difference cannot be observed. Such stipulations would be fundamental to the nature of the language.

A fair execution model for AKL is also considerably more complex. The investigation of AKL, and related languages, has not yet reached the point where it would be possible to argue conclusively for or against general fairness.

Meanwhile, it is obvious that, to be useful for, e.g., background processes, even a fair execution model has to be complemented with some form of metalevel control that allocates computational resources to different subcomputations. Such directives are available in the Oz language and system [Smolka et al 1994].

CHAPTER 6

AN ABSTRACT MACHINE

The abstract machine presented here is a refinement of the execution model, with some discrepancies related to waking. It specifies in more detail the representation of programs and execution states, and how tasks are executed on this representation. These representations are chosen to be suitable for implementation on a concrete machine, but are not discussed at the implementation level.

The abstract machine is particular to the constraint system of rational trees. The requirements of arbitrary constraint solvers are not sufficiently understood at this point to allow a fully general treatment. Carlson, Carlsson, and Diaz [1994] and Keisu [1994] have investigated particular constraint solvers—for finite domain constraints and rational tree constraints closed under all logical combinators.

A concrete implementation exists, the AGENTS system, that is based on a closely related abstract machine [Janson and Montelius 1992; Janson et al 1994]. The abstract machine of this presentation is for the most parts considerably simplified, but supports the *eager waking* behaviour of the execution model as well as the *lazy waking* of AGENTS 0.9. Some aspects of the concrete implementation are discussed.

No attempt has been made to describe an optimal machine. On the contrary, much has been sacrificed to keep the presentation simple. The purpose of this chapter is to introduce fundamental machinery for this particular style of abstract machine. Some important optimisations are described.

6.1 OVERVIEW

An execution state corresponds to a *machine state* as follows.

The (labelled) configuration is represented by *and-nodes* (corresponding to guarded goals with and-box guards or to and-boxes in or-boxes), *choice-nodes* (corresponding to choice-boxes, aggregate-boxes with or-boxes, or to the top-level or-box), *and-continuations* (corresponding to statements), *choice-continua-*

tions (corresponding to parts of choice-statements), *variables*, *symbols*, *tree constructors*, *constraints*, and *instructions* (representing statements)

The set of suspensions is represented by *suspensions* on variables and the *trail*.

The combination of and-list, choice-list, and context list is represented by the *and-stack*, the *choice-stack*, the *context stack*, and the *current and-node* register.

Agent definitions and their component statements are represented by *instructions*. The choice of instruction set is largely orthogonal to the rest of the abstract machine. An instruction set which is loosely based on the WAM is presented [Warren 1983], and some optimisations are suggested.

6.2 DATA OBJECTS

Objects are characterised by their types and attributes. All objects and some attributes of objects have unique *references* in a given machine state. For the representations of goals, these correspond to labels. It is assumed that there is a reference, called *null*, that is different from the reference of any object, and which may be used in the place of any reference. Objects reside in different data areas (which are described in Section 6.3). Occasionally, a reference will be referred to as the object and vice versa, and a reference to an attribute as a reference to the object, for the purpose of less verbose descriptions, and no confusion is expected from this relaxation.

6.2.1 And-Nodes

An *and-node* has the attributes

- *parent*: a reference to a choice-node
- *alternatives*: a pair of references to alternatives attributes
- *constraints*: a reference to a constraint (list)
- *goals*: a pair of references to goals attributes
- *continuation*: a reference to an and-continuation
- *forward*: a reference to an and-node
- *state*: one of live-stable, live-unstable, or dead

The and-node is on a double-linked list of alternatives, one link of which is in its parent choice-node. It has to support arbitrary insertion and deletion, and is double-linked for simplicity. Of the pair, one is called *left* and the other *right*.

The constraint list is null until the and-node is suspended for the first time (if ever). It will then contain bindings on external variables.

The and-node contains a double-linked list of goals, one link of which is in the and-node itself. Like the list of alternatives, it has to support arbitrary insertion and deletion, and is double-linked for simplicity. It is initially empty, i.e., its links refer to itself.

