```
send(Request, TupleSpace) :=
        hash(Request, Index),
        arg(Index, TupleSpace, Port),
        send(Request, Port).
```

To each port in the hash table is connected a tuple space server of the kind presented above.

7.7.7 Erlang

Erlang is a concurrent functional programming language designed for prototyping and implementing reliable (distributed) real time systems [Armstrong, Williams, and Virding 1991; 1993]. Although not restricted to this area, Erlang is mainly intended for telephony applications. That functional programming as in the functional component of Erlang is provided by AKL is illustrated in Section 3.4.1. Here it is argued that the process component, including its error handling capabilities, can be expressed in AKL.

Erlang is probably less well known than the other languages in this chapter. Therefore, its essential features will be summarised before the comparison.

Processes are expressed in a sequential first-order functional programming language. The functional component is augmented with side effect operations such as creation of processes and message passing. The operation which creates a process returns a process identifier (*pid*).

Pid = spawn(*Module*, *Function*, *Arguments*)

A process may acquire its own pid using the self() operation. Pids may be given *global names*, a feature not further discussed here. Messages are sent to a pid with the *Pid* ! *Message* operation. Messages are stored in the *mail-box* of a process, from which they may be retrieved at a later time using the *receive* operation

```
receive
        Pattern [when Guard] -> … ;
        …
         [after Time -> …]
end
```

If a message matching a pattern (and satisfying the guard condition) is found among the messages in the mail-box, it is removed. The pattern is a constructor expression with local variables which are bound to the corresponding components in the message. The last clause may specify a *time-out*, which will not be further discussed here.

For the purpose of error-handling, processes may be *linked*. [It is assumed that these links are unidirectional, i.e., *from* one process *to* another process.] Links may be added, deleted, and, in particular, associated with new processes atomically at the moment of spawning (in which case a link is established from the new process to the old process).

link(*Pid*)	(*Pid* to self)
link(Pid_1, Pid_2)	(Pid_2 to Pid_1)
unlink(*Pid*)	
unlink(Pid_1, Pid_2)	
Pid = spawn_link(*Module*, *Function*, *Arguments*)	(*Pid* to self)

If an error occurs in a process, exit-signals are automatically sent to the processes it is linked to. A process which receives an exit-signal will by default also exit. This can be overridden, whereby signals are trapped, and converted to ordinary messages, which can be received in the usual fashion. A process may simulate an exit, and may also force another process to exit.

process_flag(trap_exit, [true / false])	(trap exit-signals)
receive {'EXIT', *From*, *Message*} -> … ; end	
exit(*Reason*)	
exit(*Pid*, *Reason*)	(force exit of *Pid*)

Signals may also be caught and generated by *catch* and *throw* operations, which are not further discussed here.

We now proceed to express the above in AKL using ports. To simplify the presentation, the following restrictions are made:

- Communication in Erlang is somewhat similar to *tuple space* communication in Linda, which was discussed in the previous section, and does also provide for time-outs. This aspect is ignored here. Communication is sent on a plain port, and served in a first-come first-served manner.
- Erlang distinguishes between *signals* and *messages*. This distinction is not made here, where both are regarded as messages, as when trapping of exit signals is turned on in Erlang.
- In Erlang, processes notify linked processes also about successful completion. This can be detected using short-circuiting, as in Section 7.7.1, and is not deemed important here.

An Erlang process identifier is mapped to an AKL port. Erlang processes have coarser granularity than AKL processes; to each Erlang process corresponds a *group* of AKL processes, performing the computations corresponding to the (sequential) functional component of Erlang. It is assumed that the definitions of these processes are written in the flat committed choice subset of AKL, where the guards may only contain simple tests.

The processes in a group share a pair p(V, P), the first component of which is either unconstrained or equal to the constant “exit”, and the second component of which is a port. It is assumed that all atoms, except the simple tests in guards, are supplied with this pair in an extra argument. If an error occurs in a primitive operation, a message of the form exit(*Reason*) is sent on the port in the pair, otherwise no action is taken. If the first component is equal to “exit”, an operation may terminate with no further action.

All choice statements in the definitions of processes in a group should have an extra case which examines the state of the first component of the pair, terminating if it is equal to “exit”.

```
p(…, F) :- …
…
p(…, p(exit,_)) :- | true.
```

A process group is created as follows.

```
group(Args, L, P) :=
        descriptor(P, Args, D),
        error_handler(L, P, S, F),
        processes(S, Args, F, P).
```

The *descriptor* agent forms a suitable descriptor of the current process to be sent to linked processes in the event of a failure, and may be defined in any appropriate manner. The definitions of the following agents are generic, and can be shared between all process groups.

```
error_handler(L, P, S, F) :=
        open_port(P, S),
        open_port(FP, Cs),
        F = p(_,FP),
        distribute(S, FP, Ms),
        linker(Cs, D, L).
```

The *distribute* agent filters out the *special* messages to the group, concerning linking and exiting, which are sent to the linker.

```
distribute([], C, Ms) :-
    → Ms = [].
distribute([special(C) | S], FP, Ms) :-
    → send(C, FP, FP1),
        distribute(S, FP1, Ms).
distribute([M | S], FP, Ms) :-
    → Ms = [M | Ms1],
        distribute(S, FP, Ms1).
```

The *linker* agent holds the list of pids (ports) of processes which the group is linked to, and an initial list (L) may be provided at the time of creation of the group, as in the *spawn_link* operation. It informs linked processes in the event of forced exit or failure.

```
linker([link(P1) | Cs], D, L) :-
    → linker(Cs, D, [P1 | L]).
linker([unlink(P1) | Cs], D, L) :-
    → delete(P1, L, L1),
        linker(Cs, D, L1).
linker([exit(R) | Cs], D, L) :-
    → F = p(exit,_),
        send_ports(L, exit(D, R)).
```

```
send_ports([], _) :-
    → true.
send_ports([P | Ps], M) :-
    → send(M, P),
      send_ports(Ps, M).
```

The *processes* agent creates the group of processes that perform the actual computation, receive and send messages, etc.

