A constraint $v = t$, where v is single, is rewritten to true. A constraint $u = v$, where v is a temporary variable and u is not, is rewritten to $v = u$. A constraint $v = t$, where neither v nor t are temporary, is rewritten to ($u = v$, $u = t$), where u is a new local variable.

A definition of the form

$$p(v_1, \ldots, v_N) := V : A$$

where i-1 is the parameter number of v_i, may be compiled to code of the form

allocate N
⟨save parameters⟩
⟨code for body⟩
deallocate
proceed

where N is the number of permanent variables. To each permanent variable is assigned a (unique) y-register y_i, for i in 0, …, N–1. Rules for assigning x-registers to temporary variables are given in Section 6.6.8, as constraints on the code generated. For a permanent variable, R_v stands for the y-register which it is already assigned. For a temporary variable, R_v serves as a place-holder, which will be substituted by an x-register.

To save the parameters $v_1, \ldots, v_n$, produce the code

get_variable R_{v1} x_{i1}
…
get_variable R_{vn} x_{in}

where i_j is the parameter number of v_j.

6.6.3 *Atoms*

A constraint atom

$u = v$

is compiled to

put_variable R_v R_u

if both are first, to

put_value R_v R_u

if first occurrence of u, to

get_variable R_v R_u

if first occurrence of v, and to

get_value R_v R_u

otherwise.

A constraint atom

$v = S$

where S is a symbol, is compiled to

put_symbol S R_v

if v is first, and to

get_symbol S R_v

otherwise.

A constraint atom

$v = f(t_1, \ldots, t_n)$

is compiled to

X_constructor F R_v
Y_1
…
Y_n

where F is f/n, X is put if v is first, otherwise get, and Y_i is

unify_void

if t_i is a variable and single,

unify_variable R_u

if t_i is a variable u and first,

unify_value R_u

if t_i is a variable u and not first, and

unify_symbol S

if t_i is a symbol S.

The constraint true produces no code.

A program atom

$p(v_1, \ldots, v_N)$

is compiled to

Y_1
…
Y_n
call p_N N

where p_N is the label of the code for the definition of p / N, and where Y_i is

put_void x_{i-1}

if v_i is single,

put_variable R_{vi} x_{i-1}

if v_i is first, and

put_value R_{vi} x_{i-1}

otherwise.

6.6.4 Choice

A definition of the form

$$p(v_1, \ldots, v_N) := (\langle \text{clause}_1 \rangle ; \cdots ; \langle \text{clause}_n \rangle)$$

where i-1 is the parameter number of v_i, may be compiled to code of the form

p_N: switch_on_tree L_v L_s L_c
L_v: ⟨try-retry-trust N $C_1, \ldots, C_n$⟩
L_s: ⟨try-retry-trust N $C_{i1}, \ldots, C_{ik}$⟩
L_c: ⟨try-retry-trust N $C_{j1}, \ldots, C_{jm}$⟩
C_1: ⟨code for clause_1⟩
…
C_n: ⟨code for clause_n⟩

where $C_1, \ldots, C_n$, L_v, L_s, and L_c are local names that only pertain to the code for the choice statement at hand. Clauses other than C_{i1} to C_{ik} are known to fail (in the guard) if the contents of x_0 is a symbol. Clauses other than C_{j1} to C_{jm} are known to fail (in the guard) if the contents of x_0 is a tree constructor.

A zero length try-retry-trust chain is given as

fail

a chain C_i, of length one, as

try_only C_i

and longer chains, $C_{i1}, \ldots, C_{ik}$ as

try C_{i1} N
retry C_{i2}
…
trust C_{ik}

The try-retry-trust instructions build, incrementally, the choice-box and guarded goals for a choice statement. The switch_on_tree instruction makes a first choice, eliminating clauses that are known to be incompatible. Better clause selection is possible, e.g., by decision graphs [Brand 1994]. The switch_on_tree instruction only illustrates the concept.

6.6.5 Clauses

A clause is very similar to a composition statement. The chunks of a clause are the chunks of the guard and of the body. The parameters are the parameters of the head of the definition in which they occur, which also gives the parameter numbers. The local variables are given in the hiding over the clause, which is otherwise ignored. The notions of first, single, permanent, and temporary are otherwise as for composition.

A clause may be compiled as

```
        allocate N
        ⟨save parameters⟩
        ⟨code for guard⟩
    guard_%
        ⟨code for body⟩
        deallocate
        proceed
```

with the same conditions as for composition.

6.6.6 Aggregates

The treatment of aggregates is somewhat unclean[1], but adequate since it lends itself to simple implementation (“worse is better” [Gabriel 1994]).

Observe that the variable created and stored in x_M is bound to the contents of x_i. The collect instructions will replace the value of this variable repeatedly. The variable itself will not be copied by choice splitting in the aggregate, since it is external.

