time*? Our solution is to change the control strategy of the original program so that it behaves exactly like the specialized program, in particular with respect to opportunities for parallelism.

We compile recursive programs over recursive data structures (integers, lists, binary trees) to iterative programs over vectors. Consequently, we can exploit parallelism in the compiled programs by well-known parallelization techniques for iterative programs. These techniques were originally developed for Fortran-like programs.

1.2. OUTLINE OF THE THESIS

Chapter 2 contains some preliminary material.

Chapter 3 presents Tärnlund's inference system Reform. The novel idea with Reform is that it uses program transformation as computation: the recursive program is specialized at run-time with respect to a goal. The transformed program solves the goal much more efficiently than the original program and is highly parallel.

Chapter 4 describes different opportunities for parallelism in Reform computations. We identify two major sources of parallel speedup: recursion parallelism and parallel unification of large data structures. Recursion parallelism is an interesting generalization of AND-parallelism for recursive programs: all recursion levels are computed simultaneously, while the computation at each recursion level may be either sequential or parallel.

Chapter 5 analyses various patterns of variable instantiations in a Reform transformation. The analysis allows us to derive mappings from variables in the original program to vectors of terms in the transformed program. These mappings form the basis of our method of compilation.

Chapter 6 presents our compilation theory for the case of linear recursion. We develop the theory in some detail for list programs; other inductively defined data structures (for example, integers) can be handled analogously. The idea is to compile the recursive program to a program that uses bounded iteration (`for`-loops) over the vectors derived in chapter 5. We describe how recursion parallelism is implemented by application of standard parallelization techniques to the compiled program.

Chapter 7 generalizes the compilation method to nonlinear recursion. We compile binary tree programs to iteration over vectors, using a temporary linear representation of the recursion tree. This linear representation is obtained each time a tree program is invoked, as a side effect of head unification.

Chapter 8 discusses program transformation methods for extending the class of programs that can be handled by the compiler. We describe several transformation techniques informally.

Chapter 9 outlines the historical background of the subject matters of the thesis: recursion and logic programming.

Chapter 10 summarizes our results. We claim that our compilation method gives more parallelism and less implementational overhead than do earlier proposals for parallel logic programming. We expect, moreover, that the method can speed up logic programs on sequential machines.

CHAPTER 2

PRELIMINARIES

This chapter contains some preliminary material. We assume, however, that the reader is familiar with logic programming. We shall sometimes refer to the 'traditional method' for running logic programs; we are then referring to the top-down schemes, derived from Kowalski's (1974) procedural interpretation of logic programs, described by Clark (1990).

2.1. TYPOGRAPHICAL CONVENTIONS

Our object language is that of first-order predicate calculus. Elements of the object-level alphabets are written as follows. Variables: lowercase letters, for example `x`, `y`, `z`. Function symbols: uppercase letters, for example `T`, `F`, `P`. Predicate symbols: uppercase letter followed by zero or more characters, for example `Rev`, `LessAll`.

2.2. PRELIMINARY DEFINITIONS

Most general unifier. The most general unifier (Robinson, 1965) of two expressions x and y is denoted by $\mathrm{mgu}(x, y)$.

Variants. Let C be a clause and let the substitution σ be a bijection from the variables in C to a disjoint set of variables. Then the clause $C\sigma$ is a *variant* of C, and is denoted by C'.

Horn clauses. A *Horn clause* C is a (implicitly universially quantified) sentence of the form:

$$H \leftarrow B_1 \wedge \cdots \wedge B_k \wedge T_1 \wedge \cdots \wedge T_m,$$

where $k, m \geq 0$, such that H and $T_1, \ldots, T_m$ have the same predicate symbol and arity. We say that

- H is the *head* of the clause,
- $\{B_1, \ldots, B_k\}$ is the *residual body* of the clause,
- $\{T_1, \ldots, T_m\}$ is the *tail* of the clause.

Recursive Horn clauses. A Horn clause is *recursive* if its tail is non-empty.

Linear recursion. A recursive Horn clause is *linear* if its tail is a singleton. When this is the case we identify the tail with its only element.

2.3. LOOP PARALLELIZATION

In this thesis we shall take up some standard techniques for parallelization of iterative programs. For the reader who is not acquainted with these techniques we summarize the main points here.

A loop of n iterations can be computed in parallel on n processors by letting each processor perform one iteration of the loop. The loop can then get a n-fold speedup if the different iterations are independent (the loop program is assumed available on each processor). This technique, which has been successful for running Fortran-like programs in parallel, is known as *loop parallelization* (Kuck *et al.*, 1972).

Loop parallelization works only for bounded iteration (`for`-loops) since the number of iterations to be performed must be known when the loop is entered. Hence unbounded iteration (`while`-loops) cannot be parallelized in this way.

Consider a loop of n iterations computed in parallel on k processors. If $n = k$ then each of n processors is assigned one iteration of the loop. If $n > k$ then every kth iteration is assigned to the same processor. For example, the loop

```
for i=1 to 5 do
   a[i] := b[i]+c[i]
   d[i] := b[i]*2
end
```

can be computed as follows on three processors:

Time	Processor 1:	Processor 2:	Processor 3:
↓	`a[1] := b[1]+c[1]`	`a[2] := b[2]+c[2]`	
	`d[1] := b[1]*2`	`d[2] := b[2]*2`	`a[3] := b[3]+c[3]`
	`a[4] := b[4]+c[4]`	`a[5] := b[5]+c[5]`	`d[3] := b[3]*2`
	`d[4] := b[4]*2`	`d[5] := b[5]*2`	

Loops with data dependencies can often be transformed so that parallel computation is possible. Let us consider a simple example of a loop that contains a data dependency. The first statement of the ith iteration produces a value needed in the second statement of the $(i + 1)$th iteration:

```
for i=2 to n do
   a[i] := b[i]+c[i]
   d[i] := b[i]+a[i-1]
end
```

A parallelizing compiler would transform this loop into two adjacent loops, each of which can be parallelized:

```
for i=2 to n do
   a[i] := b[i]+c[i]
end
for i=2 to n do
   d[i] := b[i]+a[i-1]
end
```

2.4. LOW-LEVEL NOTATION

Below we explain some conventions used for imperative programs (compiled logic programs).

1. The effect of the operation `unify(x,y)` is that `x` and `y` are unified. The symbol `undef` denotes the value of an unbound logical variable.

2. Lists are linked structures of list cells. Each list cell has a head slot, which holds an element of the list, and a tail slot, which holds a pointer to another list cell or the empty list `nil`. We refer to the contents of the head and tail slots of a list cell `x` as `head(x)` and `tail(x)`, respectively. The expression `cons(x,y)` denotes a list cell containing `x` in the head slot and `y` in the tail slot.

3. We refer to the address of an object `x` as `addr(x)`. Thus, if we want to point the tail slot of a list `x` to an object `y` we can write `addr(tail(x)) := y`.

4. Besides the conventional notations

`for i=1 to k do ...` and `for i=k downto 1 do ...`

for sequential loops, we use the notation

```
for* i=1 to k do ...
```

for parallel loops, in the sense of the preceding section.

CHAPTER 3

REFORM

The Reform inference principle (Tärnlund, 1990) offers a way to construct a program that is specialized with respect to a particular size of the input data. For logic programming, Reform can be formulated as a Resolution (Robinson, 1965) proof procedure. In this chapter we consider the subset of Reform that is relevant for linear recursion.

The Reform principle. Consider two recursive clauses S and $\bar{S}$:

$$S = (H \leftarrow \Psi \wedge T) \qquad \text{and} \qquad \bar{S} = (\bar{H} \leftarrow \bar{\Psi} \wedge \bar{T}),$$

where H, Ψ, T is the head, residual body, and tail, respectively, of S, and $\bar{H}$, $\bar{\Psi}$, $\bar{T}$ is the head, residual body, and tail, respectively, of $\bar{S}$. Suppose that S and $\bar{S}$ have no variables in common, and that $\sigma = \text{mgu}(T, \bar{H})$. The resolvent obtained in a Reform step using (separated variants of) S and $\bar{S}$ is called a *reforment* and is denoted by $S \diamond \bar{S}$:

$$S \diamond \bar{S} = (H \leftarrow \Psi \wedge \bar{\Psi} \wedge \bar{T})\sigma.$$

Such a step is called a *reformation*.

Example 3.1. A reformation* of (1) and (2) gives (3), i.e., (1) $\diamond$ (2) = (3):

$$\texttt{Append}(\texttt{u}_1.\texttt{x}_1,\texttt{y}_1,\texttt{u}_1.\texttt{z}_1) \leftarrow \underline{\texttt{Append}(\texttt{x}_1,\texttt{y}_1,\texttt{z}_1)} \tag{1}$$

$$\underline{\texttt{Append}(\texttt{u}_2.\texttt{x}_2,\texttt{y}_2,\texttt{u}_2.\texttt{z}_2)} \leftarrow \texttt{Append}(\texttt{x}_2,\texttt{y}_2,\texttt{z}_2). \tag{2}$$

$$\texttt{Append}(\texttt{u}_1.\texttt{u}_2.\texttt{x}_2,\texttt{y}_2,\texttt{u}_1.\texttt{u}_2.\texttt{z}_2) \leftarrow \texttt{Append}(\texttt{x}_2,\texttt{y}_2,\texttt{z}_2) \tag{3}$$

The nth reforment. The next definition introduces the notion of the nth reforment, S^n, for $n \geq 1$:

$$S^1 = S;$$
$$S^{n+1} = S^n \diamond S.$$

Thus, the nth reforment of a clause $H \leftarrow \Psi \wedge T$ is:

$$H_1 \leftarrow \Psi_1 \wedge \cdots \wedge \Psi_n \wedge T_n,$$

* We adopt the convention to underline the literals selected for unification.

where H_1 is an instance of H, $\Psi_1, \ldots, \Psi_n$ are instances of Ψ, and T_n is an instance of T.

Example 3.2. The third reforment of (4) is (5), i.e., $(4)^3 = (5)$:

$$\mathtt{Append(u.x,y,u.z)} \leftarrow \mathtt{Append(x,y,z)} \tag{4}$$

$$\mathtt{Append(u_1.u_2.u_3.x_3,y_3,u_1.u_2.u_3.z_3)} \leftarrow \mathtt{Append(x_3,y_3,z_3)} \tag{5}$$

The definition of the nth reforment hints at an algorithm in which S^n is derived by repeated reformations of a variant of S and the result of the last reformation:

1. Let $S^1 \leftarrow S$ and $i \leftarrow 1$.
2. If $i = n$ then stop, else increment i by 1 and continue.
3. Set $S^i \leftarrow S^{i-1} \diamond S$ and go to step 2.

The efficiency of this algorithm is not the best possible. Its merit is simplicity, which we shall need in order analyse the sequence of unifiers produced by it.

Example 3.3. Consider again the program:

$$\begin{aligned}&\mathtt{LessAll(\emptyset,z).}\\ &\mathtt{LessAll(x.y,z)} \leftarrow \mathtt{z < x} \wedge \mathtt{Lessall(y,z).}\end{aligned} \tag{6}$$

Suppose that the program is invoked with a call:

$$\leftarrow \mathtt{LessAll(2.3.4.5.\emptyset,1)} \tag{7}$$

The Reform computation of (6) and (7) amounts to construction of the 4th reforment of the recursive clause with which the call is then resolved. The first step is a reformation of two variants (8) and (9) of the recursive clause:

$$\mathtt{LessAll(x_1.y_1,z_1)} \leftarrow \mathtt{z_1 < x_1} \wedge \underline{\mathtt{LessAll(y_1,z_1)}}. \tag{8}$$

$$\underline{\mathtt{LessAll(x_2.y_2,z_2)}} \leftarrow \mathtt{z_2 < x_2} \wedge \mathtt{LessAll(y_2,z_2)}. \tag{9}$$

The reforment derived is (10). The second step is a reformation of (10) and a new variant (11) of the recursive program clause:

$$\mathtt{LessAll(x_1.x_2.y_2,z_2)} \leftarrow \mathtt{z_2 < x_1} \wedge \mathtt{z_2 < x_2} \wedge \underline{\mathtt{LessAll(y_2,z_2)}}. \tag{10}$$

$$\underline{\mathtt{LessAll(x_3.y_3,z_3)}} \leftarrow \mathtt{z_3 < x_3} \wedge \mathtt{LessAll(y_3,z_3)}. \tag{11}$$

The reforment derived is (12). The third step is a reformation of (12) and a new variant (13) of the recursive program clause:

$$\begin{aligned}&\mathtt{LessAll(x_1.x_2.x_3.y_3,z_3)} \leftarrow\\ &\quad \mathtt{z_3 < x_1} \wedge \mathtt{z_3 < x_2} \wedge \mathtt{z_3 < x_3} \wedge \underline{\mathtt{LessAll(y_3,z_3)}}.\end{aligned} \tag{12}$$

$$\underline{\mathtt{LessAll(x_4.y_4,z_4)}} \leftarrow \mathtt{z_4 < x_4} \wedge \mathtt{LessAll(y_4,z_4)}. \tag{13}$$

In this step the 4th reforment (14) is obtained:

$$\texttt{LessAll}(x_1.x_2.x_3.x_4.y_4, z_4) \leftarrow \\ z_4 < x_1 \land z_4 < x_2 \land z_4 < x_3 \land z_4 < x_4 \land \texttt{LessAll}(y_4, z_4). \tag{14}$$

Finally, the call (7) is resolved with (14). The new goal obtained is:

$$\leftarrow 1 < 2 \land 1 < 3 \land 1 < 4 \land 1 < 5 \land \texttt{LessAll}(\emptyset, 1) \tag{15}$$

Example 3.4. Tärnlund (1990) gives a more efficient algorithm for Reform computation that requires only $\lceil \log n \rceil$ reformations to compute the nth reforment. A computation of the reforment (14) in the previous example is carried out as follows with this algorithm. The first step is a reformation of (8) and (9) which produces (10). The second step is a reformation of (10) and a new variant (16) of (10):

$$\texttt{LessAll}(x_1.x_2.y_2, z_2) \leftarrow z_2 < x_1 \land z_2 < x_2 \land \underline{\texttt{LessAll}(y_2, z_2)}. \tag{10}$$

$$\underline{\texttt{LessAll}(x_3.x_4.y_4, z_4)} \leftarrow z_4 < x_3 \land z_4 < x_4 \land \texttt{LessAll}(y_4, z_4). \tag{16}$$

The reforment derived is (14).

CHAPTER 4

PARALLELISM

A parallel logic programming system must be able to successfully exploit parallelism in recursive programs. In this chapter we shall see that Reform differs from traditional logic programming systems in that it gives rich opportunities for parallel speedup of recursive programs.

4.1. RECURSION PARALLELISM

Let us consider a recursive clause with k goals in the residual body:

$$H \leftarrow B_1 \wedge \cdots \wedge B_k \wedge T.$$

The nth reforment of this clause is on the form:

$$H \leftarrow B_{11} \wedge \cdots \wedge B_{k1} \wedge \cdots \wedge B_{1n} \wedge \cdots \wedge B_{kn} \wedge T.$$

It is instructive to depict the residual body of this reforment as a matrix of n rows and k columns:

$$\begin{pmatrix} B_{11} & \dots & B_{k1} \\ \vdots & \ddots & \vdots \\ B_{1n} & \dots & B_{kn} \end{pmatrix}$$

How can the calls in this matrix be computed in parallel? First, the k calls within each row may be computed in parallel. This is traditional AND-parallelism. Second, the n rows may be computed in parallel. This generalization of AND-parallelism may be called *recursion parallelism* since it corresponds to computing all recursion levels in parallel.

It should be noticed that typically $k \ll n$. This suggests that the potential for parallel speedup is greater with recursion parallelism than with traditional AND-parallelism.

Note that recursion parallelism cannot be exploited in a traditional logic programming system since the recursion levels must be computed in sequence with such a system. A restricted form, which may be called *recursion pipelining*, can however be obtained if the recursion levels are independent. The calls at one level can then start computing once the recursive call at the preceding level has been reduced; it is not necessary that the other calls at the preceding level have terminated.

Example 4.1. Consider the recursive clause of the `Lessall` program:

```
Lessall(x.y, z) ← z < x ∧ Lessall(y, z).
```

The matrix has only one column, since there is only one call in the residual body:

$$\begin{pmatrix} z < x_1 \\ z < x_2 \\ \vdots \\ z < x_n \end{pmatrix}$$

There are, of course, no opportunities for parallelism within each row of this matrix. The n rows, however, can be computed in parallel.

Note that this is an example where recursion pipelining is unlikely to speed up the computation: the amount of work in head unification is comparable with the amount of work in the body, hence the call $z < x_i$ is likely to terminate before the call $z < x_{i+1}$ can be initiated.

4.2. UNIFICATION PARALLELISM

A feature of logic programming is that composition and decomposition of data structures is done by unification. Hence a large share of the execution time of a clause is typically spent in head unification.

It is unusual that the execution of a program clause can be significantly speeded up by parallel unification in a traditional logic programming system. Consider, for example, the `Append` program:

```
Append(∅, y, y).
Append(u.x, y, u.z) ← Append(x, y, z).
```

All productive work in this program is done by head unifications. There are no opportunities for parallelism with a traditional system.

Let us now consider the unification parallelism of a Reform computation with this program. The nth reforment obtained by Reform computation is:

$$\texttt{Append}(u_1 \ldots u_n.x, y, u_1 \ldots u_n.z) \leftarrow \texttt{Append}(x, y, z).$$

Suppose that the transformed `Append` program is invoked with a call containing a list of n elements in the first argument position:

$$\leftarrow \texttt{Append}(\texttt{C}_1 \ldots \texttt{C}_\texttt{n}.\emptyset, \texttt{D}.\emptyset, \texttt{w})$$

Unification of the lists $\texttt{C}_1 \ldots \texttt{C}_\texttt{n}.\emptyset$ and $\texttt{u}_1 \ldots \texttt{u}_\texttt{n}.\texttt{x}$ reduces to n subunifications that might be computed in parallel. Furthermore, there is an additional source of parallelism if the Reform computation is compiled: we can then speed up the construction of the output list $\texttt{u}_1 \ldots \texttt{u}_\texttt{n}.\texttt{z}$ with parallel methods.

