```
                        JMP end_lb
              lb_N:     exptrans (N)
              end_lb:   ...
```

```
exptrans (M op N) :=  exptrans (M)
                      exptrans (N)
                      OP fail_lb
                      JMP end_lb
                fail_lb:  FORCE
                end_lb:   ...
```

op $\in$ { +, −, ∗, div, mod, $\leq$ } and the corresponding
OP $\in$ { ADD, SUB, MUL, DIV, MOD, LEQ }

```
exptrans (¬ B) := exptrans (B)
                  NOT
```

```
exptrans (B∧C) :=   exptrans (B)
                    JMF int_lb
                    exptrans (C)
                    JMP end_lb
              int_lb:   CONSFALSE
              end_lb:   ...
```

```
exptrans (B∨C) :=   exptrans (B)
                    JMT int_lb
                    exptrans (C)
                    JMP end_lb
              int_lb:   CONSTRUE
              end_lb:   ...
```

```
exptrans (M = N) := exptrans (M)
                    exptrans (N)
                    CHECKEQ
```

Final Configuration (Pr, atp_f, G)
with G (atp_f) = (TASK, argl, ACTIVE, locvl, pc, st, fptr, binf) and Pr (pc) = STOP.

graphtrans (c(t_1,...,t_n)) := *graphtrans* (t_1)
⋮
graphtrans (t_n)
CONSTRNODE (c, n)

exptrans: Expressions → Program

exptrans (X_i) := LOADX i

exptrans (c) := CONSTANT c

exptrans (true) := CONSTRUE

exptrans (false) := CONSFALSE

exptrans (n) := NUMBER n

exptrans (c(M_1,...,M_n)) := *exptrans* (M_1)
⋮
exptrans (M_n)
CONSTRNODE (c, n)

exptrans (f M_1 ... M_n) := *exptrans* (M_1)
⋮
exptrans (M_n)
if n = arity(f) then NODE (ca(f), n, n, k)
EVALUATE
else NODE (ca(f), n, program_arity(f), k)

exptrans (B → M) := *exptrans* (B)
JMT lb_M
FORCE
lb_M: *exptrans* (M)

exptrans (M M_1 ... M_n) := *exptrans*(M)
exptrans(M_1)
⋮
exptrans(M_n)
APPLY n
EVALUATE only if M $\notin FS_\Sigma \cup DC_\Sigma$

exptrans (B → M □ N) := *exptrans* (B)
JMF lb_N
exptrans (M)

```
                        ⋮
                        LOAD m
                        unifytrans (t_{n,m})
                        exptrans (body_n)
                        LASTRET
            label_n:    UNDO
                        FAILRET
```

unifytrans: Term → Program

unifytrans (X_i) := UNIFYVAR i

```
unifytrans (c) := UNIFYCONS (c, bd_lb)
                  JMP end_lb
          bd_lb:  graphtrans (c)
                  BIND
          end_lb: ...
```

```
unifytrans (n) := UNIFYNUMBER (n, bd_lb)
                  JMP end_lb
          bd_lb:  graphtrans (n)
                  BIND
          end_lb: ...
```

```
unifytrans (c (t_1,...,t_n)) :=   UNIFYCONSTR (c, n, bd_lb)
                                  unifytrans (t_1)
                                  ⋮
                                  unifytrans (t_n)
                                  JMP end_lb
                        bd_lb:    graphtrans (c (t_1,...,t_n))
                                  BIND
                        end_lb:   ...
```

graphtrans: Terms → Program

graphtrans (X_i) := LOADX i

graphtrans (true) := CONSTRUE

graphtrans (false) := CONSFALSE

graphtrans (c) := CONSTANT c

graphtrans (n) := NUMBER n

BAM Code for a BABEL Program:

```
0:          NODE objective 0 0 k_0
1:          EVALUATE
2:          PRINTRESULT
3:          MORE
4:          JMF end
5:          FORCE
            proctrans (PROC (f_1, m_1, k_1))
            ⋮
            proctrans (PROC (f_n, m_n, k_n))
objective:  BACKTRACK last_fail
            exptrans (OBJECTIVE (k_0))
last_fail:  PRINTFAILURE
end:        STOP
```

proctrans : Procedure → Program

PROC (f, m, k) = { f $t_{i,1}$ ... $t_{i,m}$:= $body_i(X_1,\ldots,X_k)$ with $1 \leq i \leq n$ }

```
proctrans (PROC (f, m, k)) :=  BACKTRACK label_1
                               LOAD 1
                               unifytrans (t_{1,1})
                               LOAD 2
                               unifytrans (t_{1,2})
                               ⋮
                               LOAD m
                               unifytrans (t_{1,m})
                               exptrans (body_1)
                               RET
                    label_1:   UNDO
                               BACKTRACK label_2
                               LOAD 1
                               unifytrans (t_{2,1})
                               ⋮
                               LOAD m
                               unifytrans (t_{2,m})
                               exptrans (body_2)
                               RET
                    label_2:   UNDO
                               ⋮
                    label_{n-1}: UNDO
                               BACKTRACK label_n
                               LOAD 1
                               unifytrans (t_{n,1})
```

SI ⟦LEQ fail_lb⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:num_1:num_2, fptr, binf)])
:= if G (num_1) = (NUMBER, n_1) and G (num_2) = (NUMBER, n_2)
then if $n_1 \leq n_2$
then (atp, G[atp|(TASK,argl,ACTIVE,locvl,pc+1,st:true_adr,fptr,binf)])
else (atp,G[atp|(TASK,argl,ACTIVE, locvl,pc+1,st:false_adr,fptr,binf)])
else (atp, G [atp | (TASK, argl, ACTIVE, locvl, fail_lb, st, fptr, binf)])

SI ⟦READTERM⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr, binf)])
:= let (G′, adr) = ReadTerm (G) in
(atp, G′[atp | (TASK, argl, ACTIVE, locvl, pc+1, st:adr, fptr, binf)])

SI ⟦MORE⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr, binf)])
:= if AskUser = true
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:true_adr, fptr, binf)])
else (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:false_adr, fptr, binf)])

ReadTerm and AskUser: Implementation dependant

SI ⟦PRINTRESULT⟧, SI ⟦PRINTFAILURE⟧, SI⟦STOP⟧: Highly implementation dependant (but they increase the program counter of the atp by one).

B Behaviour of the Machine

$\vdash$: Machine_State $\longrightarrow$ Machine_State
(Pr, atp, G,) $\vdash$ (Pr, atp′, G′)
if G (atp) = (TASK, argl, ACTIVE, locvl, pc, st, fptr, binf)
and Pr (pc) is defined
and SI ⟦Pr (pc)⟧ (atp, G) = (atp′, G′)

C Machine Specification

C.1 Translation Function

Babel Program := PROC (f_1, m_1, k_1)
⋮
PROC (f_n, m_n, k_n)
OBJECTIVE (k_0)

```
                else  if G (adr2) = (UBV)
                      then (true, G [ adr2 | (VAR, dereference (G, adr1))], tr:adr2)
                else  if G (adr2) = (VAR, ref)
                      then checkeq (G, adr1, ref, tr)
                else  if G (adr2) = (CONSTANT, c) or G (adr2) = (CONSTR, c', argl)
                      then (false, G, tr)
          else  if G (adr1) = (CONSTR, c, arg1:...:argm)
                then  if G (adr2) = (CONSTR, c', arg'1:...:arg'n)
                      then  if c = c' and m = n
                            then checkeqList (G, arg1:...:argm: arg'1:...:arg'n, tr)
                            else (false, G, tr)
                      else  if G (adr2) = (UBV)
                            then  if occur-check (G, adr2, adr1) = true
                                  then (true, G [adr2 | (VAR, dereference (G, adr1))], tr:adr2)
                                  else (false, G, tr)
                      else  if G (adr2) = (VAR, ref)
                            then checkeq (G, adr1, ref, tr)
                      else  if G (adr2) = (CONSTANT, c'') or G (adr2) = (NUMBER, n)
                            then (false, G, tr)
          else  if G (adr1) = (UBV)
                then  if occur-check (G, adr1, adr2) = true
                      then (true, G [adr1 | (VAR, dereference (G, adr2))], tr:adr1)
                      else (false, G, tr)
          else  if G (adr1) = (VAR, ref)
                then checkeq (G, ref, adr2, tr)

checkeqList : Graph × Graph_Address* × Graph_Address* × Trail →
              {true, false} × Graph × Trail

checkeqList (G, l1, l2, tr) :=
    if l1 = l2 = ε
       then (true, G, tr)
       else  let checkeq (G, hd (l1), hd (l2), tr) = (bval, G', tr') in
                if bval = true
                   then  checkeqList (G', tl (l1), tl (l2), tr')
                   else  (false, G, tr)
```

Routine *occur_check* as usual.

```
SI [[ADD fail_lb]]
    (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:num1:num2, fptr, binf)])
    :=  if G (num1) = (NUMBER, n1) and G (num2) = (NUMBER, n2)
        then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:new(G,1), fptr, binf),
                            new(G,1) | (NUMBER, n1 + n2)])
        else (atp, G [atp | (TASK, argl, ACTIVE, locvl, fail_lb, st, fptr, binf)])
```

The other arithmetic operations SUB, MUL, DIV, and MOD are defined analogously.

