pended and-nodes on the wake stack. (It is also appropriate to unlink suspensions referring to dead and-nodes.) Push a trail entry for *x*.

Note that this is one point where the abstract machine differs from the execution model. Waking does not occur at the point of adding constraints, since it is not known which x-registers are used. Instead, waking is delayed until processing the next non-constraint statement.

To *dereference a tree x*, if *x* is a bound variable, dereference its value, otherwise return *x*. To *dereference a register x*, dereference the contents of *x*, and store the result in *x*.

6.4.3 Failing

To *fail the current and-node*, change its state to dead and unlink it from alternatives. Remove all trail entries in the current context, resetting their variables to unbound. Reset all variables in the constraint list to unbound. Remove all and-tasks in the current context. Remove all references from the wake-up stack. If there remains a single solved and determinate sibling and-node, install it and proceed. Otherwise, back up.

6.4.4 Suspending

To *suspend the current and-node*, unless it has a continuation, store the program counter in the continuation pointer of the current and-continuation, create a copy of it on the heap, making the copy the continuation of the current and-node, and pop it from the and-stack. Reset all variables in the constraint list to unbound. For each trail entry in the current context: Remove it and reset its variable to unbound. Create a constraint referring to the variable and its value. and add it to the current and-node. Suspend the and-node on the variable. If the dereferenced value is a variable, suspend the and-node also on that variable.

To *suspend an and-node on a variable*, add a suspension for this and-node to the variable and, for each and-node in the path from the current and-node to, but not including, the home and-node of the variable, change its state to unstable.

6.4.5 Backing Up

To *back up*, examine the top object of the choice-stack, and do one of the following depending on its type.

- If it is a choice continuation, decode instructions starting from its continuation pointer.
- If it is a promote task, remove it, install (simply) the and-node referred to from the parent choice-node, and promote it.
- If it is a choice splitting task, remove it, set the current and-node to the node referred to by the task, and attempt choice splitting.
- If it is a no-choice task, remove the task, and remove the top context. If the parent of the current choice-node is null, terminate the abstract machine.

Otherwise, if there are no alternatives, set the current and-node to the parent and fail. Otherwise, set the insertion point to the right sibling of the current choice-node, set the current and-node to the parent, and proceed.

Note that, to avoid testing if the parent of the choice-node is null, a special no-choice task can be used in the root node.

6.4.6 Waking

To *wake at call with N arguments*, push a resume task with the program counter as continuation pointer and the N x-registers as a-registers. Push a restore insertion point task and insert it at the insertion point. Wake and proceed.

To *wake at proceed*, push a restore insertion point task and insert it at the insertion point. Wake and proceed.

To *wake at a guard*, store the program counter in the continuation pointer of the current and-continuation, wake, and proceed.

To *wake*, for each reference to an and-node on the wake stack, remove it and, if the and-node is live, push a corresponding wake task.

Note that pushing a resume or insertion point task only to discover that no live node could be waked can easily be avoided in an implementation.

6.4.7 Promoting

To *promote* the current and-node, set the forward attribute to refer to the parent of the current choice-node, here called the *target and-node*. Unlink the current choice-node, setting the insertion point to its right sibling. Promote the trail and promote constraints. Set the current and-node to the target and-node. Remove the current context and remove the no-choice task. Decode instructions starting with the next instruction (following the current guard instruction).

To *promote simply*, do as above, but do not promote the trail and constraints, which are known to be empty.

To *promote the trail*, for each trail entry in the current context, select the first applicable case below.

- If the home of its variable is the target and-node, remove the trail entry. If suspensions on the variable refer to and-nodes in the target and-node, push references to these on the wake stack and unlink the suspensions. Unlink also all suspensions referring to dead nodes.
- If the dereferenced value of the variable is a variable, the home of which is the target and-node, bind this latter variable to the former after changing the state of the former to unbound. Remove the trail entry.
- Otherwise, do nothing.

To *promote constraints*, for each constraint in the current and-node, select the first applicable case below.

- If the home of its variable is the target and-node, then, if suspensions on the variable refer to and-nodes in the target and-node, push references to these on the wake stack and unlink the suspensions. If there are no such suspensions, do nothing.
- If the dereferenced value of its variable is a variable, the home of which is the target and-node, bind this latter variable to the former after changing the state of the former to unbound.
- Otherwise, move the constraint to the target and-node.

6.4.8 Pruning

To *prune alternatives,* unlink all alternatives to the right and left of the current and-node and mark them as dead. Delete all choice-tasks in the current context and push a no-choice task.