Unification parallelism, in the context of Reform computations, is a special case of recursion parallelism since it amounts to computing recursion levels in parallel. It might seem that the problem of unifying, say, two lists of n elements exhibits very little parallelism when lists are implemented as linked structures. This is not the case, however. There are algorithms that carry out such problems in $\log n$ steps for typical lists (Barklund, 1990).

In this thesis we restrict the discussion to unification-based logic programming languages. It should be noted, however, that Reform is equally applicable to constraint logic programming languages. The analog of parallel unification in such languages is parallelism in the algorithm for testing satisfiability of constraint systems. Reform with constraint logic programming languages has the effect of adding large chunks of new constraints to the constraint system at the invocation of a recursive program. This may be of significant value in case the algorithm for testing satisfiability has an overhead in checking that the old system is still satisfiable when new components are added. This overhead is multiplied in the traditional way of executing recursive programs, since new constraints are added much more frequently.

4.3. SEQUENTIAL PROGRAMS

Unification parallelism can often be exploited in otherwise sequential recursive programs. We consider the following program for reversing a list using `Append`:

```
Nrev(∅, ∅).
Nrev(u.x, y) ← Nrev(x, z) ∧ Append(z, u.∅, y).
```

The calls to `Append` cannot be computed in parallel due to data dependencies. However, Reform uncovers a significant degree of parallelism in this seemingly sequential program: each call to `Append` gives rise to list unifications that can be parallelized, as does the initial call to `Nrev`. Furthermore, we can break the data dependencies of the `Nrev` program with a method discussed in the next section.

4.4. PARALLELIZING SEQUENTIAL PROGRAMS

We shall now describe a method for breaking data dependencies in some programs, for example the list reversal program discussed above. The method is based on the assumption that we can compute the relative sizes of certain terms in the program at compile-time. Fortunately, recent work by Debray, Lin & Hermenegildo (1990), on quite another subject, gives us an algorithm for this task. Let us again consider the list reversal program:

```
Nrev(∅, ∅).
Nrev(u.x, y) ← Nrev(x, z) ∧ Append(z, u.∅, y).
```

The algorithm of Debray, Lin & Hermenegildo infers that if $size(u.x) = n + 1$, then $size(z) = size(x) = n$. We can, therefore, rewrite the program as follows, where the goal `Length(x, n)` computes the number of elements `n` in the list `x`.

```
Nrev(x, y) ← Length(x, n) ∧ Nrev*(n, x, y).
Nrev*(0, ∅, ∅).
Nrev*(n + 1, u.x, y) ← Nrev*(n, x, z) ∧ Append*(n, z, u.∅, y).
```

Here `Append*` is the result of a similar transformation of the `Append` program:

```
Append*(0, ∅, x, x).
Append*(n + 1, v.x, y, v.z) ← Append*(n, x, y, z).
```

Now, suppose the `Nrev` program is called with a goal containing a list of four elements, `A.B.C.D.∅`, say. We compute the following matching reforment:

```
Nrev*(4, u₁.u₂.u₃.u₄.x, y₁) ←
    Nrev*(0, x, z) ∧
    Append*(0, z, u₄.∅, y₄) ∧
    Append*(1, y₄, u₃.∅, y₃) ∧
    Append*(2, y₃, u₂.∅, y₂) ∧
    Append*(3, y₂, u₁.∅, y₁).
```

Let us see what happens when the residual body of the reforment is executed. The first `Append*` call matches the base clause. The remaining three calls to `Append*` matches the following reforments, respectively:

```
Append*(1, v₁.x, y, v₁.z) ← Append*(0, x, y, z).
Append*(2, v₁.v₂.x, y, v₁.v₂.z) ← Append*(0, x, y, z).
Append*(3, v₁.v₂.v₃.x, y, v₁.v₂.v₃.z) ← Append*(0, x, y, z).
```

Suppose now that the four `Append*` calls start computing in parallel. The following events will then take place more or less simultaneously (the symbol '_' denotes a new variable):

- The call `Append*(0, z, D.∅, y₄)` binds y_4 to `D.∅`.

- The call `Append*(1,` y_4`,C.∅,` y_3`)` matches the first reforment and binds y_3 to a list `_.C.∅`.
- The call `Append*(2,` y_3`,B.∅,` y_2`)` matches the second reforment and binds y_2 to a list `_._.B.∅`.
- The call `Append*(3,` y_2`,A.∅,` y_1`)` matches the third reforment and binds y_1 to a list `_._._.A.∅`.

Depending on the exact timing of these events, the lists y_3, y_2 and y_1 might have further elements instantiated when they are created. Otherwise, the list elements are instantiated after the lists are created.

The result of our reformulation of the program is thus that each consumer of a list produces its own input list; the (former) producer then merely instantiates the individual elements of the list. This demonstrates the power of logical variables: they allow us to refer to the result of a computation before the result is actually computed.

The new program `Nrev*` does not contain the data dependencies that impeded parallelization of the original program `Nrev`. This method for breaking data dependencies can, of course, only be applied when the goals that contain data dependencies are calling recursive programs (`Append` in our example).

4.5. CONCLUSION

It is quite obvious that the greatest need for parallel speedup emerges with large computations. Large computations are specified by recursion in logic programming. Recursive programs are, thus, of vital concern also for a traditional AND-parallel system. We have seen, however, that such systems are unnecessarily sequential: the n invocations of a recursive program must be made in sequence.

The inherent parallelism in recursive programs is uncovered by Reform. We have identified *recursion parallelism* and *unification parallelism* as new techniques for accelerating logic programs with Reform. Later we shall see how recursive programs can be compiled to exhibit recursion parallelism and unification parallelism without explicit Reform transformation.

CHAPTER 5

CHARACTERISTIC UNIFIERS

In this chapter we shall pursue an analysis of Reform transformations showing that the instantiations of variables in recursive clauses conform to a few regular patterns. The obtained foreknowledge about the variables' instantiations will allow us to derive mappings from variables in the original program to vectors of terms in the transformed program.

5.1. REFORM SERIES

A new variant of the recursive program clause is created for each step in a Reform transformation. Each variant contains a new set of variables. We shall need to make a distinction between the variables that occur in the original program clause, and the corresponding variables occurring in the variant clauses used in the transformation.

Program variables and variant variables (terms). A variable occurring in a program clause C is called a *program variable*. A variable occurring in a variant C' of C is called a *variant variable*. The notions of *program term* and *variant term* are straight-forward extensions.

We can associate with each program variable a sequence of variant terms for each Reform transformation. The main objective of the analysis in this chapter is to characterize the relationship between program variables and their associated sequences for different classes of program variables.

Standard renaming of variables. It is evident that n variants of a recursive clause are needed in the derivation of the nth reforment of that clause according to the algorithm given in chapter 3. Let us number these variant clauses from 1 to n according to the order in which they are used in the algorithm.

By the *standard renaming* of variables in the variant clauses we understand the convention to rename a program variable v to v_i of the ith variant clause. The standard renaming will be tacit in the following discussion.

Unification of substitutions. We shall take up an idea introduced by Tärnlund (1975): unification of substitutions. Let θ and σ be substitutions. The substitution λ defined by

$$\theta\lambda = \sigma\lambda$$

is a unifier of θ and σ. As usual, λ is the most general unifier (mgu) of θ and σ if and only if for every unifier α, there is a substitution β such that $\alpha = \lambda\beta$. We shall denote by $\theta * \sigma$ the mgu of θ and σ. The following result follows directly from the definition of $\theta * \sigma$.

Theorem 5.1. *Let θ and σ be subsitutions and $\theta * \sigma$ the mgu of θ and σ. Then $\theta * \sigma = \sigma * \theta$.*

Composition by unification often, but not always, gives the same result as functional composition. For example, let $\theta = \{F(y)/z\}$ and $\sigma = \{F(x)/y\}$. We then have

$$z\theta\sigma = F(F(x)), \qquad z(\theta * \sigma) = F(F(x)),$$

but

$$z\sigma\theta = F(y), \qquad z(\sigma * \theta) = F(F(x)).$$

The example also shows that functional composition of substitutions is not always commutative (Robinson, 1979).

Reform series. Consider a recursive clause S and its nth reforment S^n. The computation of the S^n produces $n-1$ mgu's; let us by σ_i denote the mgu computed in deriving S^i (we define $\sigma_1 = \epsilon$, the empty substitution). We define the composition

$$\mu = \sigma_1 * \cdots * \sigma_n.$$

A program variable x of S is represented by n variant variables $x_1, x_2, \ldots, x_n$ in the derivation of S^n. The *Reform series* $\bar{x}$ of x is then the sequence

$$\bar{x} = \langle x_1\mu, \ldots, x_n\mu \rangle.$$

We shall often write $\bar{x}$ as $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$.

The notion of Reform series is easily extended to compound terms. The Reform series of a compound term t is the sequence

$$\bar{t} = \langle t_1\mu, \ldots, t_n\mu \rangle.$$

Example 5.1. Consider the recursive program for reversing a list using an accumulating parameter:

```
Rev(∅, y, y).
Rev(u.x, y, z) ← Rev(x, u.y, z).
```

Let us carry out a Reform transformation for $n = 3$ with this program. The first step is a reformation of clauses (1) and (2), the first two variants of the recursive `Rev` clause according to the standard renaming of variables:

$$\texttt{Rev(u}_1\texttt{.x}_1\texttt{, y}_1\texttt{, z}_1\texttt{)} \leftarrow \underline{\texttt{Rev(x}_1\texttt{, u}_1\texttt{.y}_1\texttt{, z}_1\texttt{)}}, \qquad (1)$$

$$\underline{\texttt{Rev(u}_2\texttt{.x}_2\texttt{, y}_2\texttt{, z}_2\texttt{)}} \leftarrow \texttt{Rev(x}_2\texttt{, u}_2\texttt{.y}_2\texttt{, z}_2\texttt{)}, \qquad (2)$$

We obtain the reforment (3). The next step is a reformation of (3) and (4), giving (5):

$$\mathtt{Rev(u_1.u_2.x_2\,,\,y_2\,,\,z_2)} \leftarrow \underline{\mathtt{Rev(x_2\,,\,u_2.u_1.y_1\,,\,z_2)}}. \tag{3}$$

$$\underline{\mathtt{Rev(u_3.x_3\,,\,y_3\,,\,z_3)}} \leftarrow \mathtt{Rev(x_3\,,\,u_3.y_3\,,\,z_3)}, \tag{4}$$

$$\mathtt{Rev(u_1.u_2.u_3.x_3\,,\,y_3\,,\,z_3)} \leftarrow \mathtt{Rev(x_3\,,\,u_3.u_2.u_1.y_1\,,\,z_3)}. \tag{5}$$

The mgu's σ_2 and σ_3 (recall that $\sigma_1 = \epsilon$) computed in the derivation of reforments (3) and (5), respectively, are:

$$\sigma_2 = \{\mathtt{u_2.x_2/x_1\,,\ u_1.y_1/y_2\,,\ z_2/z_1}\}, \qquad \sigma_3 = \{\mathtt{u_3.x_3/x_2\,,\ u_2.u_1.y_1/y_3\,,\ z_3/z_2}\}.$$

Let $\mu = (\sigma_1 * \sigma_2 * \sigma_3)$. The Reform series of the program variables u, x, y, and z are thus:

$$\begin{aligned}
\bar{\mathtt{u}} &= \langle \mathtt{u_1\,,\,u_2\,,\,u_3}\rangle\mu = \langle \mathtt{u_1\,,\,u_2\,,\,u_3}\rangle;\\
\bar{\mathtt{x}} &= \langle \mathtt{x_1\,,\,x_2\,,\,x_3}\rangle\mu = \langle \mathtt{u_2.u_3.x_3\,,\ u_3.x_3\,,\ x_3}\rangle;\\
\bar{\mathtt{y}} &= \langle \mathtt{y_1\,,\,y_2\,,\,y_3}\rangle\mu = \langle \mathtt{y_1\,,\ u_1.y_1\,,\ u_2.u_1.y_1}\rangle.\\
\bar{\mathtt{z}} &= \langle \mathtt{z_1\,,\,z_2\,,\,z_3}\rangle\mu = \langle \mathtt{z_3\,,\,z_3\,,\,z_3}\rangle;
\end{aligned}$$

5.2. THE CHARACTERISTIC MGU

Let us consider a clause $S = (H \leftarrow \Psi \wedge T)$. Suppose that the ith and $(i+1)$th variants of S $(1 \leq i < n)$, according to the standard renaming of variables, are:

$$S_i = (H_i \leftarrow \Psi_i \wedge T_i) \qquad \text{and} \qquad S_{i+1} = (H_{i+1} \leftarrow \Psi_{i+1} \wedge T_{i+1}).$$

Now, let $\theta_{i+1} = \mathrm{mgu}(T_i, H_{i+1})$. We refer to θ_{i+1} as the *characteristic mgu* of the clause S.

The characteristic mgu contains information that allows us to compute the Reform series of length n, without explicit Reform transformation, if the nth reforment exists. That is, we can compute $\mu = \sigma_1 * \cdots * \sigma_n$, and hence the Reform series, from θ_{i+1} alone. (We shall deal with the case when the nth reforment does not exist at the end of this chapter.)

Example 5.2. The characteristic mgu of the clause

$$\mathtt{Rev(u.x\,,\,y\,,\,z)} \leftarrow \mathtt{Rev(x\,,\,u.y\,,\,z)}$$

is $\theta_{i+1} = \{\mathtt{u_{i+1}.x_{i+1}/x_i\,,\,u_i.y_i/y_{i+1}\,,\,z_{i+1}/z_i}\}$.

Note that an ambiguity arises whenever one variable is substituted for another, as with $\mathtt{z_i}$ and $\mathtt{z_{i+1}}$ in the Rev example. We adopt the convention to substitute the variable occurring in the head for the variable occurring in the tail.

Note also that we can infer a variable's position in the recursive clause—whether it occurs in the head or in the tail—from the characteristic mgu. If θ_{i+1} contains a substitution element t/x_i (for some term t), then x occurs in the tail of the clause. Similarly, a substitution element t/x_{i+1} in θ_{i+1} implies that x occurs in the head.

In the rest of this section we shall outline why the Reform series can be constructed from the characteristic mgu. For this purpose we shall now consider a slightly different way of carrying out Reform transformations, without changing the result of the transformation.

Skeleton representation of reforments. In the computation of the nth reforment S^n of a clause S, we shall not construct the 'intermediate' reforments $S^2, \ldots, S^{n-1}$ explicitly. Rather, we shall represent the intermediate reforments as *skeletons*. A skeleton reforment of two clauses

$$S = (H \leftarrow \Psi \wedge T) \qquad \text{and} \qquad S' = (H' \leftarrow \Psi' \wedge T'),$$

is $H \leftarrow \Psi \wedge \Psi' \wedge T'$, if T and H' are unifiable. That is, the skeleton reforment is obtained by the usual procedure for reformation, except that the obtained mgu is not applied.

Let us now use the skeleton representation of reforments in the derivation of the nth reforment S^n. The $(i+1)$th $(1 \leq i < n)$ 'skeleton reforment' and corresponding mgu are:

$$H_1 \leftarrow \Psi_1 \wedge \cdots \wedge \Psi_{i+1} \wedge T_{i+1} \qquad \text{and} \qquad \theta_{i+1} = \text{mgu}(T_i, H_{i+1}).$$

The mgu's $\theta_2, \ldots, \theta_n$ $(\theta_1 = \epsilon$ by definition) obtained in the reformations are all variants of the characteristic mgu (and, obviously, of each other). The nth reforment S^n is obtained by applying $\nu = \theta_1 * \cdots * \theta_n$ to the 'nth skeleton.'

Let us carry out a Reform transformation for $n = 3$ with the `Rev` program of the previous example using this idea. The first step is a reformation of the variant clauses (6) and (7):

$$\mathtt{Rev(u_1.x_1, y_1, z_1)} \leftarrow \underline{\mathtt{Rev(x_1, u_1.y_1, z_1)}}\mathtt{.} \tag{6}$$
$$\underline{\mathtt{Rev(u_2.x_2, y_2, z_2)}} \leftarrow \mathtt{Rev(x_2, u_2.y_2, z_2).} \tag{7}$$

We obtain the skeleton (8) and the mgu $\theta_2 = \{\mathtt{u_2.x_2/x_1,\ u_1.y_1/y_2,\ z_2/z_1}\}$. The next step is a reformation of (8) and a new variant clause (9):

$$\mathtt{Rev(u_1.x_1, y_1, z_1)} \leftarrow \underline{\mathtt{Rev(x_2, u_2.y_2, z_2)}}\mathtt{.} \tag{8}$$
$$\underline{\mathtt{Rev(u_3.x_3, y_3, z_3)}} \leftarrow \mathtt{Rev(x_3, u_3.y_3, z_3),} \tag{9}$$

We obtain the skeleton (10) and the mgu $\theta_3 = \{\mathtt{u_3.x_3/x_2,\ u_2.y_2/y_3,\ z_3/z_2}\}$.

$$\mathtt{Rev(u_1.x_1, y_1, z_1)} \leftarrow \mathtt{Rev(x_3, u_3.y_3, z_3).} \tag{10}$$

We now compute the composition

$$\nu = \theta_1 * \theta_2 * \theta_3 = \{\mathtt{u}_2.\mathtt{u}_3.\mathtt{x}_3/\mathtt{x}_1,\ \mathtt{u}_2.\mathtt{u}_1.\mathtt{y}_1/\mathtt{y}_3,\ \mathtt{z}_3/\mathtt{z}_1\}.$$

Applying ν to the skeleton (10) yields the reforment (11):

$$\mathtt{Rev}(\mathtt{u}_1.\mathtt{u}_2.\mathtt{u}_3.\mathtt{x}_3,\mathtt{y}_1,\mathtt{z}_3) \leftarrow \mathtt{Rev}(\mathtt{x}_3,\mathtt{u}_3.\mathtt{u}_2.\mathtt{u}_1.\mathtt{y}_1,\mathtt{z}_3). \qquad (11)$$

We see that (11) is, in fact, the reforment (5) derived earlier. This example illustrates that $\mu = \nu$.