SI ⟦JMF lb⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr, fptr, binf)])
:= if G (*dereference*(adr)) = (CONSTANT, bval)
then if bval = false
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, lb, st, fptr, binf)])
else if bval = true
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st, fptr, binf)])

SI ⟦NOT⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr, fptr, binf)])
:= if G (*dereference* (adr)) = (CONSTANT, true)
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:false_adr, fptr, binf)])
else if G (*dereference* (adr)) = (CONSTANT, false)
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:true_adr, fptr, binf)])

SI ⟦CHECKEQ⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr_1:adr_2, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
:= let (bval, G′, tr′) = *checkeq* (G, adr_1, adr_2, tr) in
if bval = true
then (atp, G′[atp | (TASK, argl, ACTIVE, locvl, pc+1, st:true_adr, fptr,
(tr′, bptr, lspt, bad, spc, sst, str))])
else (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:false_adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])

checkeq : Graph × Graph_Address^2 × Trail → { true, false } × Graph × Trail

checkeq (G, adr_1, adr_2, tr)
:= if adr_1 = adr_2
then (true, G, tr)
else if G (adr_1) = (CONSTANT, c)
then if G (adr_2) = (CONSTANT, c′)
then if c = c′
then (true, G, tr)
else (false, G, tr)
else if G (adr_2) = (UBV)
then (true, G [adr_2 | (VAR, *dereference* (G, adr_1))], tr:adr_2)
else if G (adr_2) = (VAR, ref)
then *checkeq* (G, adr_1, ref, tr)
else if G (adr_2) = (NUMBER, n) or G (adr_2) = (CONSTR, c″, argl)
then (false, G, tr)
else if G (adr_1) = (NUMBER, n)
then if G (adr_2) = (NUMBER, n′)
then if n = n′
then (true, G, tr)
else (false, G, tr)

else {n ≠ n′, unification fails}
(atp, G [atp | (TASK, argl, ACTIVE, locvl, bad, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else if G (adr) = (UBV)
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, bd_lb, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else if G (adr) = (VAR, ref)
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:*dereference*(adr), fptr,
(tr, bptr, lspt, bad, spc, sst, str))])

SI [[UNIFYVAR i]]
(atp, G [atp | (TASK, argl, ACTIVE, $locv_1$:...:$locv_i$:...:$locv_k$, pc, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
:= if G ($locv_i$) = (UBV)
then (atp, G [atp | (TASK, argl, ACTIVE, $locv_1$:...:$locv_i$:...:$locv_k$, pc+1, st,
fptr, (tr:$locv_i$, bptr, lspt, bad, spc, sst, str)),
$locv_i$ | (VAR, *dereference* (G, adr))])

dereference : Graph × Graph_Address → Graph_Address

dereference (G, adr) := if G (adr) = (VAR, ref)
then *dereference* (G, ref) else adr

SI [[BIND]]
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:vadr:gadr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
:= (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st, fptr,
(tr:vadr, bptr, lspt, bad, spc, sst, str)),
vadr | (VAR, *deference* (gadr)])

SI [[UNDO]]
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr,
(tv_1:...:tv_l, bptr, lspt, bad, spc, sst, str))])
:= (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, ε, fptr,
(ε, bptr, atp, bad, spc, sst, str)),
tv_1 | (UBV),...,tv_l | (UBV)])

SI [[JMP lb]]
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr, binf)])
:= (atp, G [atp | (TASK, argl, ACTIVE, locvl, lb, st, fptr, binf)])

SI [[JMT lb]]
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr, fptr, binf)])
:= if G (*dereference*(adr)) = (CONSTANT, bval)
then if bval = true
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, lb, st, fptr, binf)])
else if bval = false
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st, fptr, binf)])

then (atp, G [atp | (TASK, argl, ACTIVE, locvl, bad, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else *dobacktrack* (atp, G)

SI ⟦UNIFYCONS (c, bd_lb)⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
:= if G (adr) = (CONSTANT, c′)
then if c = c′
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else {c ≠ c′, unification fails}
(atp, G [atp | (TASK, argl, ACTIVE, locvl, bad, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else if G (adr) = (UBV)
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, bd_lb, adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else if G (adr) = (VAR, ref)
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:*dereference*(adr),
fptr, (tr, bptr, lspt, bad, spc, sst, str))])

SI ⟦UNIFYCONSTR (c, m, bd_lb)⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
:= if G (adr) = (CONSTR, c′, $arg_1 : \ldots : arg_n$)
then if c = c′ and m = n
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:arg_n:…:arg_1, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else {c ≠ c′ or m ≠ n, unification fails}
(atp, G [atp | (TASK, argl, ACTIVE, locvl, bad, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else if G (adr) = (UBV)
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, bd_lb, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else if G (adr) = (VAR , ref)
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:*dereference*(adr), fptr,
(tr, bptr, lspt, bad, spc, sst, str))])

SI ⟦UNIFYNUMBER (n, bd_lb)⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
:= if G (adr) = (NUMBER, n′)
then if n = n′
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])

:= if lspt = atp {No sons}
and (tr = ε or bptr = fptr) {No trail or backtracking to the father}
then (fptr, G [fptr | (TASK, argl′, ACTIVE, locvl′, pc′, st′:adr, fptr′,
(tr′ : tr, bptr′, bptr, bad′, spc′, sst′, str′))])
else {Normal Return}
(fptr, G [atp | (TASK, argl, EVALUATED, locvl, pc+1, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str)),
fptr | (TASK, argl′, ACTIVE, pc′, st′:adr, fptr′,
(tr′, bptr′, lspt′, bad′, spc′, sst′, str′))])

SI [[FAILRET]]
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str)),
fptr | (TASK, argl′, ACTIVE, locvl′, pc′, st′, fptr′,
(tr′, bptr′, lspt′, bad′, spc′, sst′, str′)),
bptr | (TASK, argl″, ACTIVE, locvl″, pc″, st″, fptr″,
(tr″, bptr″, lspt″, bad″, spc″, sst″, str″))])
:= if bptr = fptr {Backtracking to the father. No brothers}
then (fptr, G [fptr | (TASK, argl′, ACTIVE, locvl′, bad′, st′, fptr′,
(tr′, bptr′, fptr, bad′, spc′, sst′, str′))])
else {Backtracking to a deeper node}
let (atp′, G′) = *dobacktrack* (bptr, *undotrail* (G, tr′, str″)) in
(atp′, G′[fptr | (TASK, argl′, ACTIVE, locvl′, spc″, sst″, fptr′,
(str″, bptr′, bptr, bad′, spc′, sst′, str′))])

undotrail : Graph×Local_Trail×Local_Trail → Graph

undotrail (G, tr, tr′) := if tr = tr′ then G
else if tr = tr″:v then undotrail (G [v | (UBV)], tr″, tr′)

dobacktrack : Graph_Address × Graph → Graph_Address × Graph

dobacktrack (ptr, G [ptr | (TASK, argl, status, locvl, pc, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str)),
lspt | (TASK, argl′, EVALUATED, locvl′, pc′, st′, ptr,
(tr′, bptr′, lspt′, bad′, spc′, sst′, str′))])
:= if ptr = lspt
then (ptr, G [ptr | (TASK, argl, ACTIVE, locvl, pc, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
else let G′ = *undotrail* (G, tr, str′) in
dobacktrack (lspt, G′ [ptr | (TASK, argl, ACTIVE, locvl, spc′, sst′, fptr,
(str′, bptr, lspt, bad, spc, sst, str))])

SI [[FORCE]]
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
:= if atp = lspt {TASK without sons}

else (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:n_0, fptr, binf),
n_0 | (FUNCTION, a_1:. . .:a_n, caf, m-n, k)])
where n_0 n_1 . . . n_k = new(G,k)

SI ⟦BACKTRACK lb⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
:= (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st, fptr,
(tr, bptr, lspt, lb, spc, sst, str))])

SI ⟦EVALUATE⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])
:= if G (adr) = (TASK, argl′, DORMANT, locvl′, pc′)
then (adr, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st, fptr,
(tr, bptr, adr, bad, spc, sst, str)),
adr | (TASK, argl′, ACTIVE, locvl′, pc′, ε, atp,
(ε, lspt, adr, –, pc+1, st, tr))])
else {G(adr) = (FUNCTION, . . .)}
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str))])

SI ⟦APPLY i⟧
(atp, G [atp | (TASK, argl, ACTIVE, locl, pc, st:adr: a_1:. . .:a_i, fptr, binf),
adr | (FUNCTION, alist, caf, j, k)])
:= if j = i
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:n_0, fptr, binf),
n_0 | (TASK, alist:a_1:. . .:a_i, DORMANT, n_1:. . .:n_k, caf),
n_1 | (UBV),. . .,n_k | (UBV)])
else (atp, G [atp | (TASK, argl, ACTIVE, locl, pc+1, st:n_0, fptr, binf),
n_0 | (FUNCTION, alist : a_1:. . .:a_i, caf, j–i, k)])
where n_0 n_1 . . . n_k = new(G,k)

SI ⟦RET⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str)),
fptr | (TASK, argl′, ACTIVE, locvl′, pc′, st′, fptr′, binf)])
:= (fptr, G [atp | (TASK, argl, EVALUATED, locvl, pc+1, st, fptr,
(tr, bptr, lspt, bad, spc, sst, str)),
fptr | (TASK, argl′, ACTIVE, locvl′, pc′, st′:adr, fptr′, binf)])

SI ⟦LASTRET⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:adr, fptr,
(tr, bptr, lspt, bad, spc, sst, str)),
fptr | (TASK, argl′, ACTIVE, locvl′, pc′, st′, fptr′,
(tr′, bptr′, lspt′, bad′, spc′, sst′, str′))])

new : Graph $\times \mathbb{N} \rightarrow$ Graph_Address*
new(G, k) := $a_1 \ldots a_k$ where $a_0 := 0$
$a_i := \min\{a \mid G(a)$ is undefined and $a_i > a_{i-1}\}$ for $i > 0$

A.3 Semantics of the Instructions (Semantic Function)

SI : Instructions $\times$ Graph_Address $\times$ Graph $\rightarrow$ Graph_Address $\times$ Graph

SI ⟦LOAD i⟧
(atp, G [atp | (TASK, arg_1:...:arg_i:...:arg_m, ACTIVE, locvl, pc, st, fptr, binf)])
:= (atp, G [atp | (TASK, arg_1:...:arg_i:...:arg_m, ACTIVE, locvl, pc+1, st:arg_i, fptr, binf)])

SI ⟦LOADX i⟧
(atp, G [atp | (TASK, argl, ACTIVE, $locv_1$:...:$locv_i$:...:$locv_k$, pc, st, fptr, binf)])
:= (atp, G [atp | (TASK, argl, ACTIVE, $locv_1$:...:$locv_i$:...:$locv_k$, pc+1, st:$locv_i$, fptr, binf)])

SI ⟦CONSTANT c⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr, binf)])
:= (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:constant(c), fptr, binf)])

SI ⟦ NUMBER n⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr, binf)])
:= (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:new(G,1), fptr, binf), new(G,1) | (NUMBER, n)])

SI ⟦CONSTRNODE (c, m)⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:a_1:...:a_m, fptr, binf)])
:= (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:new(G,1), fptr, binf), new(G,1) | (CONSTR, c, a_1:...:a_m)])