To *prune alternatives to the right,* do as above, but unlink only to the right.

6.4.9 Proceeding

To *proceed,* examine the top object of the and-stack, and do one of the following depending on its type.

- If it is an and-continuation, decode instructions starting from the continuation pointer.
- If it is a wake task, pop it. If it refers to a live and-node, dereference and install it. Proceed.
- If it is a restore insertion point task, pop it, unlink it, set the insertion point to the right of the goals attributes referred to, and proceed.
- If it is a resume task, pop it, move the a-registers to corresponding x-registers, and decode instructions starting from the continuation pointer.

6.4.10 Installing

To *install an and-node when proceeding,* for each of the and-nodes in the path from, but not including, the current and-node to the and-node to be installed, enter its parent choice-node and then enter itself. This process may be interrupted by failing when entering and-nodes.

To *install an and-node when failing, backing up, or collecting,* do as above, but start by entering the first and-node.

To *enter a choice-node,* push a context referring to the tops of the pertinent stacks, then a no-choice task.

To *enter an and-node,* make it the current and-node and make its goals attribute the insertion point. For each constraint, select the first applicable case below.

- If its dereferenced variable is equal to its dereferenced value, unlink the constraint.

- If its variable is unbound, set the value of this variable to the value in the constraint and change state to bound.
- If its variable is bound and its value is an unbound variable, swap the variable and the value of the constraint, and do as above.
- Otherwise, unlink the constraint and unify the value of the variable with the value of the constraint. This may fail, interrupting entering.

Then, push an and-continuation corresponding to the continuation of the and-node. Finally, wake.

Note that when *installing simply*, only case two is applicable when entering an and-node, since failure or entailment cannot occur due to stability.

6.4.11 Choice Splitting

To *attempt choice splitting*:

- Select a candidate. If none is found, back up.
- If the candidate's grandparent is the current and-node, then, if it has only one sibling, which is solved, push a promote task referring to the sibling, otherwise, push a choice splitting task referring to the current and-node.
- Copy w.r.t. the candidate.
- Install (simply) the copy of the candidate and proceed.

To *select a candidate*, look for solved and-nodes in the current and-node, the continuations of which refer to guard_nondet instructions. Select one according to the chosen scheme, for example, the left-most candidate. (See Section 6.7.8 for a some remarks on other possibilities.)

To *copy w.r.t. a candidate*, consider the and-node which is the grandparent of the candidate. Make an isomorphic copy of this node and its attributes, recursively, and insert it to the left of the grandparent, with the following three exceptions: (1) Do not copy the siblings of the candidate—make the copy of the candidate the only alternative. (2) Do not copy variables that are external to the copied node. (3) Do not copy suspensions that refer to dead and-nodes or to the candidate. For each external variable encountered, refer to it also from the copy, and for each of its suspensions that refers to a copied node, add a new suspension that refers to the copy, deleting the original suspension if it refers to the candidate. Finally, unlink the candidate and mark it as dead.

Note that it is permitted to dereference all variable chains during copying, since all local bindings are deinstalled, and the copied subtree is stable. (The single exception is for variables which are used as accumulators in aggregates.)

6.4.12 Collecting

To *collect* the current and-node, change its state to dead, unlink it from alternatives, and set the forward attribute to refer to the parent of the current choice-node. Store a reference to the next instruction in the current and-continuation,

and set the and-attribute in the current context to the top of the stack. If there is a left alternative, install it, otherwise install the right alternative. Proceed.

Note that by putting the continuation in the context of the parent and-node, it will be executed after having promoted the unit or after backing up. Note also that if there is a left alternative, it may be a collect alternative, whereas the right alternative might be the final unit alternative. (See Section 6.6.6 for how these relate to the aggregate statements.) Note, finally, that precisely this moving of an and-task "across" choice-tasks is what makes it necessary to have two separate task stacks instead of just one. (See Section 6.9.2 for a discussion of alternatives.)

6.5 AN INSTRUCTION SET

In the description of this instruction set, familiarity with the architecture of and compilation techniques for the Warren Abstract Machine (WAM) is assumed (such as can be acquired from Aït-Kaci [1991] and Carlsson [1990]).

6.5.1 Overview of Instructions

The instructions are divided into four groups: the *choice*, *and-*, *guard*, and *constraint* instructions. The constraint instructions are further divided into *get*, *put* and *unify* instructions.