A definition of the form

$$p(v_1, \ldots, v_M) := \text{aggregate}(u, A, w)$$

where i-1 is the parameter number of v_i, and w is v_j, may be compiled to code of the form

p_N: put_variable x_{M+1} x_M
        get_value x_{M+1} $x_{j\text{-}1}$
        try L_1 M+1
        trust L_2

L_1: allocate N_1+1
        get_variable y_{N1} x_M
        ⟨save parameters⟩
        ⟨code for A⟩
    guard_collect y_{N1} R_v $R_{v'}$
        ⟨code for *collect*(u, v, v')⟩
        deallocate
        proceed

L_2: allocate N_2+1
        get_variable y_{N2} x_M
    guard_unit y_{N2} R_v
        ⟨code for *unit*(v)⟩
        deallocate
        proceed

[1] Observe the appropriate section number.

The code generated corresponds to that for a choice with two special clauses. In the first are generated the solutions for *u* in A, which are collected by the guard, and the second ends with the unit when there are no more solutions.

The chunks of the collect clause are the chunks in A and in *collect*(u, v, v'). The chunks of the unit clause are the chunks in *unit*(v). Variables are classified and allocated accordingly, with the exception that v and v' have their first occurrences in the guard instructions and that parameters that occur in the unit are not temporary. N_1 and N_2 are the respective numbers of permanent variables.

6.6.7 Initial Statements

For simplicity, we assume that the initial statement is a program atom

initial(v_1, …, v_N)

This is no restriction, since it may be defined as anything.

```
        try_only L
L:      allocate N
        put_variable y0 x0
        …
        put_variable yN-1 xN-1
        call initial_N N
    guard_top
```

The guard_top instruction is given full freedom to report solutions for the variables in this statements, which are stored in y-registers, or other actions that would seem useful. For simplicity, we have given it the semantics of the execution model, where execution continues with other branches.

6.6.8 Register Allocation

The *lifetime* of a temporary variable is the sequence of instructions between, not including, the first and last instructions using R_v.

To *assign x-registers*, for each temporary variable v, select a register not used in its lifetime, and substitute it for R_v.

The objectives for this process are to minimise the number of x-registers used and to make possible the deletion of instructions as described in the next section

6.6.9 Editing

Having generated code according to the principles in preceding sections, some editing should be performed to improve execution efficiency.

The following instructions have no effect and can be deleted

get_variable x_i x_i

get_value x_i x_i

put_value x_i x_i

The instruction sequence

```
call L N
deallocate
proceed
```

should be replaced by

```
deallocate
execute L N
```

to ensure that tail recursive programs do not grow the and-stack.

The allocate instruction can be moved to just before the first instruction using a y-register, the first call, or the guard instruction, whichever comes first. The deallocate instruction can be moved to just after the last instruction using a y-register, the last call instruction, or the guard instruction, whichever comes last.

6.6.10 Two Examples

The familiar definition of append

```
append([], Y, Y).
append([H | X], Y, [H | Z]) :-
        append(X, Y, Z).
```

which is written as follows without syntactic sugar

```
append(X, Y, Z) :=
    (   X = [],
        Y = Z
    ?   true
    ;   H, X1, Z1 :
        X = [H | X1],
        Z = [H | Z1]
    ?   append(X1, Y, Z1) ).
```

can be translated to the following code using the principles above. Observe that ./2 is regarded as the functor for list constructors.

```
append_3:
        switch_on_tree Lv Ls Lc
Lv:     try C1 3
        trust C2
Ls:     try_only C1
Lc:     try_only C2

C1:     allocate 0
        get_symbol [] x0
        get_value x1 x2
    guard_nondeterminate
        deallocate
        proceed
```

```
C2:  allocate 3
     get_variable y1 x1
     get_constructor ./2 x0
     unify_variable x1
     unify_variable y0
     get_constructor ./2 x2
     unify_value x1
     unify_variable y2
  guard_nondeterminate
     put_value y0 x0
     put_value y1 x1
     put_value y2 x2
     deallocate
     execute append3
```

The guard instructions are given less indentation for readability.

In particular, code is generated for the second clause

```
     H, X1, Z1 :
     X = [H | X1],
     Z = [H | Z1]
  ?  append(X1, Y, Z1)
```

as follows. The variables X, Y, and Z, are parameters with numbers 0, 1, and 2, respectively. The variables X_1, Y, and Z_1 are classified as permanent and are assigned y-registers 0, 1, and 2, respectively. The variables X, H, and Z are temporary. The code before x-register allocation is

```
     allocate 3
     get_variable RX x0
     get_variable y1 x1
     get_variable RZ x2
     get_constructor ./2 RX
     unify_variable RH
     unify_variable y0
     get_constructor ./2 RZ
     unify_value RH
     unify_variable y2
  guard_nondet
     put_value y0 x0
     put_value y1 x1
     put_value y2 x2
     call append_3 3
     deallocate
     proceed
```

The variables X, H, and Z can now be allocated x-registers 0, 1, and 2, respectively, following the principles of lifetimes. Finally, two of the get_variable instructions can be deleted, and the call-deallocate-proceed instructions can be replaced by corresponding deallocate-execute instructions.