SI ⟦CONSTRUE⟧
(atp, g [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr, binf)])
:= (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:true_adr, fptr, binf)])

SI ⟦CONSFALSE⟧
(atp, g [atp | (TASK, argl, ACTIVE, locvl, pc, st, fptr, binf)])
:= (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:false_adr, fptr, binf)])

SI ⟦NODE (caf, m, k, n)⟧
(atp, G [atp | (TASK, argl, ACTIVE, locvl, pc, st:a_1:...:a_m, fptr, binf)])
:= if m = n
then (atp, G [atp | (TASK, argl, ACTIVE, locvl, pc+1, st:n_0, fptr, binf),
n_0 | (TASK, a_1:...:a_m, DORMANT, n_1:...:n_k, caf),
n_1 | (UBV),...,n_k | (UBV)])

(b) Dormant_Task_Information := { DORMANT }
× Graph_Address*
(pointers to local variables)
× Program_Address
(address of the first line of the function or predicate code)

(c) Active_Task_Information := { ACTIVE, EVALUATED }
× Graph_Address* (local variables)
× Program_Address (program counter)
× Local_Stack (father pointer)
× Graph_Address
× Backtracking_Information

(d) Local_Stack := Graph_Address*

(e) Backtracking_Information := Local_Trail
× Graph_Address^2
(backtracking pointer and last son pointer)
× Program_Address^2
× Local_Stack
× Local_Trail
(backtracking address and safe copies)

(f) Local_Trail := Graph_Address*

A.2 States of the Machine

Machine_State := Program × Graph_Address × Graph where

(a) Program := Program_Address → Instructions

(b) Program_Address := $\mathbb{N}$

Initial Configuration: (Pr, atp_0, G_0) where

- atp_0, true_adr, false_adr ∈ Graph_Address,
- constant : Constants → Graph_Address,
- true_adr, false_adr and constant(c) are different Graph addresses for each c ∈ Constants.
- $G_0(atp_0)$ = (TASK, ε, ACTIVE, ε, 0, ε, –, (ε, –, atp_0, –, –, ε, ε))
- G_0(constant (c)) = (CONSTANT, c) for all c ∈ Constants
- G_0(true_adr) = (CONSTANT, true), G_0(false_adr) = (CONSTANT, false)

6. Arithmetic_Instructions := { ADD fail_lb, SUB fail_lb, MUL fail_lb,
   DIV fail_lb, MOD fail_lb, LEQ fail_lb |
   fail_lb ∈ Program_Address }

7. Additional_Instructions := { STOP, PRINTRESULT, PRINTFAILURE,
   MORE, READTERM }

A.1.2 Data Area

1. Graph := Graph_Address $-\!\!\rightarrow$ Nodes

   (a) Graph_Address := $\mathbb{N}$

   (b) Nodes := Term_Nodes ∪ Variable_Nodes
   ∪ Function_Nodes ∪ Task_Nodes

2. Term_Nodes := Constant_Nodes
   ∪ Constructor_Nodes
   ∪ Number_Nodes

   where

   (a) Constant_Nodes := { CONSTANT } ×
   { c | c ∈ Constructors, arity (c) = 0 }

   (b) Constructor_Nodes := { CONSTR } ×
   { c | c ∈ Constructors, arity (c) > 0 } ×
   Graph_Address*

   (c) Number_Nodes := { NUMBER } × Integer

3. Variable_Nodes := { UBV } (unbound variables)
   ∪ { VAR } × Graph_Address (bound variables)

4. Function_Nodes := { FUNCTION }
   × Graph_Address* (Arguments)
   × Program_Address
   × $\mathbb{N}$ (Number of missing arguments)
   × $\mathbb{N}$ (Number of local variables)

5. Task_Nodes := { TASK }
   × Graph_Address* (Argument addresses)
   × Status_Information

   (a) Status_Information := Dormant_Task_Information
   ∪ Active_Task_Information

Appendix

A Formal Specification of the BABEL Machine

A.1 Basic Components

A.1.1 Instructions

$$
\begin{aligned}
\text{Instructions} := \quad & \text{Local_Stack_Instructions} \\
\cup \quad & \text{Graph_Instructions} \\
\cup \quad & \text{Unification_Instructions} \\
\cup \quad & \text{Control_Instructions} \\
\cup \quad & \text{Process_Instructions} \\
\cup \quad & \text{Arithmetic_Instructions} \\
\cup \quad & \text{Additional_Instructions}
\end{aligned}
$$

1. $\text{Local_Stack_Instructions} := \{ \text{NOT, CHECKEQ} \},$

2. $$
\begin{aligned}
\text{Graph_Instructions} := \quad & \{ \text{LOAD i, LOADX i} \mid i \in \mathbb{N} \}, \\
\cup \quad & \{ \text{CONSTANT c} \mid c \in \text{Constants} \} \\
\cup \quad & \{ \text{CONSTRNODE (c, arity)} \mid \\
& \qquad c \in \text{Constructors}, \text{arity} \in \mathbb{N} \} \\
\cup \quad & \{ \text{NUMBER n} \mid n \in \text{Integer} \} \\
\cup \quad & \{ \text{NODE (ca, numarg, arity, locals)} \mid \\
& \qquad \text{ca} \in \text{Program_Address}, \\
& \qquad \text{arity, locals, numarg} \in \mathbb{N} \} \\
\cup \quad & \{ \text{APPLY i} \mid i \in \mathbb{N} \} \\
\cup \quad & \{ \text{CONSTRUE, CONSFALSE} \}
\end{aligned}
$$

3. $$
\begin{aligned}
\text{Unification_Instructions} := \quad & \{ \text{UNIFYCONS (c, bd_lb)} \mid \\
& \qquad c \in \text{Constants}, \\
& \qquad \text{bd_lb} \in \text{Program_Address} \} \\
\cup \quad & \{ \text{UNIFYCONSTR (c, arity, bd_lb)} \mid \\
& \qquad c \in \text{Constructors}, \\
& \qquad \text{arity} \in \mathbb{N}, \\
& \qquad \text{bd_lb} \in \text{Program_Address} \} \\
\cup \quad & \{ \text{UNIFYNUMBER (n, bd_lb)} \mid n \in \text{Integer} \\
& \qquad \text{bd_lb} \in \text{Program_Address} \} \\
\cup \quad & \{ \text{UNIFYVAR i} \mid i \in \mathbb{N} \} \\
\cup \quad & \{ \text{BIND , UNDO} \}
\end{aligned}
$$

4. $\text{Control_Instructions} := \{ \text{JMP lb, JMT lb, JMF lb} \mid \text{lb} \in \text{Program_Address} \}$

5. $$
\begin{aligned}
\text{Process_Instructions} := \quad & \{ \text{BACKTRACK lb} \mid \text{lb} \in \text{Program_Address} \} \\
\cup \quad & \{ \text{EVALUATE, RET, LASTRET,} \\
& \quad \text{FAILRET, FORCE} \}
\end{aligned}
$$

[Turner 85] D.A.Turner: *Miranda: A non-strict functional language with polymorphic types*, Proceedings of the ACM Conference on Functional Languages and Computer Architecture 1985, Lecture Notes in Computer Science 201, Springer Verlag 1985, pp. 1–16.

[Warren 83] D.H.D.Warren: *An Abstract PROLOG Instruction Set*, Technical Note 309, SRI International, Menlo Park, California, October 1983.

References

[Bellia, Levi 86] M. Bellia, G. Levi: *The Relation between Logic and Functional Languages*, Journal of Logic Programming, Vol.3, 1986, pp. 217–236.

[Bosco et al. 88] P.G. Bosco, E. Giovannetti, C. Moiso: *Narrowing versus SLD–resolution*, Theoretical Computer Science 59, 1988, pp. 3–23.

[Damas, Milner 82] L. Damas and R. Milner: *Principal type schemes for functional programs*, ACM Symposium on Principles of Programming Languages, 1982, pp. 207–212.

[DeGroot, Lindstrom 86] D.DeGroot, G.Lindstrom: *Logic Programming: Functions, Relations, Equations*, Prentice Hall 1986.

[van Emden, Yukawa 87] M.H.van Emden, K.Yukawa: *Logic Programming with Equations*, Journal of Logic Programming, Vol.4, 1987, pp. 265–288.

[Hermenegildo 86] M.V.Hermengildo: *An Abstract Machine for Restricted AND-parallel Execution of Logic Programs*, 3rd Conf. on Logic Programming, Lecture Notes in Computer Science, Springer Verlag 1986, pp. 25–39.

[Loogen et al. 89] R.Loogen, H.Kuchen, K.Indermark, W.Damm: *Distributed Implementation of Programmed Graph Reduction*, Proc. Conference on Parallel Architectures and Languages Europe 1989, Lecture Notes in Computer Science, Springer Verlag 1989, pp. 136–157.

[Milner 78] R. Milner: *A theory of type polymorphism in programming*, Journal of Computer and System Sciences, 17(3), 1978, pp. 348–375.

[Moreno, Rodríguez 88] J.J.Moreno-Navarro, M.Rodríguez-Artalejo: *BABEL: A functional and logic programming language based on constructor discipline and narrowing*, In: I.Grabowski, P.Lescanne and W.Wechler (eds.), *Algebraic and Logic Programming*, Lecture Notes in Computer Science 343, Springer Verlag, 1989.

[Moreno, Rodríguez 89] J.J.Moreno-Navarro, M.Rodríguez-Artalejo: *Logic Programming with Functions and Predicates: The Language BABEL*, Journal of Logic Programming 1989, to appear.

[Moreno 89] J.J.Moreno-Navarro: *Diseño, semántica e implementación de BABEL, un lenguaje que integra la programación funcional y lógica*, Ph.D. Thesis, Facultad de Informática UPM, Madrid, July 1989, (in spanish).

[Reddy 85] U.S.Reddy: *Narrowing as the Operational Semantics of Functional Languages*, Proceedings of the IEEE International Symposium on Logic Programming, IEEE Computer Society Press, July 1985, pp. 138-151.

[Reddy 87] U.S.Reddy: *Functional Logic Languages, Part I*, Proceedings of a Workshop on Graph Reduction, Lecture Notes in Computer Science 279, Springer Verlag 1987, pp. 401-425.