We noted above that the mgu's $\theta_2, \ldots, \theta_n$, obtained by using the skeleton representation, are variants of each other; this is not the case for the mgu's $\sigma_2, \ldots, \sigma_n$ obtained earlier. Let us illustrate this point by comparing $\sigma_2, \ldots, \sigma_n$ (cf. Example 5.1) and $\theta_2, \ldots, \theta_5$:

$$\begin{aligned}
\sigma_2 &= \{\mathtt{u}_2.\mathtt{x}_2/\mathtt{x}_1,\ \mathtt{u}_1.\mathtt{y}_1/\mathtt{y}_2,\ \mathtt{z}_2/\mathtt{z}_1\}\\
\sigma_3 &= \{\mathtt{u}_3.\mathtt{x}_3/\mathtt{x}_2,\ \mathtt{u}_2.\mathtt{u}_1.\mathtt{y}_1/\mathtt{y}_3,\ \mathtt{z}_3/\mathtt{z}_2\}\\
\sigma_4 &= \{\mathtt{u}_4.\mathtt{x}_4/\mathtt{x}_3,\ \mathtt{u}_3.\mathtt{u}_2.\mathtt{u}_1.\mathtt{y}_1/\mathtt{y}_4,\ \mathtt{z}_4/\mathtt{z}_3\}\\
\sigma_5 &= \{\mathtt{u}_5.\mathtt{x}_5/\mathtt{x}_4,\ \mathtt{u}_4.\mathtt{u}_3.\mathtt{u}_2.\mathtt{u}_1.\mathtt{y}_1/\mathtt{y}_5,\ \mathtt{z}_5/\mathtt{z}_4\}\\
\theta_2 &= \{\mathtt{u}_2.\mathtt{x}_2/\mathtt{x}_1,\ \mathtt{u}_1.\mathtt{y}_1/\mathtt{y}_2,\ \mathtt{z}_2/\mathtt{z}_1\}\\
\theta_3 &= \{\mathtt{u}_3.\mathtt{x}_3/\mathtt{x}_2,\ \mathtt{u}_2.\mathtt{y}_2/\mathtt{y}_3,\ \mathtt{z}_3/\mathtt{z}_2\}\\
\theta_4 &= \{\mathtt{u}_4.\mathtt{x}_4/\mathtt{x}_3,\ \mathtt{u}_3.\mathtt{y}_3/\mathtt{y}_4,\ \mathtt{z}_4/\mathtt{z}_3\}\\
\theta_5 &= \{\mathtt{u}_5.\mathtt{x}_5/\mathtt{x}_4,\ \mathtt{u}_4.\mathtt{y}_4/\mathtt{y}_5,\ \mathtt{z}_5/\mathtt{z}_4\}
\end{aligned}$$

The significance of $\theta_2, \ldots, \theta_n$ being variants of each other is that formal reasoning about the Reform computation is simplified.

We obtain the following theorem which follows directly from the definition of Reform employing the skeleton representation.

Theorem 5.2. *Let x be a variable, t a term, and $\theta_1, \ldots, \theta_n$ the mgu's derived in a Reform transformation. Then:*

$$\exists i(x_{i-1}\theta_i = t_i) \rightarrow \forall i(x_{i-1}\theta_i = t_i), \qquad 1 < i \leq n;$$

and

$$\exists i(x_i\theta_i = t_{i-1}) \rightarrow \forall i(x_i\theta_i = t_{i-1}), \qquad 1 < i \leq n.$$

A consequence of Theorem 5.2 is that we can determine the composition $\nu = \theta_1 * \cdots * \theta_n$ from *any one* of the mgu's $\theta_2, \ldots, \theta_n$. In other words, we can determine ν from the characteristic mgu.

5.3. CLASSIFICATION OF VARIABLES

We shall now classify variables according to the structure of their Reform series. To this end we examine the variable bindings made in the characteristic mgu. For each type of variable in the classification scheme we shall give a function that defines the Reform series of such a variable.

We introduce three basic types of variables. This allows a complete categorization of the variables in any recursive logic program. (A variable can belong to more than one category; we deal with this case in section 5.7.)

First we introduce two auxiliary functions, φ and τ.

The φ function. We denote by $\varphi(x, i)$ the ith variant variable constructed from the program variable x and the integer i ($i > 0$). Examples:

$$\varphi(\mathtt{x}, 4) = \mathtt{x}_4, \qquad \varphi(\mathtt{z}, 1) = \mathtt{z}_1.$$

The τ function. We denote by $\tau(t, \theta_c, i)$ a variant term constructed from the program term t, the characteristic mgu θ_c of the clause in which t occur, and the integer i ($i > 0$). The variant term $\tau(t, \theta_c, i)$ is the instance of t in the ith reforment. In other words, $\tau(t, \theta_c, i)$ is ith element of the Reform series of t. Examples:

$$\tau(\mathtt{x}, \{\mathtt{y_{i+1}}/\mathtt{x_i}\}, 5) = \mathtt{y}_6, \qquad \tau(\mathtt{x}, \{\mathtt{y_{i+1}}/\mathtt{z_i}\}, 5) = \mathtt{x}_5,$$
$$\tau(\mathtt{F(x)}, \{\mathtt{y_{i+1}}/\mathtt{x_i}\}, 3) = \mathtt{F(y_4)}, \qquad \tau(\mathtt{F(x)}, \{\mathtt{G(y_i)}/\mathtt{x_{i+1}}\}, 3) = \mathtt{F(G(y_2))}.$$

(Material introduced later in this chapter is required for understanding these applications of the τ function.)

In the definitions below $\theta_1, \ldots, \theta_n$ stand for the mgu's obtained in a Reform transformation using the skeleton representation of reforments. We refer to the characteristic mgu as θ_c.

NONE-variables. A variable x is a *NONE-variable* if and only if it is not substituted for in the characteristic mgu. Examples:

$$\mathtt{P(F(42,x))} \leftarrow \mathtt{P(y)} \qquad \text{and} \qquad \mathtt{P(y)} \leftarrow \mathtt{Q(y,x)} \wedge \mathtt{R(x,z)} \wedge \mathtt{P(z)}.$$

The function NONE defines the Reform series of a program variable x:

$$\text{NONE}(n, x) = \langle \bar{x}_1, \ldots, \bar{x}_n \rangle, \qquad \text{where} \qquad \bar{x}_i = \varphi(x, i), \qquad 1 \leq i \leq n.$$

Example: $\text{NONE}(4, \mathtt{x}) = \langle \mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4 \rangle$.

The following result follows directly from the definition of NONE-variables.

Theorem 5.3. *Let x be a NONE-variable. Then the Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of x is given by: $\bar{x}_i = \varphi(x,i)(\theta_1 * \cdots * \theta_n) = \varphi(x,i)$, for $1 \leq i \leq n$.*

POS-variables. Let t be a term. A variable x is a *POS-variable* if and only if

$$\varphi(x,i)\theta_{i+1} = \tau(t, \theta_c, i+1), \qquad \text{for} \quad 1 \leq i < n.$$

That is, x occurs in the tail of the clause and there is a term t in the corresponding position of the clause head. Examples:

$$\mathtt{P(F(y))} \leftarrow \mathtt{P(x)} \qquad \text{and} \qquad \mathtt{P(F(F(x)))} \leftarrow \mathtt{P(F(x))}.$$

The function POS defines the Reform series of x:

$$\mathrm{POS}(n, x, t, \theta_c) = \langle \bar{x}_1, \ldots, \bar{x}_n \rangle, \quad \text{where} \quad \bar{x}_i = \begin{cases} \varphi(x,n), & i = n; \\ \tau(t, \theta_c, i+1), & 1 \leq i < n. \end{cases}$$

Example: $\mathrm{POS}(4, \mathtt{x}, \mathtt{y}, \{\mathtt{y_{i+1}}/\mathtt{x_i}\}) = \langle \mathtt{y_2}, \mathtt{y_3}, \mathtt{y_4}, \mathtt{x_4} \rangle$.

We obtain the following lemma from the definition of POS-variable.

Lemma 5.1. *Let x be a POS-variable and t the corresponding term in the clause head. Then, for $k \geq 1$:*

$$\varphi(x,k)(\theta_1 * \cdots * \theta_k) = \varphi(x,k) \qquad \textit{and} \qquad \varphi(x,k)(\theta_1 * \cdots * \theta_{k+1}) = \tau(t, \theta_c, k+1).$$

We shall now show that the function POS defined above does indeed compute the Reform series of a POS-variable.

Theorem 5.4. *Let x be a POS-variable and t the corresponding term in the clause head. Then the Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of x is given by:*

$$\bar{x}_i = \varphi(x,i)(\theta_1 * \cdots * \theta_n) = \begin{cases} \varphi(x,n), & i = n; \\ \tau(t, \theta_c, i+1), & 1 \leq i < n. \end{cases}$$

Proof. The proof is by induction on the length of the Reform transformation.
Base ($n = 2$): From Lemma 5.1 we know that $\varphi(x,1)\theta_2 = \tau(t, \theta_c, 2)$ and $\varphi(x,2)\theta_2 = \varphi(x,2)$. Hence $\varphi(x,1)(\theta_1 * \theta_2) = \tau(t, \theta_c, 2)$ and $\varphi(x,2)(\theta_1 * \theta_2) = \varphi(x,2)$, since $\theta_1 = \epsilon$).

Induction hypothesis: $\varphi(x,i)(\theta_1 * \cdots * \theta_n) = \begin{cases} \varphi(x,n), & i = n; \\ \tau(t, \theta_c, i+1), & 1 \leq i < n. \end{cases}$

Induction step: We have three cases.

$(i < n)$
$\varphi(x,i)(\theta_1 * \cdots * \theta_{n+1}) \quad = \tau(t, \theta_c, i+1)(\theta_1 * \cdots * \theta_{n+1}) \qquad$ by induction hypothesis

$$= \tau(t, \theta_c, i+1) \qquad \text{by definition of } \tau$$

(i=n)

$$\varphi(x,i)(\theta_1 * \cdots * \theta_{n+1}) \quad = \tau(t, \theta_c, n+1) \qquad \text{by Lemma 5.1}$$

$(i = n+1)$

$$\varphi(x,i)(\theta_1 * \cdots * \theta_{n+1}) \quad = \varphi(x, n+1) \qquad \text{by Lemma 5.1}$$

Hence $\varphi(x,i)(\theta_1 * \cdots * \theta_{n+1}) = \begin{cases} \varphi(x, n+1), & i = n+1; \\ \tau(t, \theta_c, i+1), & 1 \le i \le n. \end{cases}$ ∎

Example 5.3. Consider the recursive clause

$$\texttt{P(F(y),z)} \leftarrow \texttt{P(x,y)}.$$

We have the following classification and Reform series of the variables.

Variable	Category	Reform series
z	NONE	$\langle z_1, z_2, z_3, z_4 \rangle$
y	POS	$\langle z_2, z_3, z_4, y_4 \rangle$
x	POS	$\langle F(z_3), F(z_4), F(y_4), x_4 \rangle$

NEG-variables. Let t be a non-variable term. A variable x is a *NEG-variable* if and only if

$$\varphi(x, i+1)\theta_{i+1} = \tau(t, \theta_c, i), \qquad \text{for} \quad 1 \le i < n.$$

That is, x occurs in the clause head and there is a non-variable term t in the corresponding position of the tail of the clause. Examples:

$$\texttt{P(x)} \leftarrow \texttt{P(F(y))} \qquad \text{and} \qquad \texttt{P(F(x))} \leftarrow \texttt{P(F(F(x)))}.$$

The function NEG defines the Reform series of x:

$$\text{NEG}(n, x, t, \theta_c) = \langle \bar{x}_1, \ldots, \bar{x}_n \rangle, \quad \text{where} \quad \bar{x}_i = \begin{cases} \varphi(x, 1), & i = 1; \\ \tau(t, \theta_c, i-1), & 1 < i \le n. \end{cases}$$

Example: $\text{NEG}(4, x, F(y), \{F(y_i)/x_{i+1}\}) = \langle x_1, F(y_1), F(y_2), F(y_3) \rangle$.

We obtain the following result.

Theorem 5.5. *Let x be a NEG-variable and t the corresponding term in the tail of the clause. The Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of x is given by:*

$$\bar{x}_i = \varphi(x,i)(\theta_1 * \cdots * \theta_n) = \begin{cases} \varphi(x,1), & i = 1; \\ \tau(t,\theta_c,i-1), & 1 < i \leq n. \end{cases}$$

Proof. The proof is analogous to the proof of Theorem 5.4.

Example 5.4. Consider the recursive clause

```
P(x, z) ← P(F(z), G(y)).
```

We have the following classification and Reform series of the variables.

Variable	Category	Reform series
y	NONE	$\langle \mathtt{y}_1, \mathtt{y}_2, \mathtt{y}_3, \mathtt{y}_4 \rangle$
z	NEG	$\langle \mathtt{z}_1, \mathtt{G(y}_1\mathtt{)}, \mathtt{G(y}_2\mathtt{)}, \mathtt{G(y}_3\mathtt{)} \rangle$
x	NEG	$\langle \mathtt{x}_1, \mathtt{F(z}_1\mathtt{)}, \mathtt{F(G(y}_1\mathtt{))}, \mathtt{F(G(y}_2\mathtt{))} \rangle$

Next we show that NONE, POS and NEG allow classification of each variable in a recursive program.

Theorem 5.6. *Each variable in a recursive clause can be assigned to one of the categories NONE, POS, or NEG.*

Proof. Let x be any variable and θ_{i+1} be the characteristic mgu of the clause. We have three cases:

1. If neither x_i nor x_{i+1} is substituted for in θ_{i+1}, then x is a NONE-variable.
2. If there is a substitution element t/x_i in θ_{i+1}, then x is a POS-variable.
3. If there is a substitution element t/x_{i+1} in θ_{i+1}, then x is a NEG-variable. ∎

5.4. COROLLARIES FOR RECURRING SUBSTITUTIONS

We can identify certain special cases of POS and NEG when more direct and efficient expressions can be found for the Reform series. These special cases involve a recurrence in that a variable occurs in the term that we substitute for (a renaming of) the same variable.

The first such case that we identify is when a variable is substituted for a renaming of that same variable. We shall also identify two particulary simple special cases that are common in list programs. (Similar special cases can be identified for other cases of list recursion and, for example, integer recursion.)

INV-variables. A POS-variable x is an *INV-variable* if and only if

$$\varphi(x,i)\theta_{i+1} = \varphi(x,i+1), \qquad \text{for} \quad 1 \le i < n.$$

That is, x occurs in corresponding positions in the head and tail of the clause. Examples:

$$\texttt{P(x)} \leftarrow \texttt{P(x)} \qquad \text{and} \qquad \texttt{P(F(42,x))} \leftarrow \texttt{P(F(y,x))}.$$

It is straightforward to verify that the Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of an INV-variable x is given by:

$$\bar{x}_i = \varphi(x,n), \qquad \text{for} \quad 1 \le i \le n.$$

Example: $\mathrm{POS}(4, \texttt{x}, \texttt{x}, \{\mathtt{x_{i+1}}/\mathtt{x_i}\}) = \langle \mathtt{x_4}, \mathtt{x_4}, \mathtt{x_4}, \mathtt{x_4} \rangle$.

POS-LIST-variables. Let u be a term. A POS-variable x is a *POS-LIST-variable* if and only if

$$\varphi(x,i)\theta_{i+1} = \tau(u,\theta_c,i+1).\tau(x,\theta_c,i+1), \qquad \text{for} \quad 1 \le i < n.$$

That is, x occurs in the tail of the clause and there is a list $u.x$ in the corresponding position of the clause head. Examples:

$$\texttt{P(u.x)} \leftarrow \texttt{P(x)} \qquad \text{and} \qquad \texttt{P(v.u.x)} \leftarrow \texttt{P(w.x)}.$$

It is straightforward to verify that the Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of a POS-LIST-variable x is given by:

$$\bar{x}_i = \begin{cases} \varphi(x,n), & i = n; \\ \tau(u,\theta_c,i+1)...\tau(u,\theta_c,n).\varphi(x,n), & 1 \le i < n. \end{cases}$$

Example: $\mathrm{POS}(4, \texttt{x}, \texttt{u.x}, \{\mathtt{u_{i+1}.x_{i+1}}/\mathtt{x_i}\}) = \langle \mathtt{u_2.u_3.u_4.x_4},\ \mathtt{u_3.u_4.x_4},\ \mathtt{u_4.x_4},\ \mathtt{x_4} \rangle$.

NEG-LIST-variables. Let u be a term. A NEG-variable x is a *NEG-LIST-variable* if and only if

$$\varphi(x,i+1)\theta_{i+1} = \tau(u,\theta_c,i).\tau(x,\theta_c,i), \qquad \text{for} \quad 1 \le i < n.$$

That is, x occurs in the clause head and there is a list $u.x$ in the corresponding position of the tail of the clause. Examples:

$$\texttt{P(x)} \leftarrow \texttt{P(u.x)} \qquad \text{and} \qquad \texttt{P(w.x)} \leftarrow \texttt{P(v.u.x)}.$$

It is straightforward to verify that the Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of a NEG-LIST-variable x is given by:

$$\bar{x}_i = \begin{cases} \varphi(x,1), & i = 1; \\ \tau(u,\theta_c,i-1)...\tau(u,\theta_c,1).\varphi(x,1), & 1 < i \le n. \end{cases}$$

Example: $\mathrm{NEG}(4, \mathtt{x}, \mathtt{u.x}, \{\mathtt{u_i.x_i/x_{i+1}}\}) = \langle \mathtt{x_1},\ \mathtt{u_1.x_1},\ \mathtt{u_2.u_1.x_1},\ \mathtt{u_3.u_2.u_1.x_1}\rangle$.

Example 5.5. Consider again the clause:

```
Rev(u.x, y, z) ← Rev(x, u.y, z)
```

We have the following variable classification.

Variable	Variable class
u	NONE
x	POS (POS-LIST)
y	NEG (NEG-LIST)
z	POS (INV)

5.5. REMARKS ON THE CLASSIFICATION OF VARIABLES

1. The variables of a clause are classified by inspection of the characteristic mgu θ_c as follows. First, in order to avoid circularity, we check for substitution elements in θ_c of the form x/y, where both x and y are variables. For each such case, classify the variable occurring in the tail as POS and the variable occurring in the head as NONE. Then, we classify all other variables.