Choice instructions are used in the code for choice statements, and-instructions for program atoms, composition, and clauses, and guard instructions for the guard operators.

Choice	*And*	*Guard*
switch_on_tree L_v L_s L_c try L N retry L trust L try_only L	call L N execute L N proceed allocate N deallocate fail	guard_condition guard_commit guard_nondeterminate guard_collect y_i x_j / y_j x_k / y_k guard_unit y_i x_j / y_j guard_top

Figure 6.4. Choice, and-, and guard instructions

Get, put, and unify instructions code tree equations in terms of registers that hold references to variables. Get instructions are used when these registers are already defined, put instructions define the value, and unify instructions are used for sub-trees in both cases.

Get	*Put*	*Unify*

get_variable x_i / y_i x_j get_value x_i / y_i x_j get_symbol S x_i get_constructor F x_i	put_variable x_i / y_i x_j put_value x_i / y_i x_j put_symbol S x_i put_constructor F x_i put_void x_i	unify_variable x_i / y_i unify_value x_i / y_i unify_symbol S unify_void

Figure 6.5. Get, put, and unify instructions

6.5.2 Choice Instructions

switch_on_tree L_v L_s L_c

Dereference x_0. Go (set the program counter) to label L_v, L_s, or L_c depending on its contents (variable, symbol, or constructor, respectively).

try L N

Create a choice-node, insert it at the current insertion point, and enter it. Create an and-node as one of its alternatives. Make it the current and-node and set the insertion pointer to refer to its goals. Push a choice-continuation with the continuation pointer referring to the next instruction (which is retry or trust), and with N a-registers getting their values from corresponding x-registers. Go to L.

retry L

Restore x-registers from corresponding a-registers in the current choice-continuation. Create an and-node and insert it as the last alternative in the current choice-node. Make this the current and-node and set the insertion pointer to refer to its goals. Go to L.

trust L

Do as in retry, but pop the choice-continuation and push a no-choice task before going to L.

try_only L N

Do as in try, but push a no-choice task instead of a choice-continuation.

6.5.3 And-Instructions

call L N

Set the continuation pointer of the current and-continuation to refer to the next instruction. Set the program counter to L. If the wake stack is non-empty, wake at call with N arguments.

execute L N

Set the program counter to L. If the wake stack is non-empty, wake at call with N arguments.

proceed

If the wake stack is non-empty, wake at proceed. Otherwise, proceed.

allocate N

Push an and-continuation with N y-registers.

deallocate

Pop the current and-continuation.

fail

Fail.

6.5.4 Guard Instructions

guard_condition

If the wake stack is non-empty, wake at a guard. If the current and-node is quiet, prune alternatives to the right. If it is left-most, promote simply. Otherwise, suspend it and attempt choice splitting.

guard_commit

If the wake stack is non-empty, wake at a guard. If the current and-node is quiet, prune alternatives and promote simply. Otherwise, suspend it and attempt choice splitting.

guard_nondeterminate

If the wake stack is non-empty, wake at a guard. If the current and-node is solved and determinate, promote. Otherwise, suspend it and attempt choice splitting.

guard_collect y_i x_j/y_j x_k/y_k

If the wake stack is non-empty, wake at a guard. If the current and-node is not quiet, or, if the aggregate is ordered, the current and-node is not left-most, suspend it and attempt choice splitting. Otherwise, move the value of the variable in y_i to x_k (y_k). Create a new variable with the grandparent and-node as home, store it in x_j (y_j), and make it the new value of the variable in y_i. Collect the current and-node.

guard_unit y_i x_j/y_j

If the wake stack is non-empty, wake at a guard. If the current and-node is not quiet, suspend it and attempt choice splitting. Otherwise, move the value of the variable in y_i to y_j (x_j). Promote the current and-node (simply).

guard_top

If the wake stack is non-empty, wake at a guard. Suspend the current and-node and attempt choice splitting.

6.5.5 Tree-Constraint Instructions

get_variable x_i x_j
get_variable y_i x_j

Move the contents of x_j to x_i (y_i).

get_value x_i x_j
get_value y_i x_j

Unify the contents of x_j and x_i (y_i).

get_symbol S x_i

Dereference x_i, and do one of the following depending on the result.

- variable: Bind x_i to S (a reference to a symbol).
- symbol equal to S: Continue.
- other: Fail.

get_constructor F x_i

Dereference x_i, and perform one of the following depending on the result.