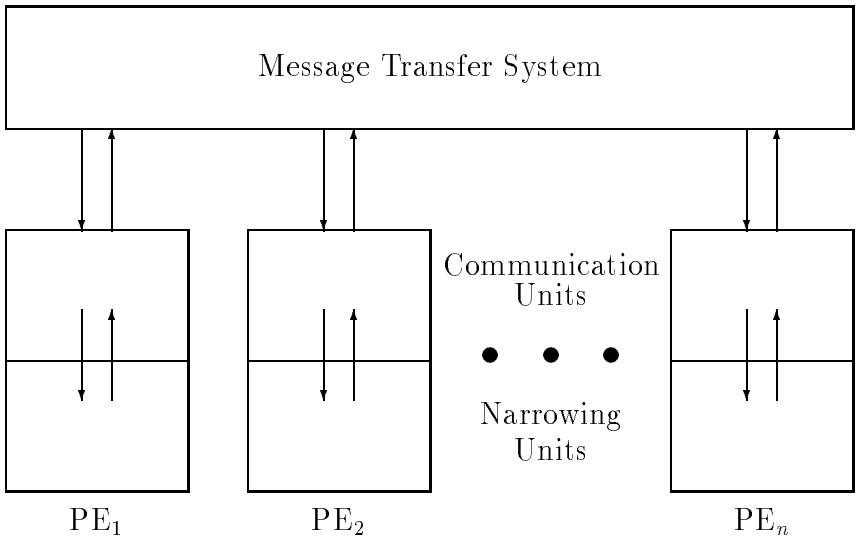

Figure 3: Global Structure of the Parallel $\mathcal{BAM}$

communicating by exchanging messages. Tightly coupled architectures are not considered since they are limited to a small number of processors.

The parallel $\mathcal{BAM}$ will have a structure (see figure 3) similar to the parallel abstract machine PAM [Loogen et al. 89], which has been developped for the parallel implementation of functional languages on a loosely coupled network of processors.

Each processing unit of the PAM contains a communication unit and a reduction unit. The communication units are responsible for the exchange of messages, while each reduction unit represents a sequential graph reducer that has been extended for the integration in a parallel machine. In the parallel $\mathcal{BAM}$ each reduction unit is replaced by a narrowing unit, which is an analogous extension of the sequential $\mathcal{BAM}$.

The parallelization of the $\mathcal{BAM}$ is more complicated than the parallel implementation of a functional language. The reason is the occurrence of side effects caused by logic variables. Thus, more synchronization between parallel processes is needed. In the literature, AND- and OR-parallel implementations of logic programs are investigated. AND-parallelism can be seen as a parallel execution of the arguments of a function. In PROLOG this function is the AND-operation, in BABEL arbitrary functions can be used (like in functional languages). OR-parallelism uses a parallel execution of the different rules for each function symbol.

We are currently working on an AND- and on an OR-parallel version of the $\mathcal{BAM}$, that will be implemented on a transputer system (like the PAM). The AND-parallel version uses a generalization of the restricted AND-parallelism (RAP) approach of [Hermenegildo 86].

A task node is no longer needed as soon as the existence of no more alternatives to try is noticed (execution of a FAILRET). But this situation can be found out still earlier, when the task finishes the execution of the last alternative (RET command of the last rule). If the information stored in a task node is no longer useful the node can be disposed. By making a compromise between time and memory efficiency, one could dispose the node of a task that finishes the execution of the last alternative if

- the task has no sons, and
- – its trail is empty or
  – the task to be re-evaluated in case of failure is the father (its trail is appended to the trail of the father).

This optimization could be handled by a new instruction LASTRET that is used in the translation of the last rule of a function.

5.3 Garbage Collection

A *garbage collection* mechanism has been implemented too. Task nodes (and, hence, their variable nodes) are only disposed when one of the situations described before happens. For terminal nodes and number nodes a classical reference counter system is used.

5.4 Results

The current state of the implementation is the following. A Pascal version has been developped to test the behaviour of the machine. A more efficient implementation has been done in OCCAM to prepare the development of parallel distributed versions.

The early results show a good behaviour in time (similar to Prolog programs) and a little bit worse in memory. Hence, the optimizations described before are very important. More optimizations need to be included in future versions, for instance a detection of purely functional computations (generating smaller task nodes by avoiding the backtracking information), special treatment of tail recursion and specific instructions for lists.

6 Discussion

To cope with infinite objects and non-strict functions it is necessary to change the evaluation strategy of $\mathcal{BAM}$. The development of a lazy $\mathcal{BAM}$ which does not evaluate arguments unless they are needed, is in progress. This machine, on which we will report in a forthcoming paper, evaluates needed arguments of functions only up to head normal form. Sharing of graph nodes is more important for the lazy machine than for the innermost machine.

Another important research subject is the development of a parallel BABEL machine. In order to simplify the parallelization of $\mathcal{BAM}$, we have chosen a very decentralized structure for $\mathcal{BAM}$. Since we are interested in using large networks of processors, we prefer a loosely coupled architecture, which consists of several processors with local memory

restoring is made along the searching path of the next node to re-evaluate in case of backtracking.

The computation of the previous example shows why these copies are needed. The call to function f creates a task (T_0) that is executed. In order to generate the code for the body of f the first argument of the call to h (constant a) is stored on the local stack. After this, a call to predicate p is done which generates calls to predicates q and r. The first value for p is true binding X to a.

The next step is to continue the execution of the $\mathcal{BAM}$ code for the body of f. The second argument of h is a conditional expression. The "then" expression is selected evaluating g (where evaluation of k is demanded) returning a to the local stack of T_0. Remember that another a was stored on this stack. These two values are eliminated from the stack to construct the task (h a a). But this task fails (there is no rule for it).

The first task to be re-evaluated is T_5 (for k). In the case that the value of this function changes, we would need to restore the local stack of T_0 to preserve the arguments of h (the first a on the stack).

In our example the backtracking to task T_5 fails as well as the backtracking to T_4 (g). The condition (p X) is re-evaluated. If the boolean value changed, the program counter of T_0 needs to be modified to execute again the code for selecting between the "then" or the "else" branch of the conditional. The backtracking to T_3 (r) also fails, and backtracking to T_2 is needed. For this purpose it is necessary to unbind the variable X because T_2 has bound it, i.e. to undo the binding of X noted in the trail of T_2.

5 Implementation

In order to implement the BABEL Machine we have designed some extensions and optimizations.

The extensions suppose the inclusion of new nodes: nodes for constants (created at the beginning of the execution and shared for all the nodes that use them) and nodes for numbers. The number nodes lead to new instructions (ADD, SUB, MUL, DIV, MOD) to execute operations, $+$, $-$, $*$, div, mod which can be used in BABEL programs as well as integer literals. The two most important optimizations are explained in the following.

5.1 Temporal variables

A *temporal variable* of a rule is a variable that occurs on the left hand side but not inside a construction (i.e. it is directly the argument of the function). Temporal variables do not need a variable node for themselves. The argument can be used directly instead of the variable. For instance, variable F in the map program is temporal. We can avoid the unification of variable 1 with argument 1 and use the instruction LOAD 1 instead of LOADX 1 in the translation of the body.

5.2 Disposition of task nodes

The second optimization deals with the *disposition of task nodes* and is related with the reusability of "binding frames" in the Warren abstract machine.

The following example should clarify this description. Consider the following rules of a BABEL program

$$f\ X := h\ a\ ((p\ X) \rightarrow (g\ X) \Box (l\ X)).$$
$$p\ X := (q\ X) \wedge (r\ X).$$
$$q\ a := \text{true}.$$
$$q\ b := \text{false}.$$
$$r\ a := \text{true}.$$
$$g\ X := k\ X.$$
$$l\ X := b.$$
$$k\ a := a.$$
$$h\ a\ b := a.$$

and the objective $(f\ X)$.
The following picture indicates the graph structure with the different pointers:

Pointers:

Father Pointer,
Backtracking Pointer,
Last son pointer,
Implicit Stack

Tasks	T_0	T_1	T_2	T_3	T_4	T_5	T_6
Function symbols	f	p	q	r	g/l	k	h

Following the policy that a task is not removed from the stack until all its alternatives have been tried it is clear that no answer is lost.

But some other informations for handling backtracking are needed in task nodes. The *backtrack address* is the address of the program where the task must continue in case of backtracking (i.e. the address of the next rule). The rest of the backtracking information is a safe copy of the state of the father: a copy of the program counter, the stack and the local trail (in fact we only need a pointer to the top of the local trail in this moment). It is used to restore the state of the father in case of backtracking, undoing all the bad decisions. The copy is made during the execution of the EVALUATE command and the

<table>
<tr><td rowspan="2">local trail</td><td rowspan="2">backtracking pointer</td><td rowspan="2">last son pointer</td><td rowspan="2">backtracking address</td><td colspan="3">safe copies of</td></tr>
<tr><td>program counter</td><td>local stack</td><td>local trail</td></tr>
<tr><td colspan="4"></td><td colspan="3">of the father</td></tr>
</table>

Figure 2: Backtracking information

is executed. The local trail was described in section 3.1. Next, there are two pointers to task nodes in the graph: a so called *backtracking pointer* to the 'previous brother' (the node generated previously by the father) and a so called *last son pointer* to the 'last son' (the last son generated by the task).

The backtracking pointer is used when a task fails — there is no possibility of returning a value — to select the task that must then be re-evaluated. If a task is the first son of its father the backtracking pointer is the address of the father. Otherwise the backtracking pointer points at the task node that was activated by the father node just before this task.

The last son pointer is used to initialize the backtracking pointer of newly generated subtasks. It is also used to find the task to re-evaluate in case of backtracking. Of course, at any moment, the address of the active task is the last son pointer of its father.

Using these pointers one can determine an implicit stack of nodes that reflects the order in which nodes have been activated. The control of the BABEL machine is based on the following ideas:

- A task returning with success (RET command) gives the control to the father which continues its execution.

- A task returning with failure (FAILRET command) gives the control to the previous task in the implicit stack.

The recursive description of this stack in terms of the pointers in the graph is:

- The top of the stack is the right-most bottom-most element of the tree (following the last son pointer of all the nodes, beginning with the very first task generated).