2. POS-LIST and NEG-LIST are just examples of conceivable special cases involving recursive data structures. Let us consider the general case where the recursion has a 'step' function ψ:

$$\mathtt{P(t)} \leftarrow \mathtt{P(x)}, \quad \text{where} \quad \psi(\mathtt{t}) = \mathtt{x};$$
$$\mathtt{P(x)} \leftarrow \mathtt{P(t)}, \quad \text{where} \quad \psi(\mathtt{t}) = \mathtt{x}.$$

We can identify a special case for each sufficiently uncomplicated* function ψ. POS-LIST and NEG-LIST represent the simple but common case when ψ is the 'tail' function:

```
P(u.x) ← P(x);
P(x) ← P(u.x).
```

For integer recursion we may, for example, consider the case $\psi(\mathtt{y}) = \mathtt{y} \oplus \mathtt{k}$, for some integer function $\oplus$ (addition, multiplication, etc.) and integer $\mathtt{k}$:

$$\mathtt{P(y)} \leftarrow \mathtt{x} = \mathtt{y} \oplus \mathtt{k} \wedge \mathtt{P(x)};$$
$$\mathtt{P(x)} \leftarrow \mathtt{x} = \mathtt{y} \oplus \mathtt{k} \wedge \mathtt{P(y)}.$$

* As an example of a 'step' function ψ which is not sufficiently uncomplicated, consider $\psi(x) = \begin{cases} x/2, & x \text{ even}; \\ 3x+1, & x \text{ odd}. \end{cases}$

5.6. EXISTENCE OF A NTH REFORMENT

We have, so far, assumed that the nth reforment of the clause exists. When this is the case we can, as is described above, express the corresponding Reform series in a parameterized way. However, it might be the case that the nth reforment exists—but not the $(n+1)$th reforment. Consider, for example, the following clause.

$$\mathtt{P(u.v,w,x,y,z,x)} \leftarrow \mathtt{P(v,A,B,w,y,z)} \tag{12}$$

The fourth reforment of the clause is:

$$\mathtt{P(u_1.u_2.u_3.u_4.v_4,B,x_1,B,B,x_1)} \leftarrow \mathtt{P(v_4,A,B,A,A,A)}$$

It is easily verified that no fifth reforment of the clause exists.

It would be satisfying if by inspecting the characteristic mgu, we could decide that the Reform transformation will fail for some $n \geq 2$ (we infer the case of failure for $n = 1$ from the fact that the characteristic mgu does not then exist). Fortunately, it turns out to be straightforward to deduce the failure of the transformation from the characteristic mgu.

We let θ_c be the characteristic mgu and $x^1, \ldots, x^m$ all program variables of a clause. Let s and t be non-unifiable terms. We say that a *failure chain* in the characteristic mgu is a chain of substitutions

$$s/x^1,\ x^1/x^2, \ldots, x^{k-1}/x^k, t/x^k, \qquad k \geq 1$$

obtained by dropping the subscripts of the variables in θ_c. The characteristic mgu of the clause (12) is:

$$\theta_c = \{\mathtt{u_{i+1}.v_{i+1}/v_i},\ \mathtt{A/w_{i+1}},\ \mathtt{y_{i+1}/w_i},\ \mathtt{z_{i+1}/y_i},\ \mathtt{B/z_i},\ \mathtt{B/x_{i+1}}\}.$$

We see that it contains the failure chain $A/w,\ w/y,\ y/z,\ B/z$.

We conjecture that the Reform transformation will fail for some $n \geq 2$ if there is a failure chain in the characteristic mgu. We can deduce which transformation step will fail by actually carrying out, at compile-time, the Reform transformation to the point of failure.

5.7. MULTI-TYPED VARIABLES

A variable may belong to more than one category. Suppose, for example, that a variable x belongs to both POS and NEG. We have then two expressions for the Reforms series $\bar{x}$ of x: $\bar{x}_+$ and $\bar{x}_-$, say. The fact that x belongs to both POS and NEG implies that neither $\bar{x}_+$ nor $\bar{x}_-$ gives a complete description of $\bar{x}$. The complete description is obtained by unifying $\bar{x}_+$ and $\bar{x}_-$.

Example 5.6. Consider the following recursive clause.

$$\mathtt{P(u.w,x,F(y))} \leftarrow \mathtt{P(w,F(z),x)}.$$

The characteristic mgu is $\theta_{i+1} = \{\mathtt{u_{i+1}.w_{i+1}/w_i}, \mathtt{F(z_i)/x_{i+1}}, \mathtt{F(y_{i+1})/x_i}\}$. Hence `x` is both a POS-variable and a NEG-variable. We have the following Reform series for POS and NEG.

$$\bar{\mathtt{x}}_+ = \langle \mathtt{F(y_2)}, \mathtt{F(y_3)}, \ldots, \mathtt{F(y_n)}, \mathtt{x_n} \rangle, \quad \bar{\mathtt{x}}_- = \langle \mathtt{x_1}, \mathtt{F(z_1)}, \mathtt{F(z_2)}, \ldots, \mathtt{F(z_{n\text{-}1})} \rangle.$$

The Reform series of `x` is obtained by unifying $\bar{\mathtt{x}}_+$ and $\bar{\mathtt{x}}_-$:

$$\bar{\mathtt{x}} = \langle \mathtt{F(y_2)}, \mathtt{F(y_3)}, \ldots, \mathtt{F(y_n)}, \mathtt{F(z_{n\text{-}1})} \rangle.$$

Example 5.7. Consider a slight variation of the preceding example:

$$\mathtt{P(u.w, x, F(y))} \leftarrow \mathtt{P(w, G(z), x)}.$$

As before, `x` is both a POS-variable and a NEG-variable. The Reform series for POS and NEG are now:

$$\bar{\mathtt{x}}_+ = \langle \mathtt{F(y_2)}, \mathtt{F(y_3)}, \ldots, \mathtt{F(y_n)}, \mathtt{x_n} \rangle, \quad \bar{\mathtt{x}}_- = \langle \mathtt{x_1}, \mathtt{G(z_1)}, \mathtt{G(z_2)}, \ldots, \mathtt{G(z_{n\text{-}1})} \rangle.$$

We note that $\bar{\mathtt{x}}_+$ and $\bar{\mathtt{x}}_-$ can be unified only if $n = 1$ or $n = 2$. This corresponds to the easily verified fact that the nth reforment exists only for $n = 1$ and $n = 2$.

CHAPTER 6

LINEAR RECURSION

In this chapter we shall introduce our compilation method, which is based on the analysis of chapter 5, and apply it to linear recursion over list structures. We compile recursion over lists to iteration over vectors. We exploit parallelism by taking up techniques developed for parallelization of Fortran-like programs.

We shall assume, for the moment, that the list program to be compiled satisfies the following recursion schema:

$$\begin{aligned}&P(\emptyset)\\&P(x.y) \leftarrow S \wedge P(y)\end{aligned}$$

Fortunately, we can handle a much larger class of programs with the compilation method. We shall see, in chapter 8, how to transform this larger class of programs into a form that satisfies the recursion schema.

We will only describe compilation of recursive clauses. Issues such as clause selection and failure handling (for example, management of backtracking and related subjects such as trailing of variables) are not dealt with. In fact, it would be possible to borrow many solutions of the implementation problems not dealt with here from, say, Warren's abstract Prolog machine (1983).

6.1. REFORMING COMPILATION

We have seen that programs resulting from Reform transformation are highly parallel. This observation might be of little value, however, unless we can obtain the transformed program in an efficient way. One approach to this problem would be speed up each step in the Reform computation by compilation. We shall, however, explore another possibility.

Reforming logic and reforming control. Kowalski's equation 'Algorithm = Logic + Control' implies that we can improve a program by changing either its logic component or its control component. With Reform we improve the original algorithm by transforming the logic component; the reformed program is then run with the standard control strategy. Our compilation method aims at obtaining the same algorithm as Reform by changing the control component, that is by 'reforming' the control while keeping the logic component unchanged.

This can be depicted as follows.

$$\begin{array}{ll} A\ = L + C & \text{(original logic, standard control)} \\ A' = L' + C & \text{(reformed logic, standard control)} \\ A' = L + C' & \text{(original logic, reformed control)} \end{array}$$

Thus, our idea is to change the control strategy of the original program in order to obtain the same operational behaviour as the program derived by Reform transformation. The crucial difference between the two ways of arriving at A' is that, in general, L' must be derived at run-time whereas C' is obtainable at compile-time.

The reformed control. The computation of a recursive program is made up of two kinds of activities: head unification and body call computation. Let H denote an activity of the former kind and B an activity of the latter kind. We can then describe, in an informal way, the structure of the computation using the original logic under the standard control regime as:

$$H\ B\ H\ B\ H\ B\ \ldots$$

That is, head unification and body call computation proceed alternatingly. We shall, in contrast, let the program compute according to the following structure:

$$H^*\ B^*$$

where H^* and B^* indicate iteration of the activities H and B. Now let us relate this to the computation of an nth reforment R^n:

- H^* corresponds to the unification resulting from invoking R^n.
- B^* corresponds to computation of the body of R^n.

Parameterized encoding of the nth reforment. We arrive at programs for H^* and B^* by compiling the recursive clause of a program to a parameterized encoding of its nth reforment. Let us motivate this encoding by an example. Recall the recursive clause the `LessAll` program:

$$\texttt{LessAll(x.y, z)} \leftarrow \texttt{z < x} \wedge \texttt{LessAll(y, z)}.$$

The nth reforment of the clause is:

$$\texttt{LessAll}(\texttt{x}_1 \ldots \texttt{x}_\texttt{n}.\texttt{y}_\texttt{n}, \texttt{z}_\texttt{n}) \leftarrow \texttt{z}_\texttt{n} < \texttt{x}_1 \wedge \cdots \wedge \texttt{z}_\texttt{n} < \texttt{x}_\texttt{n} \wedge \texttt{LessAll}(\texttt{y}_\texttt{n}, \texttt{z}_\texttt{n}).$$

We can also write the nth reforment as:

$$\texttt{LessAll}(\bar{\texttt{x}}_1.\bar{\texttt{y}}_1, \bar{\texttt{z}}_1) \leftarrow \bar{\texttt{z}}_1 < \bar{\texttt{x}}_1 \wedge \cdots \wedge \bar{\texttt{z}}_\texttt{n} < \bar{\texttt{x}}_\texttt{n} \wedge \texttt{LessAll}(\bar{\texttt{y}}_\texttt{n}, \bar{\texttt{z}}_\texttt{n}).$$

Now we see that:

- the 1st elements of the Reform series of `x`, `y` and `z` are used in place of the program variables in the head;
- the ith elements of the Reform series of `z` and `x` are used in place of the program variables in the ith instance of the residual body ($1 \leq i \leq n$);
- the nth elements of the Reform series of `y` and `z` are used in place of the program variables in the tail.

We can thus describe the computation of a recursive clause $H \leftarrow \Psi \wedge T$ and an invoking call G as follows.

1. Unify the head H with the call G, using the 1st element of the Reform series of each variable in H.
2. Compute the residual body Ψ n times, using the ith element of the Reform series of each variable in Ψ in the ith computation of Ψ ($1 \leq i \leq n$).
3. Compute the tail T, using the nth element of the Reform series of each variable in T.

Let $H(i)$, $\Psi(i)$ and $T(i)$ denote H, Ψ, and T, respectively, with all variables replaced by the ith elements of the corresponding Reform series. We can then, finally, give the parameterized encoding of the nth reforment schematically as:

```
unify $H(1)$ with $G$
for i=1 to n do call $\Psi(i)$
call $T(n)$
```

In order to compute the Reform series we add compiled versions of the functions POS, NEG, etc., defined in chapter 5, to the program when neccessary.

Example 6.1. Consider the following program for checking whether all elements of a list are different. We assume that `NotIn(x,y)` holds if `x` is not a member of the list `y`.

```
DiffAll(∅).
DiffAll(x.y) ← NotIn(x,y) ∧ DiffAll(y).
```

The Reform series of the NONE-variable `x` and the POS-LIST-variable `y` are:

$$\bar{\mathtt{x}} = \langle \mathtt{x}_1, \mathtt{x}_2, \mathtt{x}_3, \mathtt{x}_4 \rangle \qquad \text{and} \qquad \bar{\mathtt{y}} = \langle \mathtt{x}_2.\mathtt{x}_3.\mathtt{x}_4.\mathtt{y}_4, \mathtt{x}_3.\mathtt{x}_4.\mathtt{y}_4, \mathtt{x}_4.\mathtt{y}_4, \mathtt{y}_4 \rangle.$$

Let us suppose that the program is invoked by the goal

```
← DiffAll(1.2.3.4.∅).
```

The computation then proceeds as follows:

Head unification:	`1.2.3.4.∅ = x`$_1$`.x`$_2$`.x`$_3$`.x`$_4$`.y`$_4$
First iteration of the loop:	`NotIn(1,2.3.4.∅)`
Second iteration of the loop:	`NotIn(2,3.4.∅)`
Third iteration of the loop:	`NotIn(3,4.∅)`
Fourth iteration of the loop:	`NotIn(4,∅)`
Recursive call:	`DiffAll(∅)`

6.2. REPRESENTATION OF VARIABLES

We turn now to the question of how to represent variables in the compiled program. The idea is that a variable is represented at run-time by its Reform series implemented as a vector. On a parallel machine we assume that the ith element of the Reform series of a variable is stored on processor i, for $1 \leq i \leq n$. (The reader might notice the analogy with vectorization of scalar variables in Fortran-like programs.)

POS, NEG. The Reform series of a POS-variable x is $\langle \bar{t}_2, \ldots, \bar{t}_n, x_n \rangle$, where $\bar{t}_i$ is the ith element of the Reform series of some term t. This implies that we need not explicitly represent the first $n-1$ elements of the Reform series of x. References to these elements can be replaced by references to $\bar{t}_2, \ldots, \bar{t}_n$.

Similarly, the Reform series of a NEG-variable x is $\langle x_1, \bar{t}_1, \ldots, \bar{t}_{n-1} \rangle$. References to the $n-1$ last elements of the Reform series of x can be replaced by references to $\bar{t}_1, \ldots, \bar{t}_{n-1}$. Thus we represent the Reform series of a POS- or NEG-variable x as a scalar variable in the compiled program.

INV. An INV-variable x has the Reform series $\langle x_n, \ldots, x_n \rangle$. It can be represented by a scalar variable. (An optimization is possible if x occurs in the residual body and no attempt to bind x—or variables in a term bound to x in head unification—is made: we might then allocate a copy of x on each processor in order to improve the parallelization.)

NONE, POS-LIST, NEG-LIST. The Reform series of a variable x classified as POS-LIST or NEG-LIST is represented by a vector if x occurs in the residual body; otherwise it is represented by a scalar holding the nth element of x's Reform series. The Reform series of a NONE-variable is represented by a vector.

Example 6.2. Consider the recursive clause of the list reversal program:

```
Nrev(u.x,y) ← Nrev(x,z) ∧ Append(z,u.∅,y).
```

Let us see which Reform series are represented as scalars and which are represented as vectors:

Variable	Variable class	Representation
`u`	NONE	vector
`x`	POS-LIST	scalar
`z`	POS	scalar
`y`	NONE	vector

6.3. ITERATION OF HEAD UNIFICATION

Recall the computation activities H^* and B^* discussed in section 6.1. The subject of this and the next two sections is H^*.

Intuitively, H^* corresponds to the iteration of the unification work H performed at each recursion level with a traditional logic programming system. The program for H^* is a parameterized encoding of the head of the nth reforment. We divide H^* into two phases:

1. In the first phase we take in all input data from the invoking call. This is done by a modified version of traditional head unification.
2. In the second phase we compute the variable bindings made in Reform transformation. This is done by the functions defined in chapter 5 for computing Reform series.

In the following two sections we describe these two phases of H^*.

6.4. COMPILATION OF THE CLAUSE HEAD

We rely mostly on ideas developed by Warren (1977, 1983) for compilation of the clause head. However, we diverge from Warren's scheme in the handling of recursive data structures (that is, lists in the case of linear recursion). We shall refer to the arguments of the invoking call as `A1`, `A2`, etc.

Warren's unification scheme. We start by reviewing Warren's scheme for unification of terms in the invoking call with terms in the clause head. Let us first consider unification of a call argument `Ai` with a variable x in the clause head. We distinguish between the first and subsequent occurrences of x in encoding this unification:

First occurrence of x: `x := Ai`
Subsequent occurrences of x: `unify(x,Ai)`

Next we consider unification of a call argument `Ai` with a compound term in the clause head. We restrict the discussion to unification of lists, other compound terms being compiled analogously. So, assume that the term in the clause head is a list $x.y$ whose tail is not a POS-LIST-variable (we will treat lists with POS-LIST-variable tails specially). If `Ai` is a non-empty list, then we unify its head with x and its tail with y. If `Ai` is an unbound variable we create a list *x.y* and unify it with the variable in the call. Following Warren we say that the unification proceeds in *read mode* in the former case, and in *write mode* in the latter case. We assume that these are the first occurrences of x and y (otherwise the code is modified in accordance with the discussion above):

Read mode: `x := head(Ai); y := tail(Ai)`
Write mode: `x := undef; y := undef; unify(Ai,cons(x,y))`

Here we use the special symbol `undef` for a new unbound variable.

List unification. We diverge from Warren's scheme for head unification in the case of lists whose tails are POS-LIST variables. The recursion list in the clause head is always such a list, but other lists can also have tails that are POS-LIST variables. For example, the first and third terms in the head of the `Append` program are such lists:

```
Append(u.x, y, u.z) ← Append(x, y, z).
```

We discuss these two kinds of lists in turn.

The recursion list. Assume that `Ai` is the recursion list in the call and that *x.y* is the recursion list in the head. We compile the unification of `Ai` with *x.y* to the statement `n := traverse(Ai,x,y)` which has the following effects:

- `n` is assigned the number of elements in the list `Ai`.
- The elements of `Ai` are assigned to the vector `x[1:n]`.
- The successive tails of `Ai` are assigned to the vector `y[1:n]` in the case that y occurs in the residual body of the clause; otherwise the last tail of `Ai` is assigned to the scalar variable `y`.

Barklund (1990) defines a parallel algorithm for this operation. The algorithm runs in $O(\log n)$ time for typical lists.

Other lists with POS-LIST tails. Two additional issues that must be dealt with in the case when a list whose tail is a POS-LIST variable is not the recursion list. This is, for example, the case with the list `u.z` in the `Append` program above.