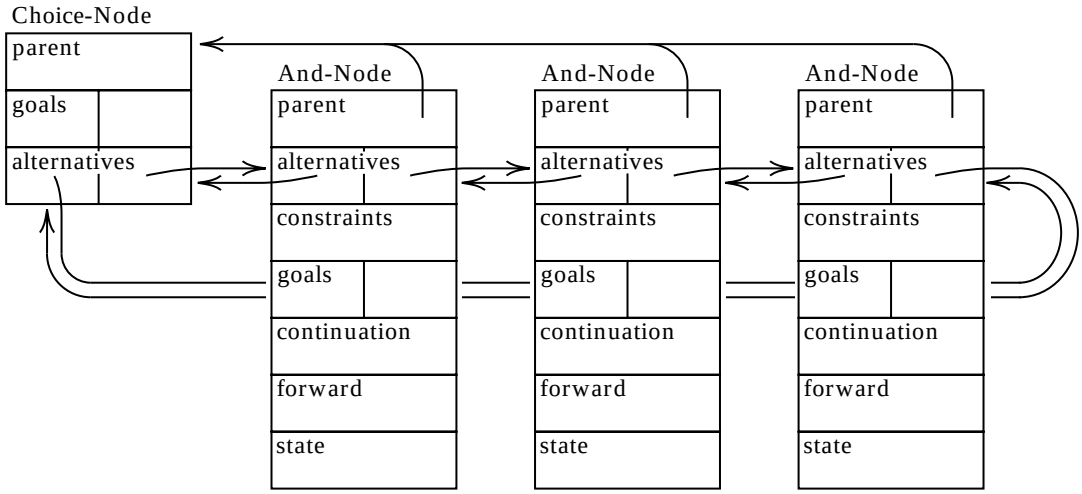

Figure 6.1. Tree of and- and choice-nodes

The continuation represents the guard operator and the body of a guarded goal, or serves as a special sentinel in and-nodes in the top-level and in aggregates. As the constraint list, it is null until the and-node is suspended.

Until promotion, the forward pointer is null. Then, it is set to refer to the and-node to which promotion took place.

The state of an and-node is either *live* or *dead*. The state of a live and-node is further subdivided into *live-stable* or *live-unstable*. In a node which is live-stable it is known that no constraint on external variables exists within the node. In a node which is live-unstable there might be such a constraint. An and-node is initially marked as *stable*. If a constraint is placed on a variable external to the and-node it is marked as *unstable*. An and-node that has been marked as unstable will never be marked as stable again. An and-node is marked as dead if it fails, is pruned, or is promoted.

6.2.2 Choice-Nodes

A *choice-node* has the attributes

- *parent*: a reference to an and-node
- *goals*: a pair of references to goals attributes
- *alternatives*: a pair of references to alternatives attributes

In the root choice-node, the parent and-node is null.

The choice-node is in a double linked-list of goals, one link of which is in the parent and-node.

The choice-node contains a double linked-list of alternatives, one link of which is in the choice-node itself.

Figure 6.1 illustrates how and- and choice-nodes are connect via parent and double links. In the figure, a choice-node is the parent and it and the and-nodes

are linked via an alternatives chain. The situation in which an and-node is the parent and the choice-nodes are linked via a goals chain is symmetric.

A node is *in* another node, if the latter can be reached from the former by following parent links. By the *grandparent* of a node is meant its parent's parent. By *empty* alternatives and goals, we mean that they contain no and-nodes or choice-nodes, respectively. An object in a double-linked list is known to be *left-most* (*right-most*) if its left (right) reference refers to an attribute in its parent. Familiar notions such as *inserting* and *unlinking* objects will be used on single- and double-linked lists.

6.2.3 Constraints and Variables

The abstract machine is specific to the constraint system of rational trees. There are three types of components in tree expressions: *symbols*, *tree constructors*, and *variables*. A *tree* is a reference to a tree component. In terms of the computation model, a tree is a variable which is constrained to be equal to the expression to which it refers.

A *symbol* has no attributes.