- The predecessor of each task is:

  1. its father if its backtracking pointer points to the father. The task is the first son.

  2. Otherwise the predecessor node is the first node of the stack below the node indicated by its backtracking pointer. The predecessor can be determined by doing one step along the backtracking pointer and then following all the last son pointers.

then, and control is given back to the calling task, which has to handle the failure:

```
fail_lab:  UNDO
           FAILRET
```

These instructions complete the translation of the procedure for map. Since no other procedures are available, the translation of the objective follows. This translation is done analogously to the translation of the right hand side of a rule. In fact, the objective can be seen as a definition of a nullary function.
For example for the objective map (+ 1) [X,2] the following code is produced:

```
obj:       BACKTRACK last_fail
           CONSTRNODE (1, 0)
           NODE (+, 1, 2, 2)
           LOADX 1
           CONSTRNODE (2, 0)
           CONSTRNODE (cons, 2)
           NODE (map, 2, 2, 3)
           EVALUATE
           RET
last_fail: PRINTFAILURE
           STOP
```

For the translation of a rule, the following translation schemes are used:

- *unifytrans* : $\mathbf{Term}_\Sigma \rightarrow \mathcal{BAM}$-**Code** generates code, which unifies an argument of the actual task with the corresponding term on the right hand side of a rule.
- *graphtrans* : $\mathbf{Term}_\Sigma \rightarrow \mathcal{BAM}$-**Code** produces code, which builds a graph for a term, which is to be bound to an unbound variable in an argument of the actual task. *graphtrans* is used by *unifytrans*.
- *exptrans* : $\mathbf{Exp}_\Sigma \rightarrow \mathcal{BAM}$-**Code** produces code, which evaluates an expression to normal form (in particular the right hand side of a rule).

The execution of a $\mathcal{BAM}$-program prog starts with the following initial configuration:

$$(\text{prog}, \text{atp}, G_0)$$

where G_0 is a graph which contains only one node representing the initial task (referenced by atp). This initial task is an active task node with the program counter initialized to 0 and its local stack and local trail are empty. The other informations are not needed.

4 Backtracking

This section describes and justifies the backtracking information needed in task nodes. The content of the backtracking information that has been described in section 3.1 can be seen in figure 2. All this information is generated each time an EVALUATE command

```
nextrule:  UNDO
           BACKTRACK fail_lab
           LOAD 1                              % translation of the lhs
           UNIFYVAR 1                          % unify with first argument
           LOAD 2
           UNIFYCONST (cons, 2, bindlab2)      % unify with second argument
           UNIFYVAR 2
           UNIFYVAR 3
           JMP continue2
bindlab2:  LOADX 2
           LOADX 3
           CONSTRNODE (cons, 2)
           BIND
continue2: exptrans([F X | map F Xs])          % translation of rhs
fail_lab:  ...
```

First, the bindings produced by the first rule are deleted (by UNDO). Then, the backtracking label "fail_lab" is stored in the actual task node. After this, the arguments of the actual task are unified with the corresponding terms on the left hand side of the rule. For the first argument, this is exactly the same as before. For the second argument, it is more complicated, since now the term consists of an application of the binary constructor cons to two variables. If the second argument points at a constructor node for the binary constructor cons, the UNIFYCONST command now leaves pointers to both substructures of this argument on the stack. They are unified with the corresponding parts of the terms on the left hand side of the rule (here using UNIFYVAR commands). If the second argument is an unbound variable, this variable is bound to the appropriate constructor term (by the four commands following bindlab2). The translation of the body of the rule looks like this:

```
continue2: LOADX 1
           LOADX 2
           APPLY 1
           EVALUATE
           LOADX 1
           LOADX 3
           NODE (map, 2, 2, 3)
           EVALUATE
           CONSTRNODE (cons, 2)
           RET
```

First, the local variable F is applied to the local variable X, and the resulting task is evaluated. Furthermore, a task for map with the local variables F and Xs as arguments is created and evaluated. Both results are combined by a node for the binary constructor cons which points at them. After this result is computed, control is given back to the calling task by RET.

If the second and last rule also fails, a jump to the command referenced by the current backtrack label "fail_lab" is performed. All bindings produced by the rule are deleted

```
map:       BACKTRACK nextrule
           LOAD 1                          % translation of lhs
           UNIFYVAR 1                      % unify with first argument
           LOAD 2
           UNIFYCONST (nil, 0, bindlab)    % unify with second argument
           JMP continue
bindlab:   CONSTRNODE (nil, 0)
           BIND
continue:  CONSTRNODE (nil, 0)             % translation of rhs
           RET
nextrule:  ...
```

This code causes the following actions. First the label "nextrule" is stored which is used when the rule fails or backtracking is forced by the user. Then, the arguments of the actual task are loaded onto the stack and unified with the corresponding terms of the rule. To unify the first argument with the variable F, a simple UNIFYVAR instruction is sufficient. The unification of the second argument is more complicated. First it is checked (by UNIFYCONST), whether the argument is the constructor nil. If this is the case, the next instruction is executed, which causes a jump to the code for the right hand side. All arguments have now been unified with the corresponding terms on the left hand side of the rule, and hence, the rule is applicable.

If the argument starts with a constructor different from nil, the rule cannot be applied and a jump to the command referenced by the backtrack label (nextrule) is executed to try the next rule.

Otherwise, the argument must be an unbound variable. To apply the rule, this variable must be bound to the constructor nil. Hence, the UNIFYCONST instruction causes a jump to the code (starting at bindlab) which installs this binding. This code consists of a CONSTRNODE instruction, which leaves a pointer to the desired constructor node on the stack, and a BIND instruction, which overwrites the unbound variable by a variable node which points to this constructor node. The first and third possible actions of the UNIFYCONST command correspond to the read and write mode of the WAM [Warren 83] respectively.

The right hand side of the first rule is very simple. A constructor node with tag nil representing the empty list is produced (by CONSTRNODE) as the result, and control is given back to the calling task.

After the first rule, the second rule

```
map F [X | Xs] := [F X | map F Xs]
```

is translated:

$$\begin{array}{l}\text{PROC}(f_1, m_1, k_1)\\ \ldots\\ \text{PROC}(f_n, m_n, k_n)\\ \text{OBJECTIVE}(k_0)\end{array}$$

where $\text{PROC}(f_i, m_i, k_i)$ denotes the set of rules defining symbol f_i with program-arity m_i and k_i local variables ($1 \leq i \leq n$). k_0 is the number of variables of the objective. The machine code for a BABEL program is the following:

$$\begin{array}{rl}0: & \text{NODE (obj, 0, 0, } k_0\text{)}\\ 1: & \text{EVALUATE}\\ 2: & \text{PRINTRESULT}\\ 3: & \text{MORE}\\ 4: & \text{JMF end}\\ 5: & \text{FORCE}\\ & \textit{proctrans } (\text{PROC}(f_1, m_1, k_1))\\ & \qquad\qquad \ldots\\ & \textit{proctrans } (\text{PROC}(f_n, m_n, k_n))\\ \text{obj}: & \text{BACKTRACK last_fail}\\ & \textit{exptrans } (\text{OBJECTIVE}(k_0))\\ \text{last_fail}: & \text{PRINTFAILURE}\\ \text{end}: & \text{STOP.}\end{array}$$

The first code generates a task node for the objective, starts evaluation, and prints the result of the program after a successful evaluation. If the programmer asks for more solutions, the FORCE-instruction is executed and the task of the objective is forced to backtrack.

After this preliminary code, the translation of the procedures follows. This translation is done using the *proctrans* scheme. Finally, code for the objective is produced, which stores the label to which to backtrack in case of a failure, executes the objective, and reports a failure, if it occurs.

In the following we explain the translation of BABEL programs by means of the map example (see section 2). In the map example, we have only one procedure consisting of two rules.

We start with the translation of the first rule:

$$\text{map } F\ [\,] := [\,],$$

for which the following code is produced:

The 'UNIFYVAR i'-instruction binds the i-th local variable, that will be unbound when executing this instruction due to the linearity restriction of the BABEL-rules, to the graph whose address is on top of the stack. The implementation of an occur-check and a general unification algorithm is not needed at this place due to the left linearity.

The UNDO-instruction is used to delete variable bindings in case of backtracking.

3.2.4 Control Instructions

Control instructions are jump instructions. The unconditional 'JMP label' sets the program counter of the active task to label. The conditional jump instructions 'JMT label' and 'JMF label' cause a jump only if the boolean value true and false, respectively, is represented by the top element of the stack.

3.2.5 Process Instructions

The activation and termination of tasks is controlled by the process instructions EVALUATE and RET, respectively. The EVALUATE-instruction performs a subroutine call to the dormant task whose address is given on top of the stack. The RET-instruction is executed when a task terminates successfully and the control can be given back to the father task.

For the organization of backtracking the instructions 'BACKTRACK label' and 'FAILRET' are necessary. The BACKTRACK-instruction initializes the backtracking information of a task. FAILRET will be executed when a task fails, i.e. no solution can be produced. The 'predecessor' of the task must then be reactivated and forced to evaluate in a different way. For more details, see the explanation of backtracking in section 4.

To control the behaviour of the machine on the top level, some more instructions are needed.

- The instruction 'MORE' asks the user if more solutions are to be searched.
- The instruction 'FORCE' forces the last successfully terminated task to backtrack and thus to compute more solutions.
- The instruction 'PRINTFAILURE' finishes the whole execution with the output 'no (more) solutions have been found'. The instruction 'PRINTRESULT' is used to output a solution which consists of the result value and bindings of the local variables within the objective.

Before we discuss the organization of backtracking in detail, we show the translation of BABEL-programs into machine code by compiling the small map-example of section 2.

3.3 Compilation of BABEL-Programs

A BABEL program consists (mainly) of a set of rules and an expression (called the objective), which is to be evaluated using the rules. The rules are grouped according to the symbol they define. Hence, a BABEL program looks like this:

3.2.2 Graph Instructions

Graph instructions are used for the graph manipulation. By LOAD-instructions the addresses of arguments of the active task (LOAD i) or pointers to local variables (LOADX i) can be pushed onto the local stack, i.e. the stack within the task node of the active task.