First, the list in the call might be only partly constructed. Say, for example, that the list has five elements and ends with a variable, while the recursion list has 20 elements so that $n = 20$. Then the head unification program for this list must match ('read mode') the five elements and then complete ('write mode') the list with 15 more elements.

Second, it is possible that the list does not represent the first occurrence of the list-head variable. For example, the list `u.z` does not represent the first occurrence of the variable `u` in the `Append` program. In this case `traverse` is modified to *unify* the list elements. Although probably less common, the same modification might be required for the list-tail variable.

Let y be a POS-LIST-variable and consider unification of $x.y$ with a list or variable `Ai` in the call. We assume that this is not the first occurrence of x and that the operation `traverse*` is a modification of `traverse` that captures this difference.

```
k := traverse*(Ai,x,y)
if k<n then
   y[n] := undef
   for* i=k to n-1 do y[i] := cons(x[i+1],nil) end
   for* i=k to n-1 do addr(tail(y[i])) := y[i+1] end
end
```

Both loops in this code are parallel. We assume the following behaviour of the operation `k := traverse(Ai,x,y)`, and its variations, when `Ai` is an unbound variable: `Ai` is bound to a new list cell `cons(x[1],y[1])` and `k` is set to `1`. In this case the program above constructs a list of $n-1$ elements with a very high degree of parallelism.

Example 6.3. Consider again the recursive clause of the `Append` program:

$$\texttt{Append(u.x, y, u.z)} \leftarrow \texttt{Append(x, y, z).}$$

The variable classification and representation is:

Variable	Variable class	Representation
u	NONE	vector
x	POS-LIST	scalar
y	INV	scalar
z	POS-LIST	scalar

We assume that the first argument is the recursion list. The compiled clause head is then:

```
n := traverse(A1,u,x)
y := A2
k := traverse*(A3,u,z)
if k<n then
   z[n] := undef
   for* i=k to n-1 do z[i] := cons(u[i+1],nil) end
   for* i=k to n-1 do addr(tail(z[i])) := z[i+1] end
end
```

Here `traverse*` is the modified version of `traverse` mentioned earlier.

6.5. COMPUTING REFORM SERIES

In order to implement our control scheme of first computing *all* head unification, and then computing the clause body in a bounded loop, we must have access to the bindings obtained by Reform transformation: that is, the Reform series of the variables.

It turns out that no extra computation, in addition to head unification as described above, is required in order to obtain the Reform series of certain classes of variables. Let us consider these cases first:

NEG, INV. A NEG-variable or INV-variable always occurs in the head of the clause. Furthermore, it suffices to represent the first element of its Reform series as discussed above. Therefore, no more elements of its Reform series other than the one created by head unification needs to be created.

POS-LIST. All elements of a POS-LIST-variable's Reform series are created during head unification, as is discussed in the previous section.

Additional code is required for computing the Reform series of variables classified as POS or NEG-LIST. This might also be the case for NONE-variables. Let us examine these variable classes in turn:

POS. A POS-variable x does not occur in the clause-head. We found above, however, that only the last element of its Reform series has to be explictly represented. It is created by `x := undef`.

NONE. The code for computing the Reform series of a NONE-variable x depends on whether or not the variable occurs in the clause head, that is whether or not head unification creates `x[1]`. If it does so, the Reform series is constructed by:

```
if n>1 then
   for* i=2 to n do x[i] := undef end
end
```

If head unification does not create `x[1]`, the Reform series is constructed by:

```
for* i=1 to n do x[i] := undef end
```

These loops are parallel. One important exception to this discussion of Reform series for NONE-variables is the case when such a variable occurs as the head of a list whose tail is a POS-LIST-variable (the recursion list, for example). In this case the Reform series of the NONE-variable is known to have been generated in head unification by the `traverse` operation.

NEG-LIST. Let $x.y$ be a list where y is a NEG-LIST-variable. Then `y[1]` is created by head unification and we can also assume that the Reform series `x[1]`,...,`x[n]` of x is available. The following code creates the remaining parts of the Reform series of y:

```
if n>1 then
   for* i=2 to n do y[i] := cons(x[i-1],nil) end
   for* i=2 to n do addr(tail(y[i])) := y[i-1] end
end
```

These loops are parallel.

It should be noted that it might enhance efficiency to create the Reform series of NONE- and NEG-LIST-variables as close to their use (as arguments to body calls) as possible. In this way we minimize unnecessary work in the case when some body call fail. However, in the case when recursion parallelism cannot be exploited for the body calls, we might increase parallelism by creating the Reform series outside the sequential body loop, since these operations can be done in parallel. The single POS-variable should, of course, be created in connection with the single recursive call.

6.6. ITERATION OF THE CLAUSE BODY COMPUTATION

We turn next to the other computation activity discussed in section 6.1: B^*. Intuitively, B^* corresponds to the iteration of the the clause body computation B performed at each recursion level with a traditional logic programming system. The program for B^* is a parameterized encoding of the body of the nth reforment. B^* is divided into two parts:

1. a single recursive call to the program, and
2. iteration of the residual body calls.

The recursive call. The single recursive call of the recursive clause is known to match the base clause. However, it may also match the recursive call, for example if the recursion list ends with an unbound variable. The appropriate arguments of the call are found by replacing each program variable in the tail by the nth element of its Reform series.

Example 6.4. The tail of the `DiffAll` program given above is `DiffAll(y)`, where `y` is a POS-LIST-variable. The code for the recursive call is:

```
call DiffAll(y[n])
```

The residual body calls. The residual body of the recursive clause is compiled to a `for`-loop with loop index $i = 1, \dots, n$. Recursion parallelism is implemented by parallelization of this loop. The loop index serves as an index to the Reform series of the variables occurring in the residual body.

When the loop cannot be parallelized, we employ the standard left-to-right control rule used by, for example, Prolog, when computing the n instances of the residual body. This means that the body calls that appear textually to the left of the tail in the clause are computed in a loop from 1 to n, and the calls that appear to the right of the tail in a loop from n down to 1.

Let us state this more precisely. Consider a recursive clause $P \leftarrow L \wedge T \wedge R$, where T is the tail. Let $L(i)$, $T(i)$ and $R(i)$ denote L, T and R, respectively, with all variables replaced by the ith elements of the corresponding Reform series. The compiled program for the clause body will then be:

```
for i=1 to n do call $L(i)$ end
call $T(n)$
for i=n downto 1 do call $R(i)$ end
```

Example 6.5. Recall the recursive clause of the `Nrev` program:

$$\texttt{Nrev(u.x, y)} \leftarrow \texttt{Nrev(x, z)} \wedge \texttt{Append(z, u.}\emptyset\texttt{, y)}.$$

The program for the clause body is:

```
call Nrev(x,z)
call Append(z,cons(u[n],nil),y[n])
for i=n-1 downto 1 do
   call Append(y[i+1],cons(u[i],nil),y[i])
end
```

(We compute the `Append` call for `i=n` before the loop in order to avoid a conditional expression in the loop body.) This loop is sequential due to data dependencies. However, the data dependencies can be broken by program transformation, as is described in chapter 4.

6.7. DISTRIBUTION OF DATA

We cannot, in general, assume that each processor in a parallel machine has available all data needed for its operation. It will occasionally need data that is located in the memory of another processor. If the overhead from this communication is too large, it may effectively impede speedups from parallelization.

Assume that processor i needs some data located at processor j. The standard approach in parallel logic programming systems has been to let processor i send a request message to processor j. The data can then be sent from processor j to processor i after the request message has arrived.

Our compilation method allows communication where each processor knows in advance when and where to communicate its data. This is more efficient than the standard approach, for two reasons. First, a lot of communication is saved since the receivers do not have to request the data. Second, it gives the compiler a chance to allocate processors in a way that minimizes communication length.

We shall describe how the compiler can handle communications that result from variables of types INV, POS, and NEG that occur in the residual body of a clause. We assume that the n processors are numbered from 1 to n.

INV-variables. Suppose that x is an INV-variable:

$$P(..., x, ...) \leftarrow Q(..., x, ...) \wedge P(..., x, ...)$$

We know that x is initialized in the head unification. Let us assume that processor 1 creates x. We know, furthermore, that each of the n calls to Q in the body will need the value of x. Hence the compiler can emit code for sending the value of x from processor 1 to processors $2, \ldots, n$.

POS-variables. Suppose that x is a POS-variable and t the corresponding term in the head:

$$P(..., t, ...) \leftarrow Q(..., x, ...) \wedge P(..., x, ...)$$

We know that the value of x_i is t_{i+1}, for $1 \leq i < n$. Hence the compiler can emit code for sending the t instance on processor i to processor $i-1$, for $2 \leq i \leq n$.

NEG-**variables.** Suppose that x is a NEG-variable and t the corresponding term in the tail:

$$P(..., x, ...) \leftarrow Q(..., x, ...) \wedge P(..., t, ...)$$

We know that the value of x_i is t_{i-1}, for $2 \leq i \leq n$. Hence the compiler can emit code for sending the t instance on processor i to processor $i+1$, for $1 \leq i < n$.

6.8. NONDETERMINISTIC BODY CALLS

We might ask the question: how does one deal with nondeterminism in a system that exploits recursion parallelism? Let us propose one possible answer. Consider a recursive clause with k goals in the residual body:

$$H \leftarrow B_1 \wedge \cdots \wedge B_k \wedge T.$$

The nth reforment of this clause is on the form:

$$H \leftarrow B_{11} \wedge \cdots \wedge B_{k1} \wedge \cdots \wedge B_{1n} \wedge \cdots \wedge B_{kn} \wedge T.$$

As in chapter 3 we depict the residual body of the nth reforment as a matrix of n rows and k columns:

$$\begin{pmatrix} B_{11} & \ldots & B_{k1} \\ \vdots & \ddots & \vdots \\ B_{1n} & \ldots & B_{kn} \end{pmatrix}$$

The k calls within each row are computed sequentially with the left-to-right computation rule of Prolog, or some co-routining variation thereof. The n rows are computed in parallel if possible. More specifically, the rows are computed in parallel unless there are data dependencies that prevent parallelism. Nondeterminism within each row and between rows is implemented with chronological backtracking.

6.9. LINEAR INTEGER RECURSION

The compilation method developed here for linear list recursion is also applicable to linear integer recursion. The theory for integer recursion follows trivially from the list theory, since integers represented with the successor function correspond to lists with nonsignificant elements. For example, the integer 3 corresponds to a list with three elements.

Some additional machinery is needed when using Prolog's representation of integers as constants. With this representation, unification is not the operation for manipulating integers. Thus we need, in addition to examine the characteristic mgu, to search the clause body for primitive operations that manipulate integers (in Prolog, for example, we inspect calls to `is/2`). We shall refrain from going into detail here.

6.10. EXAMPLES

Let us now examine a few complete examples of compiled clauses.

Example 6.6. Our first example is the insertion sort program:

`Sort(u.x, y)` $\leftarrow$ `Sort(x, z)` $\wedge$ `Insert(z, u, y)`.

The compiled code for the recursive clause is:

```
u,y : vector
x,z,n,i : scalar
n := traverse(A1,u,x)
y[1] := A2
for* i=2 to n do y[i] := undef end
z := undef
call Sort(x,z)
call Insert(z,u[n],y[n])
for i=n-1 downto 1 do
   call Insert(y[i+1],u[i],y[i])
end
```

The `traverse` operation and the first loop are parallel. The loop with calls to `Insert` exhibits data dependencies that prevent parallelization. However, each call to `Insert` can exploit parallelism (cf. chapter 8). Moreover, the technique for breaking data dependencies described in chapter 4 is applicable to this program.

Example 6.7. Let us next consider the recursive clause of the `LessAll` program:

`LessAll(x.y, z)` $\leftarrow$ `z < x` $\wedge$ `LessAll(y, z)`.

The compiled code is:

```
x,z : vector
y,n,i : scalar
n := traverse(A1,x,y)
z[1] := A2
distribute z[1] to z[2],...,z[n]
for* i=1 to n do z[i] < x[i] end
call LessAll(y,z[n])
```

The loop in this program is parallel. Here we have assumed that the compiler represents the INV-variable `z` as a vector, since the value is needed on all n processors (cf. section 6.7).

6.11. CONCLUSION

Efficiency of parallelization. The following substantial parts of the computation are reduced to `for`-loops with our compilation method: construction of output lists ('write mode' unification), construction of Reform series, and computation of the residual body calls. Consequently, our method leads to a situation where we can apply standard parallelization techniques, originally developed for Fortran-like programs, to these parts of the compiled logic program.

We have treated the operations for traversal of input lists ('read mode' unification), viz. `traverse` and its variations, as a black box. We noted, however, that efficient parallel methods for such operations do exist.

Efficiency of implementation. It appears that this compilation method allows more efficient implementation on parallel machines than do traditional AND-parallel systems. Operations such as distribution of data, task scheduling, and load balancing can be planned at compile-time. This is not the case in a traditional system where, therefore, these operations are comparatively inefficient. Let us briefly discuss each of these issues.

A processor should know in advance when and where to communicate its data. This clearly allows more efficient communication than interactive requests from other processors. For example, in the `Lessall` program above we know at compile-time that `z` should be communicated to $n-1$ processors. The code emitted by the compiler for this task is much more efficient than the handling of $n-1$ interactive data requests from other processors.

Most concurrent logic programming languages are plagued by considerable overhead in dynamic process scheduling. It may be that this inefficiency is inevitable in languages designed for programming communicating processes, since such languages handle synchronization and process suspension at the langauage level. The loop parallelization approach is not concerned with communicating processes and has no overhead for task scheduling: all iterations of a parallel loop are computed simultaneously and only one parallel loop is computed at any one time.

The problem of load balancing has a straightforward approximative solution with the loop parallelization technique: compute one iteration of the parallel loop on each processor or, if there are fewer processors than iterations of the loop, compute every kth iteration on the same processor (where k is the number of available processors). In contrast, load balancing is a difficult problem, which might impose severe run-time overhead, on traditional AND-parallel systems.

Sequential speedup. We anticipate that the compilation method may accelarate logic programs on sequential machines. The reason for this is, of course, that our method does not use recursion as a control structure. Let us review what is involved in a recursive procedure call in a traditional system:

1. The arguments of the call are set up.
2. The appropriate clause of the recursive procedure is selected.
3. A binding environment may be allocated.

These steps are replaced by a decrement-and-jump-if-not-zero operation in our method.

Tail recursion. Our method generalizes and improves the notion of tail-recursion optimization. The idea behind tail-recursion optimization is that certain instances of recursion can be implemented as iteration. This is the case if the recursive call is the last call in the procedure body (and, in the case of logic programs, if the computation is deterministic). Our compilation method implements recursion as iteration even if the recursive call is not the last call in the procedure body. Moreover, it implements recursion as *bounded* iteration. Therefore, the iterations can be parallelized.

New applications. Recursive programs that perform simple and fast operations at each recursion level are often not worth running in parallel on traditional AND-parallel systems—the overhead in spawning separate processes for the simple operations is too large. Our method's low overhead and high parallelization factor for this type of programs opens up new applications for parallel logic programming. For example, numerical computation and every-day data processing often involve simple operations on large volumes of data.

CHAPTER 7

NONLINEAR RECURSION

Up to this point the discussion has been focused on linear recursion. We shall now turn to nonlinear recursion and, more specifically, to binary tree programs. We compile recursion over binary trees to iteration over vectors. Not surprisingly, it turns out that the compilation theory will be slightly more complicated than was the case for the linear recursion.

We shall follow the methodology used for linear recursion and assume that the binary tree programs satisfy a restricted recursion schema. The recursion schema for binary tree programs is:

$$\begin{aligned}&P(\emptyset)\\&P(T(u,x,y)) \leftarrow S \wedge P(x) \wedge P(y)\end{aligned}$$

We refer to $P(x)$ as the *l-tail* and to $P(y)$ as the *r-tail* of the recursive clause.

The recursion schema does not cater for programs that traverse only part of the recursion tree. Fortunately, a large class of such programs can be straightforwardly transformed to the restricted form. The transformation is discussed in chapter 8.

7.1. REFORM

In the case of linear recursion the nth reforment is uniquely determined by the size n of the input data (the length of a list or the magnitude of an integer). For nonlinear tree programs, things are more complicated: there are many trees, having different shapes with n nodes, and hence no unique nth reforment for each n.

Example 7.1. Consider the program for inorder traversal of a binary tree:

```
Inorder(∅, ∅).
Inorder(T(u, x, y), z) ←
    Inorder(x, w) ∧ Inorder(y, v) ∧ Append(w, u.v, z).
```

Let us assume that this program is invoked by the following call for which $n = 4$:

```
← Inorder(T(A, T(B, ∅, ∅), T(C, T(D, ∅, ∅), ∅)), s).
```

The 4th reforment corresponding to this call is:

```
Inorder(T(u1,T(u2,x2,y2),T(u3,T(u4,x4,y4),y3)),z1) ←
    Inorder(x2,w2) ∧ Inorder(y2,v2) ∧ Inorder(y3,v3)
    Inorder(x4,w4) ∧ Inorder(y4,v4) ∧
    Append(w4,u4.v4,z4) ∧ Append(z4,u3.v3,z3) ∧
    Append(w2,u2.v2,z2) ∧ Append(z2,u1.z3,z1).
```

The 4th reforment of another call might be quite different, due to the different shape of its recursion tree.

Thus, in order to derive the nth reforment we must know both n and the structure of the recursion tree in the invoking call. The Reform transformation is controlled by this structure: only tail literals corresponding to nonempty subtrees may be selected in each reformation.

We need not develop the runtime details of the Reform procedure here, since we are interested in compilation. It suffice to note that we will obtain two sequences

$$\theta_1, \ldots, \theta_k, \qquad \text{and} \qquad \gamma_1, \ldots, \gamma_m, \qquad \text{where} \quad k + m = n + 1$$

of mgu:s in a Reform transformation (by definition $\theta_1 = \epsilon$ and $\gamma_1 = \epsilon$). The mgu:s $\theta_1, \ldots, \theta_k$ are obtained when resolving tail literals corresponding to nonempty *left* subtrees. Analogously, the mgu:s $\gamma_1, \ldots, \gamma_m$ are obtained when resolving tail literals corresponding to nonempty *right* subtrees. We use the skeleton representation of reforments introduced in chapter 5.