A *tree constructor* has the attributes

- *functor*: the name and arity
- *arguments*: a vector of trees

A *variable* has the attributes

- *value*: a tree
- *state*: one of bound or unbound
- *home*: a reference to an and-node
- *suspensions*: a reference to a suspension

In a new variable, suspensions is null and the state is unbound.

A *suspension* has the attributes

- *next*: a reference to a suspension (list)
- *suspended*: a reference to an and-node

The constraints in an and-node correspond to bindings to external variables.

A *constraint* has the attributes

- *next*: a reference to a constraint
- *variable*: a reference to a variable
- *value*: a tree

6.2.4 And-Continuations

An *and-continuation* has the attributes

- *continuation pointer*: a reference to a sequence of instructions

- *y-registers*: a vector of trees

The instructions represent a statement of which the trees in the vector are the (constrained) free variables.

And-continuations serve a dual purpose, as the representation of a statement and as the representations of corresponding **s** (statement) tasks.

6.2.5 Choice-Continuations

A choice-continuation has the attributes

- *continuation pointer*: a reference to a sequence of instructions
- *a-registers*: a vector of trees

The instructions represent unexplored guarded goals in a choice-box of which the trees in the vector are the (constrained) free variables.

Like and-continuations, choice-continuations serve a dual purpose, as the representation of guarded goals and as the representation of a corresponding sequence of **c** (clause) tasks.

6.2.6 Trail Entries

A *trail entry* has the attribute

- *variable*: a reference to a variable

A trail-entry refers to an external variable which has been bound locally.

6.2.7 Tasks

The tasks of the abstract machine correspond closely to the tasks of the execution model.

The **s** (statement) and **a** (and-box) tasks are represented by and-continuations, which also serve as representations of the pertaining statements. The **w** (wake) task is explicit also in the abstract machine.

The **c** (clause) tasks are represented by choice-continuations, which also serve as representations of the pertaining guarded goals. The **o** (or-box, here called no-choice), **cs** (choice splitting), and **p** (promote) tasks are explicit also in the abstract machine.

When regarded as tasks, and- and choice-continuations are referred to as tasks.

There is a need for new tasks related to waking—*resume* and *restore insertion point*. At the point of a call, the program counter and the x-registers represent a program atom that has to be made explicit when switching context. The insertion point corresponds to the label in the **s** tasks.

The *no-choice* task has no attributes.

The *wake*, *choice splitting*, and *promote* tasks have the attribute

- *node*: a reference to an and-node

The *resume* task has the attributes

- *continuation pointer*: a reference to a sequence of instructions
- *a-registers*: a vector of trees

The *restore insertion point* task has the attribute

- *goals*: a pair of references to goals attributes

6.2.8 Contexts

A *context* has the attributes

- *and*: a reference to an and-task or and-continuation
- *choice*: a reference to a choice-task or choice-continuation
- *trail*: a reference to a trail entry

Contexts serve the purpose of the context list in the execution model, namely to organise tasks in and- and choice-node levels. The trail attribute is a part of the representation of the set of suspensions.

6.3 DATA AREAS AND REGISTERS

6.3.1 Data Areas

The abstract machine has seven data areas: heap, trail stack, and-stack, choice stack, context stack, wake-up stack, and static store (Figure 6.2, next page).

The *heap* is an unordered collection of and-nodes, choice-nodes, tree constructors, variables, suspension lists, constraint lists, and and-continuations. New objects can be *created* on the heap and are then assigned unique references. Objects on the heap that are not referenced are removed by *garbage collection* (which is not discussed further).

A *stack* is a sequence of objects, regarded as extending upwards. New objects can be *pushed* on the stack and are then assigned unique references. For a given stack, the reference that would be assigned to a pushed object is known, and is called *top*. Any object in a stack can be *removed*, giving new references to all objects above it. When the top-most object is removed, it is *pop'ed*.

The *trail-stack* is a stack of trail entries. The *and-stack* is a stack of and-continuations and and-tasks. The *choice-stack* is a stack of choice-continuations and choice-tasks. The *context-stack* is a stack of contexts. The *wake-up* stack is a stack of references to and-nodes.