Constructor nodes are generated by execution of the instruction

$$\text{CONSTRNODE } (c,\ m),$$

where c is the constructor name and m is the number of components whose addresses are taken from the local stack and replaced by the address of the new node.

Task and function nodes are created by the instruction

$$\text{NODE (ca, numarg, arity, locals)},$$

which has four parameters: the code address of the function, the number of arguments of the function application that are given on the stack, the program arity of the function, and finally the number of local variables. If enough arguments are given, i.e. numarg = arity, a dormant task node will be constructed. Otherwise, a function node will be created.

To add further arguments to a function node, we use the instruction APPLY i that expects i graph addresses and a pointer to a function node on top of the stack. The i graph addresses are added to the argument list of the function node. If this yields a complete application, a dormant task node is generated. Otherwise, a new function node is built. The existing function node must not be overwritten because of the possibility of sharing.

3.2.3 Unification Instructions

For the organization of unification we need the following instructions.

'UNIFYCONST (c, arity, label)' tries to unify the top element of the local stack with the constructor c. The top element of the data stack points at a constructor node or at a variable node.

In the case of a constructor node with constructor name c the argument list of this node is copied onto the stack. Further unification steps will be controlled by the code that follows the UNIFYCONST-instruction. If the constructor name in the node is different from c backtracking occurs.

If the top of the stack points at an unbound variable node, this variable must be bound to a term whose top level constructor will be c. In this case a graph representation for the term must be constructed. The address of the code that generates such a graph representation is given as the third parameter (label) of the UNIFYCONST-instruction. Thus in the case of an unbound variable the program counter of the active task is set to 'label'. After the construction of the graph the binding of the unbound variable to the new graph is performed by executing the instruction BIND, that expects a pointer to an unbound variable node and a pointer to a constructor node on top of the stack and that binds the variable to the term graph whose root is the constructor node.

Figure 1 indicates the structure of the different graph nodes. The computation is controlled by the *task nodes* which represent applications.

Each task node contains the address of the first line of code for the corresponding function symbol, pointers to the graph representation of the arguments and a list of pointers to unbound variable nodes which represent local variables. Depending on the status of a task — we distinguish *dormant*, *active* and *evaluated* tasks — additional information is provided within the task node. The *status-information* of active and evaluated tasks consists of a local stack which is needed for the organization of data manipulations, a program counter which indicates the next instruction and a pointer to the father node, i.e. the node that created the current node. This father pointer is used when a task finishes successfully and the control has to be returned to the father task.

Since BABEL is also a logic language, tasks must be able to perform backtracking. For this reason, active and evaluated task nodes contain certain *backtracking* information. This consists of a local trail which keeps track of variable bindings to be removed in case of backtracking; pointers to the 'previous brother' and the 'last son', which help to find the task to be reactivated when backtracking occurs; a backtracking address, which points to the code that must be executed if backtracking reactivates the task; and safe copies of the father's program counter, local stack and local trail, that are needed to restore the machine state in case of backtracking. All this will be explained in detail in section 4, where the organization of backtracking will be discussed.

In addition to the task nodes the graph contains *terminal* and *variable nodes*. Terminal nodes are constructor or function nodes.

Constructor nodes represent structured data. They contain the constructor name and a list of pointers to the graphs of the components of the structure.

Function nodes represent functional data which always correspond to partial function applications. Consequently a function node contains essentially the same base information as a task node (address of first line of code of the function, partial list of arguments, number of local variables) and additionally the number of arguments that are necessary to make the function application complete.

Variable nodes are needed for the organization of unification. We distinguish nodes for unbound and bound variables. Unbound variable nodes consist only of a tag (UBV). When a variable is bound to some terminal node the graph address of this node is written into the bound variable node which we indicate by the tag VAR. No more information needs to be stored for variables.

3.2 Machine Instructions

This subsection gives a short discussion of the machine instructions of $\mathcal{BAM}$. Five classes of machine instructions are distinguished.

3.2.1 Stack Instructions

Stack instructions are needed for the efficient implementation of some of the primitive functions of BABEL, especially for the logic negation (NOT) and the equality operator (CHECKEQ).

- Task Nodes:

TASK	argument list	status	local variables	code address
status-information (only if status = active or evaluated)				
backtracking information (only if status = active or evaluated)				

- Terminal Nodes:
  - Constructor Nodes:

CONSTR	constructor name	pointers to components

  - Function Nodes:

FUNCTION	partial argument list	code address
number of missing arguments	number of local variables	

- Variable Nodes:
  - Unbound Variable Nodes: UBV
  - Bound Variable Nodes: VAR | Graph-address

Figure 1: Structure of Graph Nodes

- the *graph*, which may contain task-, variable- and terminal nodes, and
- the *active task pointer* which points at the task node which corresponds to the currently executed procedure call.

Thus, we define

$$\mathcal{BAM} := \langle \underbrace{\mathrm{St}}_{\text{Store}}, \underbrace{\vdash}_{\substack{\text{Transition}\\ \text{relation}}} \rangle$$

where

$$\mathrm{St} := \mathrm{Program_store} \times \mathrm{Active_task_pointer} \times \mathrm{Graph}.$$

A state of the machine is usually denoted by (p,atp,G), where p $\in$ Program_store, atp $\in$ Active_task_pointer and G $\in$ Graph.

3.1 The Graph Component

The graph component is modelled as a mapping from graph addresses into the graph nodes:

$$\mathrm{Graph} := \mathrm{Graph_addresses} \rightarrow \mathrm{Graph_nodes}, \text{ where}$$
$$\mathrm{Graph_nodes} := \mathrm{Task_nodes} \cup \mathrm{Terminal_nodes} \cup \mathrm{Variable_nodes}.$$

This formal definition clearly forces the innermost behaviour of evaluation, as well as the evaluation of conditions before the evaluation of their corresponding "then" and "else" parts. Expressions M such that $\vdash M \rightarrow_\sigma N$ is not *derivable* for any σ and N are either final results with status *nf* or failures with status *fl*. Final results are printed out by the BABEL machine, while failures force it to backtrack.

Some features of the machine behaviour are not specified by the formal definition of innermost narrowing. We briefly refer to some of them.

The BABEL machine tries the program's rules in their textual ordering and evaluates arguments from left to right; it backtracks when a failure or a user's request for alternative solutions occurs.

Primitive symbols are handled as if the user had introduced them through their predefined rules. However, this is not exactly so in all cases: The evaluation of the second member in conjunctions and disjunctions is avoided whenever possible, and strong equality is implemented through unification for the sake of efficiency. This implementation works fine with any equality $t_1 = t_2$ between two terms whose unification either succeeds or finitely fails; but it does not work properly in cases such as $X = c(Y)$, which allow for infinitely many outcomes with result false and different answers, according to the predefined rules of strong equality. The actual implementation will ignore these outcomes. The user can overcome this limitation by explicitly programming his own equality. The price to pay will be a risk of nontermination.

There is also a declarative — i.e. logical — semantics for first-order BABEL, related to a lazy version of narrowing through soundness and completeness results; cfr. [Moreno, Rodríguez 89], [Moreno 89].

3 Structure of the BABEL Machine

The BABEL machine is a sequential abstract graph narrowing machine by which the functional logic language BABEL will be implemented using an innermost evaluation strategy. This strategy has been chosen for simplicity in order to develop a first version of an abstract machine for BABEL.

The machine has a very decentralized organization. The main component of the machine is a graph, which contains, among others, so called task nodes which correspond to ordinary activation records but contain much more information about a procedure call[3]. Thus, a task node contains e.g. a local data stack for data manipulations and a local program counter. For the organization of backtracking a local trail is needed. It is used in the same way as in the Warren Abstract Machine (WAM) [Warren 83] to keep track of variable bindings which must be removed in the case of backtracking.

The use of local data stacks and local trails will simplify a later parallelization of the machine, i.e. the incorporation of the machine in a parallel environment.

The store of the BABEL machine consists of three components:

- the *program store* which contains the translations of the BABEL rules into machine code,

[3] A procedure consists of all program rules corresponding to one function symbol.

R_1. No calculus rule. Final result. $(st\ t = nf)$.

R_2.

$$\frac{\vdash M_i \rightarrow_\sigma N_i}{\vdash c(M_1, \ldots, M_i, \ldots, M_n) \rightarrow_\sigma c(M_1\sigma, \ldots, N_i, \ldots, M_n\sigma)}$$

(a rule scheme for each $1 \leq i \leq n$)

R_3. No calculus rule. Failure. $(st\ (XM_1 \ldots M_k) = fl)$.

R_4. Let m be the program-arity of f. We distinguish three subcases:

$R_{4.1}$. $k < m$. No calculus rule in this case. $(fM_1 \ldots M_k)$ is either a final result (if it has status nf) or a failure.

$R_{4.2}$. $k \geq m$, and $st\ M_i = nw$ for some $1 \leq i \leq m$. Then we have

$$\frac{\vdash M_i \rightarrow_\sigma N_i}{\vdash (fM_1 \ldots M_i \ldots M_k) \rightarrow_\sigma (f(M_1\sigma) \ldots N_i \ldots (M_k\sigma))}$$

(a rule scheme for each $1 \leq i \leq m$ such that $st\ M_i = nw$)

$R_{4.3}$. $k \geq m$, and $st\ M_i = nf$ for all $1 \leq i \leq m$. Then, for each variant $f\ t_1 \ldots t_m := R$ of a BABEL rule in the program, which shares no variables with $(fM_1 \ldots M_k)$ and whose lhs unifies with $(fM_1 \ldots M_m)$ with m.g.u. $\sigma \cup \theta$ (where σ binds variables in $(fM_1 \ldots M_m)$ and θ binds variables in $(f\ t_1 \ldots t_m)$) we have

$$\vdash (fM_1 \ldots M_m M_{m+1} \ldots M_k) \rightarrow_\sigma (R\theta)(M_{m+1}\sigma) \ldots (M_k\sigma)$$

R_5. Let M be a guarded or a conditional expression, Then:

$$\frac{\vdash M \rightarrow_\sigma N}{\vdash (M\ M_1 \ldots M_k) \rightarrow_\sigma (N\ M_1\sigma \ldots M_k\sigma)}$$

R_6. Let ε stand for the empty substitution. Then, we have:

$$\vdash\ (\text{true} \rightarrow M)\ \rightarrow_\varepsilon\ M$$

and, for B syntactically different from 'true' and 'false'

$$\frac{\vdash (B \rightarrow_\sigma C)}{\vdash (B \rightarrow M) \rightarrow_\sigma (C \rightarrow M\sigma)}$$

R_7. We have:

$$\vdash (\text{true} \rightarrow M_1 \square M_2)\ \rightarrow_\varepsilon\ M_1 \quad \text{and} \quad \vdash (\text{false} \rightarrow M_1 \square M_2)\ \rightarrow_\varepsilon\ M_2$$

and, for B syntactically different from 'true' and 'false'

$$\frac{\vdash (B \rightarrow_\sigma C)}{\vdash (B \rightarrow M_1 \square M_2) \rightarrow_\sigma (C \rightarrow M_1\sigma \square M_2\sigma)}$$

The outcome consists of result true and answer $\{X/4, Y/5\}$. Some predefined rules for $+$, $=$ and $\wedge$ have been used.