Example 7.2. In Example 7.1 we have:

$$\theta_2 = \{\mathtt{T(u_2,x_2,y_2)/x_1, z_2/w_1}\} \qquad \gamma_2 = \{\mathtt{T(u_3,x_3,y_3)/y_1, z_3/v_1}\}$$
$$\theta_3 = \{\mathtt{T(u_4,x_4,y_4)/x_3, z_4/w_3}\}$$

7.2. LINEAR REPRESENTATION OF TREE STRUCTURES

The idea to compile recursion over binary trees to iteration over vectors implies a linear representation of tree structures. This linear representation will be derived when the recursive tree program is invoked (as a side-effect of head unification).

We shall represent the structure of the recursion tree as its preorder traversal together with some additional structural information. In the interest of efficiency of the compiled program we need to impose two conditions on this structural information:

1. It must be efficiently obtainable while traversing the tree in preorder so that no extra traversal of the tree is needed.

2. It must allow efficient use. For example, it must be a fast operation to determine whether the left or right subtree of a node is empty.

We shall employ three tables of size n as structural information.

Structure tables. Let $u_1, \ldots, u_n$ be the preorder traversal of the recursion tree of the invoking call. The structure tables l, r and q are then set up as follows:

$l[i] = 1$	if u_i has a nonempty left subtree;	$l[i] = 0$	otherwise.
$r[i] = j$	if u_j is the right son of u_i;	$r[i] = 0$	if u_i has no right son.
$q[i] = 1$	if u_i is the left son of a node;	$q[i] = 0$	otherwise.

Example 7.3. A binary tree with eight nodes and its structure tables:

i	$l[i]$	$r[i]$	$q[i]$
1	1	5	0
2	1	4	1
3	0	0	1
4	0	0	0
5	1	8	0
6	0	7	1
7	0	0	0
8	0	0	0

7.3. CLASSIFICATION OF VARIABLES

In this section we classify variables according to the structure of their Reform series. For each type of variable we give a function that defines the Reform series of a variable of that type.

In the linear recursion case we inspected a characteristic mgu to classify the variables of a clause. For nonlinear recursion we need two characteristic mgu's, one involving the left subtree and the other involving the right subtree.

Characteristic mgu's. Let $S = (H \leftarrow \Psi \wedge L \wedge R)$ be a clause with tail $L \wedge R$, where L is the l-tail and R is the r-tail. Let

$$S_i = (H_i \leftarrow \Psi_i \wedge L_i \wedge R_i), \qquad \text{and} \qquad S_{i+1} = (H_{i+1} \leftarrow \Psi_{i+1} \wedge L_{i+1} \wedge R_{i+1}),$$

be the ith and $(i+1)$th variants of S according to the standard renaming of variables. Then $\theta_{i+1} = \mathrm{mgu}(L_i, H_{i+1})$ and $\gamma_{i+1} = \mathrm{mgu}(R_i, H_{i+1})$ are the *characteristic mgu's* of S. We shall refer to the characteristic mgu's as θ_c and γ_c.

Example 7.4. Consider again the recursive clause of the inorder program:

```
Inorder(T(u, x, y), z) ←
    Inorder(x, w) ∧ Inorder(y, v) ∧ Append(w, u.v, z).
```

The ith and $(i+1)$th variants variants of this clause are:

```
Inorder(T(u_i, x_i, y_i), z_i) ←
    Inorder(x_i, w_i) ∧ Inorder(y_i, v_i) ∧
    Append(w_i, u_i.v_i, z_i).
Inorder(T(u_{i+1}, x_{i+1}, y_{i+1}), z_{i+1}) ←
    Inorder(x_{i+1}, w_{i+1}) ∧ Inorder(y_{i+1}, v_{i+1}) ∧
    Append(w_{i+1}, u_{i+1}.v_{i+1}, z_{i+1}).
```

The characteristic mgu's are thus:

$$\theta_c = \{\mathtt{T(u_{i+1}, x_{i+1}, y_{i+1})/x_i,\ z_{i+1}/w_i}\}, \qquad \text{and}$$
$$\gamma_c = \{\mathtt{T(u_{i+1}, x_{i+1}, y_{i+1})/y_i,\ z_{i+1}/v_i}\}.$$

Variable categories. Let us now extend the variable classification carried out for linear recursion to tree recursion. The variable type NONE is unchanged. POS and NEG have two counterparts each in the nonlinear case:

- Category POS corresponds to two categories POSL and POSR (acronyms for 'POS left tail' and 'POS right tail', respectively). Example: x is a POSL-variable and y is a POSR-variable in the clause

$$P(T(u,v,w),z) \leftarrow P(v,x) \wedge P(w,y).$$

- Category NEG corresponds to two categories NEGL and NEGR. Example: x is a NEGL-variable and y is a NEGR-variable in the clause

$$P(T(u,v,w),x,y) \leftarrow P(v,q,r) \wedge P(w,s,t).$$

For the purpose of comprehensibility we shall only discuss POSL and POSR in detail here. We will use the variables of the `Inorder` program as examples in describing the variable classes. We assume that the program is invoked by a call that has a recursion tree with the size and structure of the tree in Example 7.2. Recall the recursive clause of the program:

```
Inorder(T(u, x, y), z) ←
    Inorder(x, w) ∧ Inorder(y, v) ∧ Append(w, u.v, z).
```

The variables `x` and `w` are of type POSL, `y` and `v` are of type POSR, and `z` is of type NONE.

In the definitions below $\theta_1, \ldots, \theta_k$ denote the mgu's obtained when resolving left tails and $\gamma_1, \ldots, \gamma_m$ denote the mgu's obtained when resolving right tails. We refer to the characteristic mgu for the l-tail and r-tail as θ_c and γ_c, respectively.

POSL-variables. Let t be a term. A variable x is a *POSL-variable* if and only if

$$\varphi(x,i)\theta_{i+1} = \tau(t,\theta_c,i+1), \qquad \text{for} \quad 1 \le i < n.$$

That is, x occurs in the l-tail of the clause and there is a matching term t in the clause head. Example:

$$\mathtt{P(T(u,v,w),t)} \leftarrow \mathtt{P(v,x)} \wedge \mathtt{P(w,y)}.$$

The function POSL defines the Reform series of x:

$$\mathrm{POSL}(n,l,x,t,\theta_c) = \langle \bar{x}_1,\dots,\bar{x}_n\rangle,$$
$$\text{where} \quad \bar{x}_i = \begin{cases} \varphi(x,i), & \text{if} \quad l[i]=0; \\ \tau(t,\theta_c,i+1), & \text{if} \quad l[i]=1; \end{cases} \qquad \text{for} \quad 1 \le i \le n.$$

Example (cf. Example 7.2):

$$\mathrm{POSL}(8,[1,1,0,0,1,0,0,0],\mathtt{w},\mathtt{z},\{\mathtt{T(u_{i+1},x_{i+1},y_{i+1})/x_i,\ z_{i+1}/w_i}\})$$
$$= \langle \mathtt{z_2},\mathtt{z_3},\mathtt{w_3},\mathtt{w_4},\mathtt{z_6},\mathtt{w_6},\mathtt{w_7},\mathtt{w_8}\rangle$$

We obtain the following lemma from the definition of POSL-variable.

Lemma 7.1. *Let x be a POSL-variable and t the corresponding term in the clause head. Then, for $k \ge 1$:*

$$\varphi(x,k)(\theta_1 * \cdots * \theta_k) = \varphi(x,k) \qquad \textit{and}$$
$$\varphi(x,k)(\theta_1 * \cdots * \theta_{k+1}) = \begin{cases} \varphi(x,k), & \textit{if} \quad l[k]=0; \\ \tau(t,\theta_c,k+1), & \textit{if} \quad l[k]=1. \end{cases}$$

We shall now show that the function POSL defined above indeed computes the Reform series of a POSL-variable.

Theorem 7.1. *Let x be a POSL-variable and t the corresponding term in the clause head. Then the Reform series $\langle \bar{x}_1,\dots,\bar{x}_n\rangle$ of x is given by:*

$$\bar{x}_i = \varphi(x,i)(\theta_1 * \cdots * \theta_n) = \begin{cases} \varphi(x,i), & \textit{if} \quad l[i]=0; \\ \tau(t,\theta_c,i+1), & \textit{if} \quad l[i]=1; \end{cases} \qquad \textit{for} \quad 1 \le i \le n.$$

Proof. The proof is by induction on n, the length of the Reform transformation. Base ($n=2$): From Lemma 7.1 we know that $\varphi(x,1)\theta_2 = \varphi(x,1)$ if $l[1]=0$, and $\varphi(x,1)\theta_2 = \tau(t,\theta_c,2)$ if $l[1]=1$. Hence, since $\theta_1 = \epsilon$,

$$\varphi(x,1)(\theta_1 * \theta_2) = \begin{cases} \varphi(x,i), & \text{if} \quad l[i]=0; \\ \tau(t,\theta_c,i+1), & \text{if} \quad l[i]=1. \end{cases}$$

From Lemma 7.1 we also know that $\varphi(x,2)\theta_2 = \varphi(x,1)$. Hence $\varphi(x,2)(\theta_1 * \theta_2) = \varphi(x,1)$. Obviously $l[2]=0$, since node 2 is the last node in the preorder traversal—it cannot have a nonempty left subtree.

Induction hypothesis:

$$\varphi(x,i)(\theta_1 * \cdots * \theta_n) = \begin{cases} \varphi(x,i), & \text{if} \quad l[i]=0; \\ \tau(t,\theta_c,i+1), & \text{if} \quad l[i]=1; \end{cases} \qquad \text{for} \quad 1 \le i \le n.$$

Induction step: We have three cases.

$(i \le n$ and $l[i]=0)$

$\varphi(x,i)(\theta_1 * \cdots * \theta_{n+1}) \quad = \varphi(x,i)$ — by definition, if $i<n$; by Lemma 7.1, if $i=n$

$(i \le n$ and $l[i]=1)$

$\varphi(x,i)(\theta_1 * \cdots * \theta_{n+1}) \quad = \tau(t,\theta_c,i+1)(\theta_1 * \cdots * \theta_{n+1})$ — by induction hypothesis

$\quad = \tau(t,\theta_c,i+1)$ — by definition of τ

$(i = n+1)$

$\varphi(x,i)(\theta_1 * \cdots * \theta_{n+1}) \quad = \varphi(x,n+1)$ — by Lemma 7.1

Hence, for $1 \le i \le (n+1)$:

$$\varphi(x,i)(\theta_1 * \cdots * \theta_{n+1}) = \begin{cases} \varphi(x,i), & \text{if} \quad l[i]=0; \\ \tau(t,\theta_c,i+1), & \text{if} \quad l[i]=1. \quad \blacksquare \end{cases}$$

POSR-variables. Let t be a term. A variable x is a *POSR-variable* if and only if

$$\varphi(x,i)\gamma_{i+1} = \tau(t,\gamma_c,i+1), \qquad \text{for} \quad 1 \le i < n.$$

That is, x occurs in the r-tail of the clause and there is a matching term t in the clause head. Example:

$$\texttt{P(T(u,v,w),t)} \leftarrow \texttt{P(v,y)} \land \texttt{P(w,x)}.$$

The function POSR defines the Reform series of x:

$$\text{POSR}(n,r,x,t,\gamma_c) = \langle \bar{x}_1,\ldots,\bar{x}_n \rangle,$$
$$\text{where} \quad \bar{x}_i = \begin{cases} \varphi(x,i), & \text{if} \quad r[i]=0; \\ \tau(t,\gamma_c,r[i]), & \text{if} \quad r[i]>0; \end{cases} \qquad \text{for} \quad 1 \le i \le n.$$

Example (cf. Example 7.2):

$$\text{POSR}(8,[5,4,0,0,8,7,0,0],\texttt{v},\texttt{z},\{\texttt{T(u}_{\texttt{i+1}}\texttt{, x}_{\texttt{i+1}}\texttt{, y}_{\texttt{i+1}}\texttt{)/y}_{\texttt{i}}\texttt{, z}_{\texttt{i+1}}\texttt{/v}_{\texttt{i}}\})$$
$$= \langle \texttt{z}_5,\texttt{z}_4,\texttt{v}_3,\texttt{v}_4,\texttt{z}_8,\texttt{z}_7,\texttt{v}_7,\texttt{v}_8 \rangle$$

We obtain the following result.

Theorem 7.2. *Let x be a POSR-variable and t the corresponding term in the clause head. Then the Reform series $\langle \bar{x}_1, \ldots, \bar{x}_n \rangle$ of x is given by:*

$$\bar{x}_i = \varphi(x,i)(\gamma_1 * \cdots * \gamma_n) = \begin{cases} \varphi(x,i), & \text{if } \ r[i] = 0; \\ \tau(t, \gamma_c, r[i]), & \text{if } \ r[i] > 0; \end{cases} \qquad \text{for} \quad 1 \leq i \leq n.$$

Proof. The proof is analogous to the proof of Theorem 7.1.

It should be noted that the somewhat asymmetric relation between the Reform series of POSL- and POSR-variables—as compared to the left-right symmetry of the program in which the variables occur—is due to our use of the preorder traversal for linear representation of binary trees.

7.4. COROLLARIES FOR RECURRING SUBSTITUTIONS.

As in the case of linear recursion we identify certain cases when more efficient expressions for the Reform series can be given. Hence INV, POS-LIST and NEG-LIST in the linear case have the following nonlinear counterparts.

- INV corresponds to INVL and INVR ('INV left tail' and 'INV right tail'). Example: x is an INVL-variable and y is an INVR-variable in the clause

$$P(T(u,v,w),x,y) \leftarrow P(v,x,s) \wedge P(w,t,y).$$

- POS-LIST corresponds to POSL-LTREE, POSL-RTREE, POSR-LTREE, and POSR-RTREE (acronyms for 'POSL left subtree', 'POSL right subtree', 'POSR left subtree', and 'POSR right subtree', respectively). Example: v is a POSL-LTREE-variable, z is a POSL-RTREE-variable, y is a POSR-LTREE-variable, and w is a POSR-RTREE-variable in the clause

$$P(T(u,v,w),T(x,y,z)) \leftarrow P(v,z) \wedge P(w,y).$$

- NEG-LIST corresponds to NEGL-LTREE, NEGL-RTREE, NEGR-LTREE, and NEGR-RTREE. Example: w is a NEGL-LTREE-variable, x is a NEGL-RTREE-variable, y is a NEGR-LTREE-variable, and z is a NEGR-RTREE-variable in the clause

$$\begin{aligned} &P(T(t,u,v),w,x,y,z) \leftarrow \\ &\qquad P(u,T(...,w,...),T(...,...,x),...,...) \wedge \\ &\qquad P(v,...,...,T(...,y,...),T(...,...,z)) \end{aligned}$$

Let us now consider two of these special cases, POSL-LTREE and POSR-RTREE, in greater detail.

POSL-LTREE-variables. Let u and y be terms. A POSL-variable x is a *POSL-LTREE-variable* if and only if

$$\varphi(x,i)\theta_{i+1} = T(\tau(u,\theta_c,i+1),\tau(x,\theta_c,i+1),\tau(y,\theta_c,i+1)), \qquad \text{for} \quad 1 \le i < n.$$

That is, x occurs in the l-tail of the clause and there is a matching term $T(u,x,y)$ in the clause head. Example:

$$\mathtt{P(T(u,x,y))} \leftarrow \mathtt{P(x)} \wedge \mathtt{P(y)}.$$

It is straightforward to verify that the Reform series $\langle \bar{x}_1,\ldots,\bar{x}_n\rangle$ of a POSL-LTREE-variable x is given by $(1 \le i < n)$:

$$\bar{x}_i = \begin{cases} \varphi(x,i), & \text{if} \quad q[i+1]=1; \\ T(\tau(u,\theta_c,i+1),T(\ldots,T(\tau(u,\theta_c,i+m), & \\ \qquad \varphi(x,i+m),\tau(y,\theta_c,i+m)),\ldots),T(\tau(y,\theta_c,i+1)), & \text{if} \quad q[i+1]=0, \end{cases}$$

where m is the smallest $j > 0$ such that $q[i+j+1]=1$. (This is the only time we use the q structure table.) Example (cf. Example 7.2):

$$\begin{aligned}&\mathrm{POSL}(8,[1,1,0,0,1,0,0,0],\mathtt{x},\mathtt{T(u,x,y)},\{\mathtt{T(u_{i+1},x_{i+1},y_{i+1})/x_i},\ \mathtt{z_{i+1}/w_i}\})\\ &\quad = \langle \mathtt{T(u_2,T(u_3,x_3,y_3),y_2)},\mathtt{T(u_3,x_3,y_3)},\mathtt{x_3},\mathtt{x_4},\mathtt{T(u_6,x_6,y_6)},\mathtt{x_6},\mathtt{x_7},\mathtt{x_8}\rangle\end{aligned}$$

POSR-RTREE-variables. Let u and y be terms. A POSR-variable x is a *POSR-RTREE-variable* if and only if

$$\varphi(x,i)\gamma_{i+1} = T(\tau(u,\gamma_c,i+1),\tau(y,\gamma_c,i+1),\tau(x,\gamma_c,i+1)), \qquad \text{for} \quad 1 \le i < n.$$

That is, x occurs in the r-tail of the clause and there is a matching term $T(u,y,x)$ in the clause head. Example:

$$\mathtt{P(T(u,y,x))} \leftarrow \mathtt{P(y)} \wedge \mathtt{P(x)}.$$

It is straightforward to verify that the Reform series $\langle \bar{x}_1,\ldots,\bar{x}_n\rangle$ of a POSR-RTREE-variable x is given by $(1 \le i < n)$:

$$\bar{x}_i = \begin{cases} \varphi(x,i), & \text{if} \quad r[i]=0; \\ T(\tau(u,\gamma_c,r[i]),T(\tau(y,\gamma_c,r[i]),T(\ldots,\ldots, & \\ \qquad T(\tau(u,\gamma_c,r^m[i]),\varphi(y,r^m[i]),\tau(x,\gamma_c,r^m[i]))\ldots)), & \text{if} \quad r[i]>0, \end{cases}$$

where m is the smallest j such that $r^{j+1}[i]=0$. Example (cf. Example 7.2):

$$\begin{aligned}&\mathrm{POSR}(8,[5,4,0,0,8,7,0,0],\mathtt{y},\mathtt{T(u,x,y)},\{\mathtt{T(u_{i+1},x_{i+1},y_{i+1})/y_i},\ \mathtt{z_{i+1}/v_i}\})\\ &\quad = \langle \mathtt{T(u_5,y_5,T(u_8,y_8,x_8))},\mathtt{T(u_4,y_4,x_4)},\mathtt{y_3},\mathtt{y_4},\\ &\qquad\qquad \mathtt{T(u_8,y_8,x_8)},\mathtt{T(u_7,y_7,x_7)},\mathtt{y_7},\mathtt{y_8}\rangle\end{aligned}$$

7.5. COMPILATION

The basic ideas from compilation of linear recursion carry over to tree programs. However, the procedures for head unification and recursive invocation need to be modified. Let us condense the main points in compilation of tree programs as follows.