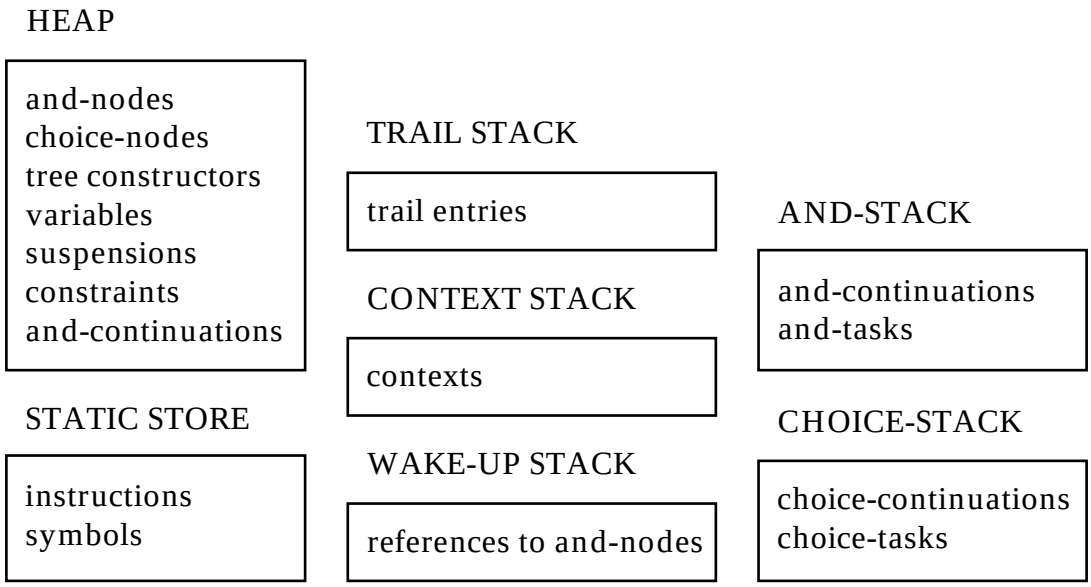

Figure 6.2. Data areas of the abstract machine

The *static store* contains symbols and sequences of instructions.

Since the data area can often be inferred from the context, it will not always be explicitly named.

6.3.2 Registers

Registers are implicit in the management of the data areas. We will speak of the *current* objects of stacks, meaning the objects immediately below their tops. Other registers are

- the *program counter*: a reference to a sequence of instructions
- the *current and-node*: a reference to an and-node
- the *insertion point*: a reference to a goals attribute
- the *x-registers*: a vector of trees
- the *current argument*: a reference to a tree
- the *unification mode*: one of read or write

By the notation x_i is meant the *i*th x-register, counting from 0 (in accordance with the tradition). It is assumed that all x-registers exist that are accessed by instructions currently in the static store.

By the notation y_i is meant the *i*th element of the y-registers attribute of the current and-continuation, counting from 0. By the *current choice-node* we mean the parent of the current and-node.

6.3.3 Initial States

In an initial state, the static store contains instructions and symbols for a program, as given by Section 6.6. All other data areas are empty. The current and-

node is null. The instruction pointer refers to the code for the initial statement, as given by Section 6.6.7. The machine is thus ready to enter the instruction decoding state.

6.3.4 Useful Concepts

An object in the and-stack, choice-stack, or trail stack is *in the current context* if it is equal to or above the object referred to by the corresponding attribute in the current context, or if the context stack is empty.

An and-node is *solved* when its goals attribute is empty and the program counter refers to a guard instruction.

An and-node is *quiet* when it is solved, the constraints list is empty, and there are no trail entries in the current context.

A variable is *local* if its home attribute, dereferenced, refers to the current and-node. The variable is *external* if the current and-node is in the and-node referred to. Otherwise the variable is unrelated (an insignificant case).

To *dereference an and-node,* if its forward attribute is non-null, dereference it, otherwise return the and-node.