Now, as preparation for our formal definition, we observe that any well-typed Σ-expression belongs to exactly one of the following kinds:

1.	t	% term
2.	$c(M_1, \ldots, M_n)$	% construction, $c/n \in DC_\Sigma$
3.	$(X M_1 \ldots M_k)$	% X higher-order variable of type arity $n \geq k \geq 0$
4.	$(f\ M_1 \ldots M_k)$	% f function symbol of type arity $n \geq k \geq 0$
5.	$(M\ M_1 \ldots M_k)$	% M guarded or conditional expression
6.	$(B \to M)$	% guarded expression
7.	$(B \to M_1 \Box M_2)$	% conditional expression

The reason is that in any well-typed application (MN), M must be either a variable, a function symbol, a guarded or a conditional expression or another application.

What follows depends implicitly on some fixed BABEL program. The *status*, $st\ M$, of a BABEL expression M can be *normal form* (*nf*), *narrowable* (*nw*) or *failure* (*fl*) and is defined by recursion on M's syntactical structure, taking care of the 7 kinds of expressions:

St_1. $st\ t\ :=\ nf$

St_2. $st\ c(M_1, \ldots, M_n)\ :=$ **if** $st\ M_i = nf$ for all $1 \leq i \leq n$ **then** nf
else if $st\ M_i = nw$ for some $1 \leq i \leq n$ **then** nw **else** fl

St_3. $st\ (X\ M_1 \ldots M_k)\ := fl$

St_4. Let m be the program-arity of f.

$st\ (f\ M_1 \ldots M_k) :=$ **if** $st\ M_i = nw$ for some $1 \leq i \leq m \leq k$ **then** nw
else if $st\ M_i = nf$ for all $1 \leq i \leq m \leq k$ and some rule
in the BABEL program is applicable **then** nw
else if $k < m$ **then** nf **else** fl

where a BABEL-rule is said to be applicable iff it has some variant sharing no variables with $f\ M_1 \ldots M_k$ and whose lhs $f\ t_1 \ldots t_m$ unifies with $f\ M_1 \ldots M_m$.

St_5. $st\ (M\ M_1 \ldots M_k) = st\ M$ if M is a guarded or conditional expression.

St_6. $st\ (B \to M)\ :=$ **if** $st\ B = nf$ or $st\ B = nw$ **then** nw **else** fl

St_7. $st\ (B \to M_1 \Box M_2)\ :=$ **if** $st\ B = nf$ or $st\ B = nw$ **then** nw **else** fl

Now we are in a position to define a formal calculus for deriving statements of the form

$$\vdash M \to_\sigma N$$

where M, N are BABEL expressions and σ is a finite substitution of some first order variables occurring in M by terms. By definition, we regard $M \to_\sigma N$ as an allowed innermost narrowing step iff $\vdash M \to_\sigma N$ can be derived in the formal calculus. We group the rules of the calculus according to the 7 kinds of expressions:

form by means of narrowing. This means that the lhs of rules for the defined and predefined function symbols are unified with appropriate subexpressions, which are then replaced by the corresponding instance of the rule's rhs. This process is repeated until a normal form N is reached. Then, N is taken as the *result* of the evaluation, and all bindings of variables occurring in M that have been accumulated during the reduction are regarded as the *answer*, similarly as in PROLOG. The combination of result and answer will be called *outcome* in the sequel.

The restrictions imposed on higher-order variables are worth to be noted carefully. Higher-order variables may occur in the lhs of rules, but are forbidden to occur free in either rhs of rules or goals. This means that they are used only for rewriting, as in applicative functional programming.

For simplicity, we restrict ourselves in this paper to an innermost narrowing strategy. Of course, this implies that infinite objects cannot be used. Before we give a formal definition, we show innermost narrowing-evaluation working at the small map example given above. The notation

$$M \rightarrow_\sigma N$$

is intended to mean that M evaluates to N in *one* innermost narrowing step, while binding some of the variables occurring in M to terms as indicated by the finite substitution σ. We also write

$$M \stackrel{*}{\rightarrow}_\sigma N$$

to indicate the result of *several* narrowing steps, where σ is the composition of the substitutions involved in the single steps, restricted to the variables occurring in M. We now show the evaluation of a goal for the map program. The redex at each step is underlined, and some intermediate steps (corresponding to predefined functions) are skipped.

$$\begin{array}{ll}
 & \underline{\text{map } (+\ 2)[X,3]} = [6,Y] \\
\rightarrow & [\underline{(+\ 2\ X)} \mid \text{map } (+\ 2)[3]] = [6,Y] \\
\stackrel{*}{\rightarrow} & [s^2(X) \mid \underline{\text{map } (+\ 2)[3]}] = [6,Y] \\
\rightarrow & [s^2(X), \underline{(+\ 2\ 3)} \mid \text{map } (+\ 2)[\,]] = [6,Y] \\
\stackrel{*}{\rightarrow} & [s^2(X), 5 \mid \underline{\text{map } (+\ 2)[\,]}] = [6,Y] \\
\rightarrow & \underline{[s^2(X), 5] = [6,Y]} \\
\rightarrow & \underline{(s^2(X) = 6)} \wedge ([5] = [Y]) \\
\stackrel{*}{\rightarrow}_{\{X/4\}} & \underline{([5] = [Y])} \\
\rightarrow & (\underline{(5 = Y)} \wedge ([\,] = [\,])) \\
\stackrel{*}{\rightarrow}_{\{Y/5\}} & \text{true}
\end{array}$$

- Rules for the boolean operations

$\neg$ false	:=	true	false $\wedge$ Y	:=	false	false $\vee$ Y	:=	Y
$\neg$ true	:=	false	true $\wedge$ Y	:=	Y	true $\vee$ Y	:=	true

- Rules for strong equality

$(c = c) := \text{true}$ % $c/0 \in DC_\Sigma$, constant
$(c(X_1, \ldots, X_n) = c(Y_1, \ldots, Y_n)) := (X_1 = Y_1) \wedge \ldots \wedge (X_n = Y_n)$
% $c/n \in DC_\Sigma$
$(c = d(Y_1, \ldots, Y_n)) := \text{false}$ % $c/0 \in DC_\Sigma$, $d/n \in DC_\Sigma$, $n > 0$
$(c(X_1, \ldots, X_n) = d) := \text{false}$ % $c/n \in DC_\Sigma$, $d/0 \in DC_\Sigma$, $n > 0$
$(c(X_1, \ldots, X_n) = d(Y_1, \ldots, Y_m)) := \text{false}$ % $c/n \in DC_\Sigma$,$d/m \in DC_\Sigma$, different

The rules for $\wedge, \vee$ reflect the sequential character of these connectives. The rules for strong equality must be used respecting the types of constructors. They specify that an expression $(M_1 = M_2)$ will evaluate to true if M_1, M_2 evaluate both to the same term, and will evaluate to false if M_1, M_2 evaluate to different terms. $(M_1 = M_2)$ will be undefined if the evaluation of M_1 or M_2 does not terminate.

BABEL, as described in [Moreno, Rodríguez 89], supports infinite terms through lazy evaluation. This gives rise to a more sophisticated behaviour of strong equality.

As a last remark on BABEL programs, let us mention that pure PROLOG can be straightforwardly translated to BABEL. For instance, the PROLOG program

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs,Ys,Zs).
```

can be translated as follows:

```
typevar A.
data list(A) = nil | cons (A, list(A)).
fun  append : list(A) → list(A) → list(A) → bool.
     append [ ] Ys Zs := (Zs = Ys) → true.
     append [X|Xs] Ys [Z|Zs] := (Z=X ∧ (append Xs Ys Zs)) → true.
```

The idea is that PROLOG clauses translate into guarded BABEL rules whose body is identical to true. Strong equality must be used to ensure left linearity. Boolean valued functions play the role of predicates. In fact, a concrete BABEL implementation could allow a PROLOG-like syntax as a syntactical sugar, translating PROLOG-like programs to pure BABEL, cfr. [Moreno, Rodríguez 89].

Of course, append can also be programmed as a function in BABEL. The point is that the just given version behaves like the PROLOG append predicate under BABEL's evaluation mechanism.

2.2 Narrowing Semantics

A goal for a given BABEL Σ-program is any Σ-expression M which includes no higher-order variables. To solve a goal, the BABEL machine tries to reduce it to a normalized

Each defining rule for f must have the form

$$\underbrace{f\ t_1 \ldots t_m}_{\text{lhs}} := \underbrace{\underbrace{\{B \rightarrow\}}_{\substack{\text{guard}\\(\text{optional})}}\ \underbrace{M}_{\text{body}}}_{\text{rhs}}$$

for some $m \leq n$ (called the *arity* of the rule) and satisfy the following restrictions:

1. *Flatness*: $t_i \in \mathbf{Term}_\Sigma$.
2. *Left Linearity*: $f\ t_1 \ldots t_n$ does not contain multiple variable occurrences.
3. *Well-Typedness*: Under appropriate type assumptions for the variables we may infer the types τ_i for the terms t_i $(1 \leq i \leq m)$, the type *bool* for the guard B and the type $\tau_{m+1} \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ for the body M.
4. *Restrictions on free variables*: Any variable that occurs in the rhs but not in the lhs is called *free*. Occurrences of free variables are allowed in the guard, but not in the body. Moreover, free variables must be first-order. By this we mean that their types must be constructed types (under the type assumption used to well-type the rule).

In any program, all rules for a fixed f must have the same arity. This is called the *program-arity* of f, and is less or equal than f's type-arity. Programs are also required to satisfy a nonambiguity condition:

5. *Nonambiguity*: Given any two rules for the same function symbol f:

$$\begin{array}{lcl} f\ t_1 \ldots t_m & := & \{B \rightarrow\}M \\ f\ s_1 \ldots s_m & := & \{C \rightarrow\}N \end{array}$$

one of the three following cases must hold:

   (a) *No superposition*: $f\ t_1 \ldots t_m$ and $f\ s_1 \ldots s_m$ are not unifiable.

   (b) *Fusion of bodies*: $f\ t_1 \ldots t_m$ and $f\ s_1 \ldots s_m$ have a most general unifier (m.g.u.) σ such that $M\sigma, N\sigma$ are identical[2].

   (c) *Incompatibility of guards*: $f\ t_1 \ldots t_m$ and $f\ s_1 \ldots s_m$ have a m.g.u. σ such that $(B \wedge C)\sigma$ is unsolvable.

This depends on a notion of unsolvability that must be chosen decidable and such that unsolvable boolean expressions cannot yield the value true under any valuation of their variables; cfr. [Moreno, Rodríguez 89].

We assume some predefined rules for the primitive function symbols. Among them, we have

[2] As usual, $M\,\sigma$ denotes the expression M where all variables are replaced according to σ.

We assume that Σ contains the predefined types:

- *bool* with data constructors true, false and function symbols

$$\begin{array}{lcll} \neg & : & \mathit{bool} \to \mathit{bool} & \text{(logic negation)} \\ \wedge & : & \mathit{bool} \to \mathit{bool} \to \mathit{bool} & \text{(sequential and)} \\ \vee & : & \mathit{bool} \to \mathit{bool} \to \mathit{bool} & \text{(sequential or)} \end{array}$$

  and

- *nat* with data constructors 0 and s (successor function) and the usual arithmetic functions.

A special primitive function symbol is the strong equality symbol $=$ with type $\alpha \to \alpha \to$ *bool* whose definition will be given later.

In example programs we declare constructed types and data constructors in a MIRANDA-like style, cfr. [Turner 85].
We distinguish the following syntactic domains:

- *variables* ranged over by X, Y, Z ... $\in$ **Var**,
- *terms* ranged over by s, t, ... $\in \mathbf{Term}_\Sigma$:

$$\begin{array}{rcll} t & ::= & X & \% \text{ variable} \\ & | & c & \% \text{ constant (nullary constructor, } c/0 \in DC_\Sigma) \\ & | & c(t_1, \ldots, t_n) & \% \ c/n \in DC_\Sigma, \text{ construction, respecting types} \end{array}$$

- *expressions* ranged over by B,C,M,N ... $\in \mathbf{Exp}_\Sigma$

$$\begin{array}{rcll} M & ::= & t & \% \text{ term} \\ & | & c(M_1, \ldots, M_n) & \% \ c/n \in DC_\Sigma, \text{ construction, respecting types} \\ & | & f & \% \ f \in FS_\Sigma, \text{ function symbol} \\ & | & (MN) & \% \text{ application, respecting types} \\ & | & (B \to M) & \% \text{ guarded expression, } B\text{: } \mathit{bool} \\ & | & (B \to M_1 \Box M_2) & \% \text{ conditional expression} \\ & & & \% \ B\text{: } \mathit{bool},\ M_1, M_2 : \tau \text{ for some } \tau \end{array}$$

Expressions should be well-typed. We omit the formal definition of a type inference system for expressions; cfr. [Milner 78, Damas, Milner 82]. In the sequel we will reserve B, C for boolean expressions.

We remark that $B \to M$ and $B \to M_1 \Box M_2$ are intended to mean "**if** B **then** M **else** undefined" and "**if** B **then** M_1 **else** M_2," respectively.

We shall assume that application associates to the left and omit brackets accordingly.

A BABEL-*program* of signature Σ consists of a set of defining rules for the non predefined symbols in FS_Σ. The rules for the predefined symbols are implicitly added to every program, and will be presented later.

Notice that any $f \in FS_\Sigma$ must have type $\tau_1 \to \ldots \to \tau_n \to \tau$, for some $n \geq 0$ and some τ that is not of the form $\tau' \to \tau''$. Here, n is the *type-arity* of f.

2 The functional logic language BABEL

The language BABEL has been designed by Mario Rodríguez-Artalejo and Juan José Moreno-Navarro to achieve integration of functional and logic programming in a flexible and mathematically well-founded way [Moreno, Rodríguez 88], [Moreno, Rodríguez 89], [Moreno 89]. It is based on a constructor discipline and uses narrowing as an evaluation mechanism; cfr. [Reddy 85], [Reddy 87]. In this paper we work with an extension of BABEL that supports polymorphic types and higher-order functions. However, higher-order *logic* variables are *not* allowed. Thus higher-order variables are never affected by narrowing.

Before we give the formal syntactic description of BABEL-programs, we consider a small example program:

```
typevar A, B.
data     list(A) = nil | cons (A, list(A)).

fun map: (A → B) → list(A) → list(B).
    map F [ ]       := [ ].
    map F [X|Xs]  := [ F X | map F Xs ].

eval (map (+ 2) [X,3]) = [6,Y].
```

that will yield the boolean value true with variable bindings {X/4, Y/5}.

Note that we assume numbers and arithmetic operations as well as the equality operator to be predefined. We also allow a PROLOG-like syntax for lists.

The higher-order function map can be used in a very flexible way due to the first-order logic variables. The higher-order variable F can only be used as in functional programs.

2.1 Syntax

Let $\Sigma = \langle TC_\Sigma, DC_\Sigma, FS_\Sigma \rangle$ be a polymorphic signature, i.e.

- TC_Σ is a set of *ranked type constructors* : tc/n, e.g. nat/0, list/1, ...
- DC_Σ is a set of *typed data constructors*: $c : \tau$, $c' : \tau_1 \times \ldots \times \tau_n \to \tau$ with $\tau, \tau_i \in \mathbf{CType}_\Sigma$, where $\mathbf{CType}_\Sigma$ is the set of *constructed types* over type variables $\alpha, \beta, \gamma \ldots \in \mathbf{TVar}_\Sigma$ defined recursively by

$$\tau ::= \alpha \mid \mathrm{tc}/0 \mid \mathrm{tc}/n(\tau_1, \ldots, \tau_n) \;\; \% \; n \geq 2.$$

- FS_Σ is the set of *typed function symbols* $f : \tau$ with $\tau \in \mathbf{Type}_\Sigma$, where $\mathbf{Type}_\Sigma$ is the set of *polymorphic types* defined by

$$\tau ::= \alpha \mid \mathrm{tc}/0 \mid \mathrm{tc}/n(\tau_1, \ldots, \tau_n) \mid (\tau \to \tau').$$

  In the sequel we assume that "→" associates to the right and omit brackets accordingly.

1 Introduction

During the last years, several attempts have been made to achieve an integration of functional and logic programming languages in order to combine the advantages of the different programming paradigms in a single framework [DeGroot, Lindstrom 86], [Bellia, Levi 86]. Usually one argues that logic languages have more expressive power than functional languages, while the latter have a simpler execution model, particularly suited to parallel implementations.

We investigate in this paper the implementation of the functional logic language BABEL [Moreno, Rodríguez 89] on a graph reduction architecture. We extend the original version of BABEL, which was first-order and type-free, to a higher-order functional logic language with a polymorphic type discipline. Its operational semantics is based on narrowing, an evaluation mechanism that uses unification for parameter passing [Reddy 85], [Reddy 87].

Our approach has been to extend a graph reduction machine[1] that has been designed for the execution of functional programming languages [Loogen et al. 89] by features that are necessary to execute functional logic programs, i.e. by unification and backtracking facilities. Accordingly, we may guarantee that purely functional programs can be executed almost as efficient as in the original machine. Of course, some overhead due to the different parameter passing mechanism cannot be avoided. But our way of implementing narrowing seems to be an interesting alternative to the reduction of narrowing to SLD-resolution, as it has been proposed by several researchers; cfr. e.g. [van Emden, Yukawa 87], [Bosco, Giovannetti, Moiso 88].

The BABEL abstract machine is controlled by machine code. We describe the compilation of BABEL programs into this abstract code. The compilation rules are very similar to the functional case because backtracking will not be controlled by code but is performed by an implicit mechanism that is started by a special machine instruction.

For simplicity we discuss in this paper the realization of an innermost narrowing strategy, i.e. the arguments of a function call are evaluated before the function call.

This paper is organized as follows. In section 2 we give a short review of the language BABEL. The structure of the BABEL machine will be described in section 3. Then we discuss the compilation of BABEL programs. In section 5 we explain the organization of backtracking, i.e. the most important novel feature of the graph reduction machine.

The BABEL machine has been implemented in OCCAM on a transputer. We have chosen the transputer implementation because we intend to incorporate the sequential BABEL machine in a parallel environment to exploit parallelism in the program execution whenever possible. In the appendix the complete formal specification of the BABEL machine is given.

[1] more precisely the sequential kernel of a parallel graph reduction machine

Graph-based Implementation of a Functional Logic Language*

Herbert Kuchen, Rita Loogen
RWTH Aachen†

Juan José Moreno-Navarro
Universidad Politécnica de Madrid‡

Mario Rodríguez-Artalejo
Universidad Complutense de Madrid§

Aachener Informatik–Berichte Nr. 89–20

Abstract

We investigate the development of a graph reduction machine for a higher-order functional logic language by extension of an appropriate architecture for purely functional languages. To execute logic programs the machine must be capable of performing unification and backtracking. We show the integration of these mechanisms in a programmed (functional) graph reduction machine. The new machine has been implemented on a transputer system.

*This work has been partially supported by a german-spanish cooperation action, funded by the german D.A.A.D. and the spanish M.E.C

†Lehrstuhl für Informatik II, Ahornstraße 55, 5100 Aachen, West Germany

‡Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software, Facultad de Informática, Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain

§Departamento de Informática y Automática, Facultad de C.C. Matemáticas, 28040 Madrid, Spain