As with linear recursion, the recursive clause is invoked just once. The recursion tree of the invoking call is traversed in preorder at the invocation. The structure tables are set up during the traversal and the parameter n is obtained.

There is one recursive call for each empty subtree of the recursion tree. Each such call is known to match the base clause but may also match the recursive clause (if the empty subtree is an unbound variable). We implement this part of the clause as a loop for $i = 1 \ldots, n$ where the structure table entries $l[i]$ and $r[i]$—which indicate if the left and right subtree, respectively, of the ith node of the preorder traversal is empty—determine if any recursive calls should be issued in the ith iteration.

The residual body is compiled to a loop for $i = 1 \ldots, n$. The ith elements of the Reform series are used in place of the program variables in the ith iteration.

We shall now discuss these issues in more detail. We use the `Inorder` program as an example:

```
    Inorder(∅, ∅).
    Inorder(T(u, x, y), z) ←
        Inorder(x, w) ∧ Inorder(y, v) ∧ Append(w, u.v, z).
```

We have the following variable classification.

Variable	Variable class
u, z	NONE
w	POSL
v	POSR
x	POSL (POSL-LTREE)
y	POSR (POSR-RTREE)

Head unification. Assume that the ith argument `Ai` of the invoking call is the recursion tree and let `T(u,x,y)` be the corresponding tree in the clause. The unification of the two trees is compiled to the operation

```
<n,l,r,q> := preorder(Ai,u,x,y)
```

which has the following effects:

- The vector `u` is obtained as the preorder traversal of the tree `Ai`.

- The vector x (y) is obtained as the successive left (right) subtrees encountered during the preorder traversal. If x (y) does not occur in the residual body of the clause, then only empty left (right) subtrees need to be stored.
- The number of nodes n is obtained and the structure tables l, r and q are set up.

Barklund (1990) gives a parallel algorithm for this operation and for tree construction operations ('write mode' unification).

Example 7.5. The head of the `Inorder` program is compiled to:

```
<n,l,r,q> := preorder(A1,u,x,y)
z[1] := A2
```

Representation and computation of Reform series. Let us consider representation of the Reform series of POSL-variables and POSR-variables. For a POSL-variable x we need only represent explicitly those elements of the series that correspond to empty left subtrees, that is those $\bar{x}_i$ for which $l[i] = 0$. For a POSR-variable x we need only represent explicitly those elements of the series that correspond to empty right subtrees, that is those $\bar{x}_i$ for which $r[i] = 0$. Hence there will be empty entries or 'holes' in the vector representing x where $l[i] = 1$ and $r[i] > 0$, respectively, since those entries correspond to nonempty subtrees.

We turn next to computation of the Reform series. Let w[1:n] and v[1:n] be Reform series of a POSL-variable and a POSR-variable, respectively, and let z[1:n] be the Reform series of the corresponding term in the clause head. Then w[1:n] and v[1:n] are initiated as follows, for $1 \leq i \leq n$.

```
if  l[i] = 0  then  w[i] := undef  else  w[i] := z[i+1];
if  r[i] = 0  then  v[i] := undef  else  v[i] := z[r[i]].
```

In other words, we utilize the 'holes' in the vectors x and y that correspond to nonempty subtrees, for storing those elements of the vector z that are used in place of x and y later in the computation.

Example 7.6. The Reform series of w and v in the `Inorder` program are computed by the following code.

```
for* i = 1 to n do
   if l[i] = 0 then w[i] := undef else w[i] := z[i+1]
   if r[i] = 0 then v[i] := undef else v[i] := z[r[i]]
end
```

This is a parallel loop.

Compilation of the clause body. The program is called recursively with each empty subtree. Suppose that the vector x (y) holds pointers to all left (right) subtrees. We know which recursive calls to execute by observing that the subtree `x[i]` (`y[i]`) is empty if `l[i]=0` (`r[i]=0`).

Example 7.7. The recursive calls of the `Inorder` program are compiled to:

```
for* i = 1 to n do
   if l[i] = 0 then call Inorder(x[i],w[i])
   if r[i] = 0 then call Inorder(y[i],v[i])
end
```

This is a parallel loop. Note that this loop can be fused with the loop computing Reform series.

The residual body is compiled exactly as for linear recursion. The body calls that appear textually to the left of the tail are executed in a loop from 1 to n, and the calls that appear to the right of the tail in a loop from n to 1. These loops can be parallelized unless there are impeding data dependencies.

Example 7.8. The residual body of the `Inorder` program is compiled to:

```
for i = n downto 1 do
   call Append(w[i],cons(u[i],v[i]),z[i])
end
```

This loop exhibits data dependencies that prevent parallelization. It might seem that the loop contains no data dependencies, but observe that `w[i] = z[i+1]` if `l[i] = 1` and `v[i] = z[r[i]]` if `r[i] > 0`.

7.6. EXAMPLES

Example 7.9. The following program counts the number of nodes in a tree:

```
Count(∅, 0).
Count(T(u, x, y), z) ← Count(x, w) ∧ Count(y, v) ∧ z = w + v + 1.
```

We have the following variable classification.

Variable	Variable class
u, z	NONE
w	POSL
v	POSR
x	POSL (POSL-LTREE)
y	POSR (POSR-RTREE)

The compiled code for the recursive clause is:

```
u,x,y,z,w,v,l,r,q : vector
n,i : scalar
<n,l,r,q> := preorder(A1,u,x,y)
z[1] := A2
for* i = 2 to n do z[i] := undef end
for* i = 1 to n do
   if l[i] = 0 then
      w[i] := undef
      call Count(x[i],w[i])
   else w[i] := z[i+1]
   if r[i] = 0 then
      v[i] := undef
      call Count(y[i],v[i])
   else v[i] := z[r[i]]
end
for i = n downto 1 do z[i] := w[i]+v[i]+1 end
```

The only sequential part of this program is the last loop. There are data dependencies analogous to those in the `Inorder` program above.

Example 7.10. The following tree program is from chapter 8, where it is obtained as the result of a program transformation.

```
IsoTree(∅, ∅).
IsoTree(T(u, x, y), T(u, x1, y1)) ←
    Iso(x1, x2, y1, y2) ∧ IsoTree(x, x2) ∧ IsoTree(y, y2).
```

where

```
Iso(x, x, y, y).
Iso(x, y, y, x).
```

We have the following variable classification.

Variable	Variable class
u, x1, y1	NONE
x2	POSL
y2	POSR
x	POSL (POSL-LTREE)
y	POSR (POSR-RTREE)

The compiled code for the recursive clause is:

```
u,x,y,x1,y1,x2,y2,l,r,q : vector
n,i : scalar
<n,l,r,q> := preorder(A1,u,x,y)
```

```
for* i = 1 to n do x1[i] := undef end
for* i = 1 to n do y1[i] := undef end
unify A2 with new term T(u[1],x1[1],y1[1])
for* i = 1 to n do
   if l[i] = 0 then x2[i] := undef
   else x2[i] := new term T(u[i+1],x1[i+1],y1[i+1])
   if r[i] = 0 then y2[i] := undef
   else y2[i] := new term T(u[r[i]],x1[r[i]],y1[r[i]])
   call Iso(x1[i],x2[i],y1[i],y2[i])
   if l[i] = 0 then call IsoTree(x[i],x2[i])
   if r[i] = 0 then call IsoTree(y[i],y2[i])
end
```

All loops in this program can be parallelized (the calls to `Iso` give no data dependencies that impede parallelization).

7.7. CONCLUSION

We have generalized our compilation theory for linear recursion to nonlinear recursion over binary trees. We compile tree programs to iteration over vectors, using a temporary linear representation of the recursion tree. This linear representation is obtained each time a tree program is invoked, as a side effect of head unification.

CHAPTER 8

DISCUSSION

We have assumed that the programs to be compiled satisfy rather restricted recursion schemas. In this chapter we discuss transformation of a larger class of programs to this restricted form. We describe several transformation techniques informally.

8.1. BRANCHING RECURSION

We consider first those programs that have several recursive branches. The outcome of exclusive tests might determine which branch is followed in each recursion, or there might be a nondeterministic choice between the branches. We say that such programs use *branching recursion*.

Branching recursion is transformed by fusing the recursive branches into one single recursion, hiding the different outcomes of the branches within an auxiliary predicate.

Example 8.1. The list partitioning program shown below is a typical example of branching recursion. The tests in each recursive branch determine the bindings of the two output variables.

```
Partition(∅, w, ∅, ∅).
Partition(u.x, w, u.y, z) ← u ≤ w ∧ Partition(x, w, y, z).
Partition(u.x, w, y, u.z) ← u > w ∧ Partition(x, w, y, z).
```

Let us now transform this program. The alternative results of the tests are hidden in the predicate `Part`, and the two recursive clauses of `Partition` are merged into one:

```
Partition(∅, w, ∅, ∅).
Partition(u.x, w, y, z) ←
    Part(u, w, y, y1, z, z1) ∧
    Partition(x, w, y1, z1).
```

where

```
Part(u, w, u.y, y, z, z) ← u ≤ w.
Part(u, w, y, y, u.z, z) ← u > w.
```

This program obeys the recursion schema.

Example 8.2. Another example of branching recursion is the program for testing whether two binary trees are isomorphic (two trees t and t' are isomorphic if t' can be obtained from t by reordering the branches of the subtrees of t):

```
IsoTree(∅, ∅).
IsoTree(T(u, x, y), T(u, x1, y1)) ← IsoTree(x, x1) ∧ IsoTree(y, y1).
IsoTree(T(u, x, y), T(u, x1, y1)) ← IsoTree(x, y1) ∧ IsoTree(y, x1).
```

The order of the left and right subtree is unaltered in the first recursive clause but exchanged in the second recursive clause. We conceal this nondeterministic reordering with an auxiliary predicate `Iso` and transform the two recursive clauses into one:

```
IsoTree(∅, ∅).
IsoTree(T(u, x, y), T(u, x1, y1)) ←
    Iso(x1, x2, y1, y2) ∧ IsoTree(x, x2) ∧ IsoTree(y, y2).
```

where

```
Iso(x, x, y, y).
Iso(x, y, y, x).
```

This program obeys the recursion schema.

8.2. CONDITIONAL RECURSION

The next variation of structural recursion we consider is *conditional recursion.* Recursion is conditional if there are nontrivial termination conditions.

We shall discuss two approaches to conditional recursion. The first approach (referred to as Method A) is parallel in nature, in that as much parallelism as possible is squeezed out of the program, at the cost of performing some redundant computation. The second approach (Method B) is more economical, but may constrain parallel head unification.

Method A. Conditional flow of control in a program impedes efficient exploitation of parallelism on some types of parallel machines. One solution to this problem is to ignore the conditional flow of control, and to perform redundant computation in the computation branches not selected. On a sequential machine, this may of course be very inefficient. On a parallel machine, however, it may be advantageous to perform some redundant computation if a more parallel algorithm can be used.

The idea is to transform a conditionally recursive program by inserting 'dummy' recursive calls. In this way we obtain a program that uses branching recursion; this program can be further transformed with the technique described in the previous section.

Example 8.3. The following program inserts an integer z into an ordered list of integers.

```
Insert(∅, z, z.∅).
Insert(x.y, z, z.x.y) ← x ≥ z.
Insert(x.y, z, x.w) ← x < z ∧ Insert(y, z, w).
```

The recursion terminates if the element to be inserted is smaller than the first element in the list. We introduce an auxiliary predicate Ins that performs the termination test and binds the tail of the recursion list to a dummy variable in case the test should succeed:

```
Insert(∅, z, z.∅).
Insert(x.y, z, u) ← Ins(x, y, z, u, w) ∧ Insert(y, z, w).
```

where

```
Ins(x, y, z, z.x.y, w) ← x ≥ z.
Ins(x, y, z, x.w, w) ← x < z.
```

The transformed program compares the element to be inserted with every element in the list. Fortunately, all comparisons, and the resulting bindings of variables, can be done simultaneously.

Example 8.4. Another example of conditional recursion is the program for inserting a new node into a leaf of an ordered binary tree, keeping the tree ordered:

```
Leafinsert(∅, v, T(v, ∅, ∅)).
Leafinsert(T(v, x, y), v, T(v, x, y)).
Leafinsert(T(u, x, y), v, T(u, x1, y)) ← u > v ∧ Leafinsert(x, v, x1).
Leafinsert(T(u, x, y), v, T(u, x, y1)) ← u < v ∧ Leafinsert(y, v, y1).
```

We introduce an auxiliary predicate Lins that performs the test and binds the subtree not chosen for recursion to a dummy variable. The transformed program, which satisfies the recursion schema for binary trees, is:

```
Leafinsert(∅, v, T(v, ∅, ∅)).
Leafinsert(T(u, x, y), v, T(u, x1, y1)) ←
    Lins(u, v, x, x1, x2, y, y1, y2) ∧
    Leafinsert(x, v, x2) ∧ Leafinsert(y, v, y2).
```

where

```
Lins(v, v, x, x, x1, y, y, y1).
Lins(u, v, x, x1, x1, y, y, y1) ← u > v.
Lins(u, v, x, x, x1, y, y1, y1) ← u < v.
```

This program leads to redundant computation in that the entire tree is traversed in head unification and there is one call to `Lins` for each node in the tree. Fortunately, the program is now highly parallel.

Method B. If we decide that it is not worth the expense to perform some redundant computation in order to gain parallelism, we can modify the scheme for head unification in the compiled program in order to incorporate all termination conditions of the recursion in the operation `Traverse` (or the corresponding operation for integer and tree programs). Thus, we employ specialized versions of `Traverse` for this type of program.

Example 8.5. The program for list membership is:

```
Member(z.y, z).
Member(x.y, z) ← Member(y, z).
```

We compile the recursive clause to:

```
x : vector
y,z,n : scalar
z := A2
n := Member_traverse(A1,x,y,z)
call Member(y,z)
```

Here `Member_traverse` does the job of `Traverse` but, in addition, it tries to unify each element in the list `A1` with `z` since this is the termination condition of the recursion. It terminates when the unification succeeds; `n` is then the number of failed attempts.

Example 8.6. Let us reconsider the list insertion program discussed above.

```
Insert(∅, z, z.∅).
Insert(x.y, z, z.x.y) ← x ≥ z.
Insert(x.y, z, x.w) ← x < z ∧ Insert(y, z, w).
```

We compile the recursive clause to (we assume that `Traverse*` is the variant of `Traverse` employed when this is a subsequent occurrence of the head element):

```
x,w : vector
y,z,n,k : scalar
z := A2
```

```
n := Insert_traverse(A1,x,y,z)
k := Traverse*(A3,x,w)
if k<n then
   w[n] := undef
   for* i=k to n-1 do w[i] := cons(x[i],nil)
   for* i=k to n-1 do addr(tail(w[i])) := w[i+1]
for* i=1 to n do z<x[i] end
call Insert(y,z,w[n])
```

Here, `Insert_traverse` performs two tests for each sublist of the input list `A1`: whether the head element is greater than or equal to `z`, and whether the tail is not a list. The traversal terminates when one of these tests succeeds, with `n` being the number of failed attempts.

8.3. BOUNDED RECURSION

Sometimes we can transform a complicated recursion to simple integer recursion if we know the recursion depth n when the program is invoked. We can then provide n as recursion argument to the transformed program.

Example 8.7. Our first example is the list permutation program below. This program is typically called with the first argument being input and the second argument output. The predicate `Select` called from the body of the recursive clause nondeterministically selects one element u from the input list x; z is the list obtained by removing u from x.

```
Permutation(∅, ∅).
Permutation(x, u.y) ← Select(x, u, z) ∧ Permutation(z, y).
```

This is a case of non-structural recursion if we consider the input argument to be the recursion term. We add the size of the input list as an extra argument to the program:

```
Permutation(x, y) ← Length(x, n) ∧ Permutation*(n, x, y).
Permutation*(0, ∅, ∅).
Permutation*(n + 1, x, u.y) ← Select(x, u, z) ∧ Permutation*(n, z, y).
```

This program satisfies the schema for integer recursion. We note that the call `Length(x, n)` can be computed in $O(\log n)$ time (Barklund, 1990).

Example 8.8. Our next example is a program that, when given two integers x and y, such that $x \leq y$, produces a list of integers $x, \ldots, y$.

```
Range(x, x, x.∅).
Range(x, y, x.z) ← x < y ∧ Range(x + 1, y, z).
```

This is in fact an example of conditional recursion. However, we note that $n = y - x + 1$ and obtain a more efficient program by transformation:

```
Range(x, y, z) ← n = y - x + 1 ∧ Range*(n, x, y).

Range*(0, x, x, x.∅).
Range*(n + 1, x, y, x.z) ← x < y ∧ Range*(n, x + 1, y, z).
```

Example 8.9. Let us finally consider an example in which this transformation technique gives rise to conditional recursion. The following program merges two sorted lists of integers:

```
Merge(∅, y, y).
Merge(x, ∅, x).
Merge(u.x, v.y, u.z) ← u ≤ v ∧ Merge(x, v.y, z).
Merge(u.x, v.y, v.z) ← u > v ∧ Merge(u.x, y, z).
```

Here the recursion argument is the first and second argument alternatingly. However, we observe that $n \le (n_1 + n_2)$, where n_1 and n_2 is the length of the first and second input list, respectively. We provide n as an extra argument and obtain the program:

```
Merge(x, y, z) ←
    Length(x, n1) ∧ Length(y, n2) ∧
    n = n1 + n2 ∧ Merge*(n, x, y, z).

Merge*(0, ∅, ∅, ∅).
Merge*(n, ∅, y, y).
Merge(n, x, ∅, x).
Merge*(n + 1, u.x, v.y, w.z) ←
    Mer(u, v, w, x, x1, y, y1) ∧ Merge*(n, x1, y1, z).
```

where

```
Mer(u, v, u, x, x, y, v.y) ← u ≤ v.
Mer(u, v, v, x, u.x, y, y) ← u > v.
```

This program is conditionally recursive and can be compiled according to the earlier discussion.