An and-node is *dead* if it, dereferenced, has state dead or is in an and-node, the state of which is dead. Otherwise, it is *live.*

An and-node is *determinate* if it is the only alternative and the topmost element of the choice-stack is a no-choice task.

6.4 EXECUTION

During execution, the abstract machine moves between main states as illustrated by Figure 6.3 (next page). These are completely defined by

- contents of data areas
- values for pertaining registers

The abstract machine can be halted and restarted again at these points with no extra information. Other, here transient, states exist that could be made well-defined in an implementation with the aid of extra registers.

The registers needed to enter a state are

- for Decode Instructions, (potentially) all except unification mode
- for Fail, the current and-node
- for Back Up, the current and-node
- for Wake, (potentially) all except unification mode
- for Proceed, the current and-node, insertion pointer
- for Split?, the current and-node

The following sections describe these states and their pertaining actions.

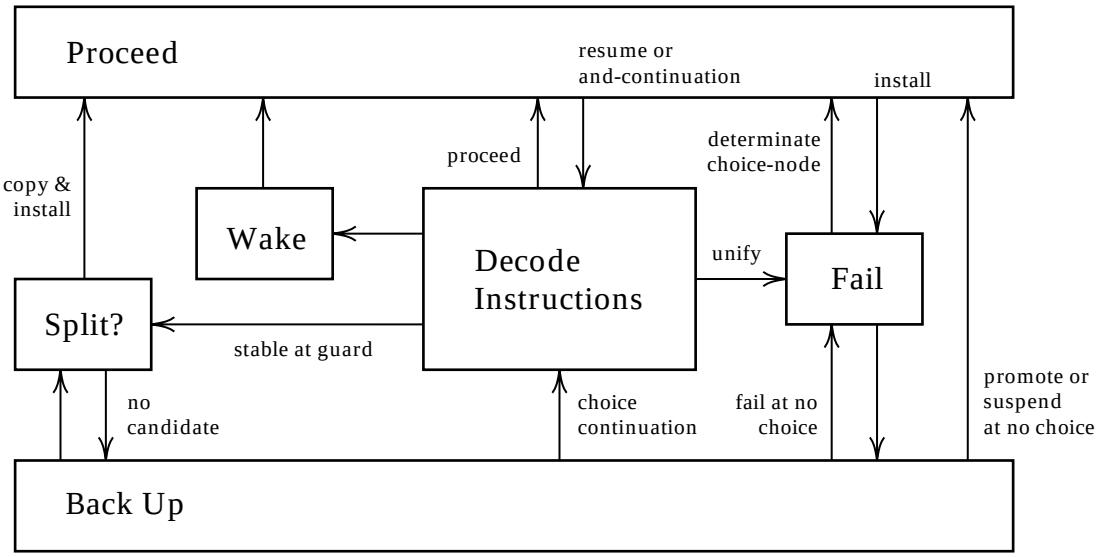

Figure 6.3. States of the abstract machine

6.4.1 Instruction Decoding

To *decode instructions*, interpret the instruction referenced by the program counter (according to Section 6.5). Unless the instruction moves to another state, continue decoding instructions, and, unless the instruction changes the program counter, move to the next instruction.

6.4.2 Unifying and Binding

To *unify x and y* perform the following. A set of pairs of references is used for unification of cyclic structures. It is assumed to be empty when unify is entered from instruction decoding. Dereference *x* and *y*. If *y* is a local variable, swap *x* and *y*, then select the first applicable case below.

- If *x* and *y* are identical references, do nothing.
- If *x* is a variable, bind *x* to *y*.
- If *y* is a variable, bind *y* to *x*.
- If *x* and *y* are tree constructors with the same functor, then, if *x.y* is found in the set of pairs, then do nothing, otherwise, add *x.y* to the set and unify each pair of corresponding arguments.
- Otherwise, fail.

To *bind x to y*, set the variable to state bound and store *y* in its value attribute, then select the applicable case below.

- If *x* is a local variable, unlink all suspensions on *x* and push references to the and-nodes on the wake stack.
- If *x* is an external variable, unlink all suspensions to and-nodes in the current and-node from *x*, and push corresponding references to the sus-