- variable: Create a constructor with functor F. Bind x_i to a reference to it. Set the current argument to refer to its first argument. Enter write mode.
- constructor with functor F: Set the current argument to refer to its first argument. Enter read mode.
- other: Fail.

put_variable x_i x_j
put_variable y_i x_j

Create a variable and store a reference to it in x_i (y_i) and x_j.

put_value x_i x_j
put_value y_i x_j

Move the contents of x_i (y_i) to x_j.

put_symbol S x_i

Store S in x_i.

put_constructor F x_i

Create a constructor with functor F and store a reference to it in x_i. Set the current argument to refer to its first argument. Enter write mode.

put_void x_i

Create a variable and store a reference to it in x_i.

unify_variable x_i
unify_variable y_i

In read mode, move the contents of the current argument to x_i (y_i). In write mode, create a variable and store a reference to it in the current argument and in x_i (y_i). In both modes, step to the next argument.

unify_value x_i
unify_value y_i

In read mode, unify the contents of the current argument and the contents of x_i (y_i). In write mode, move the contents of x_i (y_i) to the current argument. In both modes, step to the next argument.

unify_symbol S

In read mode, dereference the contents of the current argument, and do one of the following depending on the result.

- variable: Bind it to S.
- symbol equal to S: Continue.
- other: Fail.

In write mode, store a reference to S in the current argument. In both modes, step to the next argument

unify_void

In write mode, create a variable and store a reference to it in the current argument. In both modes, step to the next argument.

6.6 CODE GENERATION

The code generation principles that follow are not optimal and should not be construed as a compiler. They serve as an introduction, or to confirm what should already be clear to a reader acquainted with compilation of Prolog to the WAM (see, e.g., [Carlsson 1990], which presents this in great detail).

Seen from the clausal point of view, code generation is similar to WAM code generation. The differences are the guard and try_only instructions. The guard instructions are like calls in that variables that should survive them are stored in y-registers. Since choice-nodes and and-nodes have to be created for guard execution, the try_only instruction has to be used when selecting a single clause.

6.6.1 Normal Programs

To simplify the description of code generation, it is assumed that definitions are *normal* in the following sense. Hiding occurs only as hiding over clauses in choice statements, statements in aggregate statements, and over bodies of definitions. The body of a definition is either a choice statement, aggregate statement, or does not contain choice or aggregate statements. The compositions are

then compositions of atoms, and are regarded as sequences composed by an associative operator. It is also assumed that all tree constraints are of the form $v = t$, where v is a variable.

Hiding is normalised by the following transformations (on the abstract syntax), renaming variables as necessary.

$$A, (V : B) \Rightarrow V : A, B$$

$$(V : A), B \Rightarrow V : A, B$$

$$U : (V : A) \Rightarrow U \cup V : A$$

$$U : (V : A) \,\%\, B \Rightarrow U \cup V : A \,\%\, B$$

$$U : A \,\%\, (V : B) \Rightarrow U \cup V : A \,\%\, B$$

A body of a definition properly containing a choice or aggregate statement B is normalised by replacing B with a program atom A, which is given a new name, and the parameters of which are the free variables of B. A definition A := B is then added to the program.

6.6.2 Composition and Chunks

A (normal) composition statement consists of a sequence of *chunks* of the form

$$A_1, \ldots, A_k, B$$

where A_i are constraint atoms and B is a program atom, possibly followed by a final chunk

$$A_1, \ldots, A_k$$

For composition statements enclosed in hiding, we speak of variables that are *parameters* and variables that are *local* (hidden), and otherwise ignore hiding. We may add new local (hidden) variables in transformations. A variable is *temporary* if it is local and occurs only in one of the chunks, or if it is a parameter and occurs only in the first chunk. We speak of *first* and *single* occurrences of variables. For the parameters, (unique) *parameter numbers* are known.

As a preparation for code generation we rewrite constraint atoms and program atoms in a sequence of chunks as follows, repeatedly if necessary.

If v is local and has its first occurrence in a constraint atom

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

it is rewritten to

$$u_{i1} = t_{i1}, \ldots, u_{im} = t_{im}, v = f(x_1, \ldots, x_n)$$

where t_{ij} are constructor expressions, u_{ij} are new local variables, and $x_k = u_k$, if $k = i_j$, and $x_k = t_k$, if $k \neq i_j$. Otherwise, it is rewritten to

$$v = f(x_1, \ldots, x_n), u_{i1} = t_{i1}, \ldots, u_{im} = t_{im}$$

with conditions as above.