A call to bagof, such as the one discussed in Section 6.8.1,

p(L, S) := bagof(X, tail(X, L), S)

can be translated to the following code using the principles above

```
p_2:    put_variable x3 x2
        get_value x3 x1
        try L1 3
        trust L2

L1:     allocate 2
        get_variable y1 x2
        get_variable x1 x0
        put_variable y0 x0
        call tail_2 2
    guard_collect y1 x0 x1
        get_constructor ./2 x1
        unify_value y0
        unify_value x0
        deallocate
        proceed

L2:     allocate 1
        get_variable y0 x2
    guard_unit y0 x0
        get_symbol [] x0
        deallocate
        proceed
```

assuming that guard_collect is ordered. It is of course perfectly possible to provide both ordered and unordered versions of this instruction for use in different aggregates.

6.7 OPTIMISATIONS

This section lists a few optimisations that are available in the AGENTS system, and others that are believed to be obvious for future efficient implementations.

6.7.1 Flat Guards

Using only the simple machinery introduced so far, the execution speed of the AGENTS system is roughly a factor of four slower, for comparable programs without don't know nondeterminism, than a comparable Prolog implementation (such as SICStus Prolog with emulated code [Carlsson et al. 1993]), and much more memory is needed for execution. This is almost entirely due to the unnecessary creation of choice-nodes, and-nodes, and and-continuations for every choice statement.

However, a few simple instructions and notions of suspending and waking calls almost completely bridge the gap for a wide range of programs. The idea is to short-cut to promotion, suspension, or failure, for guards that only make simple tests on the first argument. This is a special case of the *flat* guards, that only contain constraints (with composition and hiding).

A *suspended call* has the attributes

- *parent*: a reference to an and-node
- *continuation pointer*: a reference to a sequence of instructions
- *a-registers*: a vector of trees
- *goals*: a pair of references to goals

Admit references to suspended calls in place of and-nodes in suspensions, in the wake stack, and in wake tasks.

To *suspend L with N arguments on a variable X*, create a suspended call with L as continuation pointer and N a-registers with contents from corresponding x-registers. Insert it at the insertion point, and add a suspension on the variable X referring to the suspended call.

To *proceed*, if the task is wake task referring to a suspended call, pop it. Install its parent and-node. Restore the program counter and x-registers from the suspended call, set the insertion pointer to its right sibling, unlink it, and decode instructions.

Add the following three instructions.

switch_on_constructor $L_f\, F_1\text{-}L_1 \ldots F_n\text{-}L_n$
switch_on_symbol $L_f\, S_1\text{-}L_1 \ldots S_n\text{-}L_n$

If x_0, dereferenced, is a constructor with functor F_i (is a symbol S_i), go to L_i. Otherwise, go to L_f.

suspend_call L N

Suspend L with N arguments on the variable in x_0. Proceed.

A definition such as

q(X) :=
        (X = a → true
        ; Y: X = f(Y) → q(Y))

which had to be encoded as

q_1:	switch_on_tree L_v L_s L_c
L_v:	try C_1 1
	trust C_2
L_s:	try_only C_1
L_c:	try_only C_2

```
C1:    allocate 0
       get_symbol a x0
    guard_nondet
       deallocate
       proceed

C1:    allocate 1
       get_constructor f/1 x0
       unify_variable y0
    guard_nondet
       put_value y0 x0
       deallocate
       execute q_1 1
```

can now be encoded as

```
q_1:   switch_on_tree Lv Ls Lc
Lv:    suspend_call q_1 1
Ls:    switch_on_symbol Lf a-C1
Lc:    switch_on_constructor Lf f/1-C2
Lf:    fail
C1:    proceed
C2:    get_constructor f/1 x0
       unify_variable x0
       execute q_1 1
```

Example	AGENTS without opt.	AGENTS with opt.	SICStus emulated
nreverse(300)	610 (4.5)	215 (1.6)	137 (1.0)
nreverse(1000)	7499 (2.9)	2433 (0.9)	2612 (1.0)
sort(medi)	394 (3.8)	250 (2.4)	105 (1.0)
sort(maxi)	8613 (4.1)	4031 (1.9)	2077 (1.0)

Figure 6.6. Performance of AGENTS w/wo optimisation vs. SICStus Prolog

The behaviour of the guards is entirely captured by the combination of switch instructions and the suspend_call instruction, and neither choice-nodes, and-nodes, nor and-continuations have to be created.

The impact on the performance of AGENTS for simple benchmarks where this coding of flat guards is applicable is shown in Figure 6.6. The comparison with SICStus only serves to indicate that the execution speed of AGENTS ends up in an acceptable order of magnitude.

The systems compared are AGENTS 0.9, with and without the above flat guard optimisation, and SICStus Prolog 2.1 #8 (emulated code) on a DECstation 5000/240. The times are in milliseconds and include the time for garbage collec-