8.4. NONLINEAR RECURSION ON INTEGERS AND LISTS

Many cases of nonlinear recursion on integers and lists can be dealt with as if they were linear recursion. One of the recursive calls is selected as *the* recursive call, and the other recursive calls are considered to be part of the residual body of the clause.

Example 8.10. Consider the program for 'flattening' a list:

```
Flatten(∅, ∅).
Flatten(x, x) ← Constant(x).
Flatten(x.y, z) ← Flatten(x, x1) ∧ Flatten(y, y1) ∧ Append(x1, y1, z).
```

We can view this program to use linear recursion by considering the first recursive call to be part of the residual body.

Example 8.11. Consider next Takeuchi's `Tarai`-function, expressed as a logic program:

```
Tarai(x, y, z, z) ← x ≤y.
Tarai(x, y, z, w) ←
    Tarai(x - 1, y, z, x1) ∧ Tarai(y - 1, z, x, y1) ∧
    Tarai(z - 1, x, y, z1) ∧ Tarai(x1, y1, z1, w).
```

We consider `Tarai(x - 1, y, z, x1)` to be *the* recursive call, and the other three literals of the recursive clause as part of the residual body.

8.5. NON-STRUCTURAL RECURSION

Some programs use non-structural recursion because the programmer chooses to conceal the structure of his data, using relations instead of terms for data representation. Other programs implement algorithms that are non-structural in nature. In both cases the compilation method defined in this thesis might be used for subprograms that are structural.

Example 8.12. The following program sorts a list into a difference list (Clark & Tärnlund, 1977) with the Quicksort algorithm:

```
Quick(∅, y-y).
Quick(u.x, y-z) ←
    Partition(x, u, x1, x2) ∧
    Quick(x1, y-u.z1) ∧ Quick(x2, z1-z).
```

This is an example of non-structural recursion. However, `Partition` might be compiled with our method, although `Quick` is not.

CHAPTER 9

HISTORICAL NOTES

In this chapter we shall review some related developments of historical significance.

9.1. RECURSION

The first programming language that allowed recursion was IPL-II developed by Allan Newell, Herbert Simon and J. C. Shaw[1] in 1956. IPL-II was a machine language that did not allow symbolic computation. Closed subroutines was a well-known idea in 1956 and the designers of IPL-II implemented recursion as subroutine calls. However, the usual techniques for parameter passing were not applicable when the subroutines were recursive; the solution was to use a public stack for parameter passing and return addresses. The language implementation took care of the return addresses, but it was the programmer's responsibility to pass the parameters via the stack. (Stacks had been used earlier, mainly for evaluation of arithmetical expressions.)

John McCarthy's Lisp[2] was the first high-level language supporting recursion. As a programming language Lisp was very different from IPL-II. It was designed with the dual purpose of being a language for symbolic computation and a tool for theoretical work. Recursion accompanied by McCarthy's novel conditional expression form **if** p **then** a **else** b, where a and b were arbitrary expressions, provided a powerful programming methodology.

In 1959 McCarthy[3] proposed that recursion and the conditional expression should be included in the language Algol 60 which was then being devised. The conditional expression was readily accepted. Recursion was also included in the document defining the language, the Algol 60 report.[4]

The same method was used for dealing with recursion in the first implementations of Lisp and Algol 60: procedure calls with a stack for parameter passing and return addresses.[5] This was essentially the IPL-II concept with one modification: the stack was stored in sequential memory allocations, while IPL-II represented the stack as a linked structure.

This technique for implementation of recursion is now the standard approach. However, the scheme can be optimized by recognizing the case when the final instruction of a procedure is a procedure call. Such a procedure call might be replaced by a jump to the called procedure's entry point. The effect of this optimization when the procedure is tail-recursive, i.e., when the last procedure call of a procedure is to the procedure itself, is that tail-recursion is transformed to iteration. This is known as *tail-recursion optimization.* The origin of tail-recursion optimization is wrapped in obscurity. However, Steele[6] hints that it was adapted in Lisp compilers in the mid-1960s.

9.2. LOGIC PROGRAMMING

Let us now turn to the historical development of logic programming. We shall not attempt to review the development of the entire field of logic programming. Rather, we shall restrict our attention to (*a*) the background to logic programming, (*b*) the development of implementations, and (*c*) explorations of parallel computation models.

Background

The concept of logic programming emerged in the early 1970s. Let us consider the relevant developments prior to 1972.

Alain Colmerauer had been studying nondeterministic parsing algorithms and their implementation, as well as natural language processing, during the second half of the 1960s. This work resulted in 1970 in a system for natural language translation based on Q-grammars.[7] This system had several features which later appeared in Prolog, the first logic programming language. Programs written in the system were rewriting grammar rules which incorporated nondeterminism and pattern matching. Essential to the implementation was Floyd's[8] idea of using backtracking for nondeterministic procedures; this included a trail stack for recording variable bindings that should be undone while backtracking. One essential difference between Q-grammars and Prolog is that the former was used for bottom-up parsing while the latter is a top-down system.

Problems with 'combinatorical explosions' had been a major obstacle in the field of automatic theorem proving since its inception. The development of the Resolution Principle by Robinson[9] in 1965 was a great improvement but the problems 'combinatorical explosions' were still overwhelming (and still are). A series of refinements of the resolution idea reached its culmination with Kowalski and Kuehner's SL-resolution[10] in 1971. This resolution procedure had a problem-solving behaviour in that it admitted goal-directed deductions. At that time it had been recognized that Horn clauses constitute an interesting subclass of the clausal form of predicate logic. In fact, SLN-resolution for Horn clauses defined by Kuehner[11] in 1971 was very close to the procedure which later was used in logic programming.

So, in 1971 there was the Q-grammar system, SL-resolution, and SLN-resolution. The one remaining factor was the idea that axioms may convey pragmatic information as well as logical meaning. This idea, attributed to Kowalski, is absolutely essential to the concept of logic programming. Kowalski's 1974 paper on predicate logic as a programming language[12] marks the emergence of logic programming as we know it today.

Implementation

The first Prolog system was written in Algol-W (a variation of Algol 60 designed by Wirth) by Roussel[13] in 1972. Improved versions, employing Boyer and Moore's[14] structure sharing technique, were soon written in Marseilles and other places.

Advances in Prolog implementation technology were rapid in the second half of the 1970s. Important contributions were, in particular, made by Maurice Bruynooghe and David Warren.

Bruynooghe[15] investigated memory-efficient implementation techniques for Prolog interpretation. For example, his interpreter contained the first Prolog implementation of tail-recursion optimization.[16]

The first compiler for Prolog was developed by Warren[17] in 1977. Warren's implementation introduced several new techniques, including a two-stack allocation scheme for variables and the use of efficient indexing techniques for clause selection. Perhaps the most important advance, however, resulted from compilation of the unification algorithm. The idea was that a compiler can detect many cases when simple tests and assignments can replace invocation of a general unification algorithm.

Warren[18] presented in 1983 a refinement of his compiler-based implementation model. This design employed new representation and administration schemes for program states, allowing several new space and time saving compiler optimizations.

Parallelism

The possibility of computing the individual calls of a goal statement in parallel if the calls are independent was noted by Kowalski in his original paper from 1974. This form of parallelism was later coined AND-parallelism. Dependent calls, Kowalski suggested, could be computed using co-routining.

The potential for AND-parallelism in logic programs was also pointed out by Tärnlund[19] in 1975. It was suggested that the programmer could increase this potential in recursive programs by unfolding the recursions and expanding the terms. This is the idea that was later developed to become Reform. Tärnlund noted that the same idea could be applied to the unification algorithm expressed as a recursive logic program, thereby obtaining a parallel unification algorithm.

Kowalski's proposal for co-routining and other control schemes was carried further by Clark and McCabe[20] in their language IC-Prolog and by Hansson, Haridi and Tärnlund.[21] These languages provided control structures for expressing concurrent algorithms. Although they were not designed for implementation on parallel machines, these languages mark the beginning of a wide-spread interest in concurrent logic programming languages for parallel machines.

The first such language was Clark and Gregory's Relational Language,[22] presented in 1981. The Relational Language supported programming of concurrent processes using one-way stream communication via shared variables.

General nondeterminism was abandoned in favour of 'committed-choice' nondeterminism based on Dijkstra's[23] guarded commands. In this vein a whole family of concurrent languages were developed: Clark and Gregory's Parlog[24] in 1983, Shapiro's Concurrent Prolog[25] in 1983, Ueda's Guarded Horn Clauses[26] in 1985, and others. These languages relaxed the inter-process constraints of the Relational Language, allowing two-way communication via variables in stream messages.

Conery and Kibler[27] in 1981 focused on the problem of exploiting parallelism rather than on the problem of expressing concurrency. They envisaged a model in which logic programs could be executed with both AND-parallelism and OR-parallelism (parallel exploration of alternative branches in a nondeterministic computation) while retaining the usual logic semantics. In this model AND-parallelism requires that parallel calls do not share variables at run-time. Maximizing AND-parallelism for nondeterministic programs with this constraint turned out to be quite inefficient. This led DeGroot[28] to suggest a simpler model for AND-parallelism with less run-time overhead at the expense of losing some parallelism. Pure OR-parallelism was first considered by Pollard.[29]

NOTES

1. A. Newell & H. A. Simon. The Logic Theory Machine—A complex information processing system. *IRE Trans. Information Theory* IT–2, 1956, 61–79.
2. J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Comm. ACM* 3, no. 4, 1960, 184–195.
3. J. McCarthy. Letter to the editor. *Comm. ACM* 2, no. 8, 1959.
4. Peter Naur recalls (P. Naur. The European side of the last phase of the development of Algol 60. In *History of Programming Languages* (ed. R. L. Wexelblat), Academic Press, New York, 1981, 92–138.) that he included recursion in the Algol 60 report following a suggestion made by A. van Wijngaarden and E. W. Dijkstra in a telephone call on February 10, 1960.
5. The Lisp implementation is briefly described by McCarthy (ibid.). The Algol implementation is by Dijkstra (E. W. Dijkstra. Recursive programming. *Numerische Mathematik* 2, no. 5, 1960, 312–318.)
6. G. L. Steele, Jr. Debunking the "expensive procedure call" myth. In *Proc. Natl. Conf. ACM*, 1977, 153–162.
7. A. Colmerauer. Les Systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur. Rep. 43, Dept. of Computer Science, Univ. of Montreal, Quebec, 1970.
8. R. W. Floyd. Nondeterministic algorithms. *J. ACM* 14, no. 4, 1967, 636–644.
9. J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM* 12, 1965, 23–41.
10. R. A. Kowalski & D. G. Kuehner. Linear resolution with selector function. *Artificial Intelligence* 2, 1971, 227–260.
11. D. G. Kuehner. Strategies for improving the efficiency of automatic theorem-proving. PhD. Thesis, Univ. of Edinburgh, 1971.
12. R. A. Kowalski. Predicate logic as a computer language. Proc. of IFIP-74, North-Holland Publ., Amsterdam, 1974, 569–574.

13. A. Colmerauer, H. Kanoui, R. Pasero & P. Roussel. Un système De communication homme-machine en Français. Groupe Intelligence Artificielle, Univ. Aix-Marseille II, France, 1973.

14. R. S. Boyer & J. S. Moore. The sharing of structure in theorem proving programs. In *Machine Intelligence* 7 (eds. M. Meltzer & D. Michie), Edinburgh UP, 1972, 101–116.

15. M. Bruynooghe. An interpreter for predicate logic programs: part I. Report CW 10, Applied Mathematics and Programming Division, Katholieke Universiteit, Leuven, Belgium, 1976.

16. Unpublished, but referenced in: M. Bruynooghe. The memory management of Prolog implementations, *Logic Programming* (eds. K. L. Clark & S.-Å. Tärnlund), Academic Press, New York, 1982. Other early implementations of tail-recursion optimization were due to Clark & McCabe (K. Clark, personal communication) and Warren (D. H. D. Warren. An improved PROLOG implementation which optimizes tail recursion. Proc. Int. Workshop on Logic Programming, von Neumann Comp. Sci. Soc., Debrecen, Hungary, 1980.)

17. D. H. D. Warren. Implementing PROLOG—compiling predicate logic programs. Research reports nos. 39 and 40, Dept. of Artificial Intelligence, Univ. of Edinburgh, 1977.

18. D. H. D. Warren. An abstract Prolog instruction set. Report 309, SRI International, Menlo Park, Calif., 1983.

19. S.-Å. Tärnlund. Logic information processing. Report TRITA-IBADB 1034, Dept. of Information Processing and Computer Science, The Royal Institute of Technology and the Univ. of Stockholm, Sweden, 1975.

20. K. L. Clark & F. McCabe. The control facilities of IC-Prolog. In *Expert Systems in the Micro-Electronic World* (ed. D. Michie), Edinburgh UP, 1979.

21. Å. Hansson, S. Haridi & S.-Å. Tärnlund. Some aspects of a logic machine prototype. In *Proc. Logic Programming Workshop* (ed. S.-Å. Tärnlund), Debrecen, Hungary, 1980.

22. K. L. Clark & S. Gregory. A relational language for parallel programming. In *Proc. ACM Conf. Functional Programming Languages and Computer Architecture*, 1981.

23. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM*18, no. 8, 1975, 453–457

24. K. L. Clark & S. Gregory. PARLOG: a parallel logic programming language. Report DOC 83/5, Dept. of Computing, Imperial College, London, 1983.

25. E. Y. Shapiro. A subset of Concurrent Prolog and its interpreter. ICOT Technical Report TR-003, Institute for New Generation Computing Technology, Tokyo, Japan, 1983.

26. K. Ueda, Guarded Horn Clauses. ICOT Technical Report TR-103, Institute for New Generation Computing Technology, Tokyo, Japan, 1985.

27. J. S. Conery & D. F. Kibler. Parallel interpretation of logic programs. In *Proc. ACM Conf. Functional Programming Languages and Computer Architecture*, 1981.

28. D. DeGroot. Restricted AND-parallelism. In *Proc. Int. Conf. Fifth Generation Computer Systems*, North-Holland, Amsterdam, 1984.

29. G. H. Pollard. Parallel execution of Horn clause programs. PhD Thesis, Imperial College, London, 1981.

CHAPTER 10

CONCLUSION

We have presented a novel compilation method for recursive logic programs. The method is based on Tärnlund's Reform system. We can view this system as an inference system that efficiently runs a transformation as computation. The program obtained by a Reform transformation represents a highly parallel algorithm for solving a particular goal. With our compilation method we arrive at this algorithm by changing the control strategy of the original program at compile-time. We compile recursion over inductively defined data structures (integers, lists, trees) to bounded iteration over vectors.

Our method gives parallel speedup where it is significant: in running recursive programs. Recursive programs are the primary target also for traditional AND-parallel systems—nobody is interested in using parallelism to speed up small computations, and large computations are specified by recursive programs. However, such systems cannot exploit the inherent parallelism in a recursive program, since the n recursive calls to the program must be made in sequence.

The performance of a parallel system is, of course, determined not only by the degree for parallelism it allows but also by the overheads imposed by the implementation. Our method allows simple and efficient implementation on parallel machines. Operations such as distribution of data, task scheduling and load balancing can be planned at compile-time. This is not the case in a traditional system where, therefore, these operations impose severe run-time overheads.

An added advantage of the method is that it can speed up programs even on sequential machines. The recursive calls in a traditional system involve setting up the arguments, dispatching to the right clause and, possibly, the allocation/deallocation of a binding environment. These operations correspond in our method to a decrement-and-jump-if-not-zero operation.

For these reasons we believe that the compilation method presented in this thesis offers an interesting alternative to earlier schemes for running logic programs in parallel.

REFERENCES

BARKLUND, J. (1990) Parallel unification. *Uppsala Theses in Computing Science 9*, Ph.D. thesis, Computing Science dept., Uppsala University, Uppsala.

CLARK, K. L. (1990) Logic programming schemes and their implementations. *UPMAIL Technical Report* 59. Computing Science department, Uppsala University, Uppsala.

CLARK, K. L. & TÄRNLUND, S.-Å. (1977) A first order theory of data and programs. In *Information Processing 77* (ed. B. Gilchrist), pp. 939–944. North-Holland, Amsterdam.

DEBRAY, S. K., N.-W. LIN & M. HERMENEGILDO (1990) Task granularity analysis in logic programs. Proc. ACM SIGPLAN'90 Conf. Programming Language Design and Implementation, pp. 174–188. New York.

KOWALSKI, R. A. (1974) Predicate logic as a computer language. In *Information Processing 74*, pp. 569–574. North-Holland, Amsterdam.

KUCK, D. J., Y. MURAOKA & S. CHEN. (1972) On the number of operations simultaneously executable in Fortran-like programs and their resulting speedup. *IEEE Trans. Computers* C-21, no. 12, 1293–1310.

ROBINSON, J. A. (1965) A machine-oriented logic based on the resolution principle. *Journal of the ACM* 12, 23–41.

ROBINSON, J. A. (1979) *Logic: Form and Function*. Edinburgh University Press, Edinburgh.

TÄRNLUND, S.-Å. (1975) Logic information processing. Report TRITA-IBADB 1034, Dept. of Information Processing and Computer Science, The Royal Institute of Technology and the University of Stockholm, Stockholm.

TÄRNLUND, S.-Å. (1990) Reform. Computing Science dept., Uppsala University, Uppsala.

WARREN, D. H. D. (1977) Implementing Prolog—compiling predicate logic programs. Research Reports 39 and 40, Dept. of AI, Univ. of Edinburgh, Edinburgh

WARREN, D. H. D. (1983) An abstract Prolog instruction set. Report 309, SRI International, Menlo Park, Calif.