As demonstrated above, the degree of safety achieved with Erlang error handling can also be achieved in AKL, but at the cost of slightly more verbose programs: some process group set-up, and an extra argument to all atoms. Syntactic sugar could easily remedy this, if such capabilities warrant special treatment.

The computational overhead is the time to set up the error handler and the storage it occupies, the filtering of messages, the passing around of the extra argument, and (occasionally) suspending on the extra argument.

As it is currently defined, the error_handler will remain also when computation in the corresponding process group has terminated. It will terminate only when there are no more references to the port of the group.

7.8 DISCUSSION

The notion of ports provides AKL with a model of mutable data, which is necessary for effective object-oriented programming and effective parallel programming. Ports can serve as an efficient interface to foreign objects, e.g., objects imported from C++. They also support a variety of process communication schemes. Thus, ports reinforce otherwise weak aspects of AKL, and can also be introduced and used for the same purpose in other concurrent (constraint) logic programming languages.

Other useful communication mechanisms are conceivable that do not fall into the category discussed here. For example, the *constraint communication* scheme of Oz [Smolka, Henz, and Würtz 1993], has similar expressive power (except for automatic closing), but has to be supported by the additional concept of a blackboard. It can, however, be emulated by ports, using code equivalent to that for M-fields in Id (Section 7.7.4).

CHAPTER 8

RELATED WORK

Which are the ancestors and siblings of AKL and how does it compare to them? As a complement to the comments and small sections on related work scattered over the other chapters, this chapter offers more extensive comparisons between AKL and its main ancestors, Prolog, CLP, GHC, and the cc framework, and with a couple of selected languages that address similar problems.

8.1 AKL VS. PROLOG

Prolog is the grand old man of logic programming languages, conceived in 1973 and still going strong (see, e.g., [Clocksin and Mellish 1987; Sterling and Shapiro 1986; O'Keefe 1990]). AKL was designed so as to encompass as much as possible of the good sides of Prolog, while leaving out the not so good.

This section attempts to illuminate the relation between AKL and Prolog/CLP from several different perspectives. First, a simple computation model for constraint logic programming is shown, and a comparison is made with the AKL model. Then, we look at the issue of definite clause programming in AKL, how to translate a definite clause program to AKL, and how to interpret it using AKL. Prolog extends definite clause logic programming in a number of ways. Some of these can be dealt with in AKL, some cannot. It is shown how to interpret a logic program with side effects in AKL. Then, it is suggested how AKL provides the different forms of parallel execution suggested for Prolog as concurrency, potentially parallel execution, in its computation model. Finally, the Basic Andorra Model for the parallel execution of definite clause logic programs, a major source of inspiration for AKL, is described and discussed.

8.1.1 A Constraint Logic Programming Model

A computation in constraint logic programming (CLP) is a proof-tree, obtained by an extended form of input resolution on Horn clauses [Jaffar and Lassez 1987; Lloyd 1989]. The states are negative clauses, called *goal clauses*. The program to be executed consists of an initial state, the *query*, and a (finite) set of

program clauses. The computation proceeds from the initial state by successive reductions, transforming one state into another. In each step, an atomic goal of a goal-clause is replaced by the body of a program clause, producing a new goal clause. When an empty goal clause is reached, the proof is completed. As a by-product, an *answer constraint* for the variables occurring in the query has been produced.

Syntax of Programs

The syntax of CLP programs is defined as follows.

⟨program clause⟩ ::= ⟨head⟩ :- ⟨sequence of constraint atoms⟩, ⟨body⟩

⟨head⟩ ::= ⟨program atom⟩

⟨body⟩ ::= ⟨sequence of program atoms⟩

The letter C stands for a sequence of constraint atoms, and the expression $\sigma(C)$ stands for the conjunction of these constraints (or **true** if the sequence is empty). The *local variables* of a clause are the variables occurring in the clause but not in its head.

The computation states are goal clauses, but for the sake of easier comparison with the AKL model, we use a simplified and-box.

⟨and-box⟩ ::= $\mathbf{and}(\langle\text{sequence of program atoms}\rangle)^{\langle\text{constraint}\rangle}$

The *clausewise reduction* rule

$\mathbf{and}(R, A, S)^{\theta} \Rightarrow \mathbf{and}(R, B, S)^{\theta \wedge \sigma(C)}$

is applicable if $\theta \wedge \sigma(C)$ is satisfiable and if a clause

A :- C, B

can be produced by replacing the formal parameters of a program clause with the actual parameters of the program atom A, and replacing the local variables of the clause by variables not occurring in the rewritten and-box.

Syntax of Goals

A *configuration* is an and-box. The *initial* configuration is of the form

$\mathbf{and}(R)^{\mathbf{true}}$

containing the query (the sequence of program atoms R). Configurations of the form $\mathbf{and}()^{\theta}$ are called *final*. The constraint θ of a final configuration is called an *answer*. An answer describes a set of assignments for variables for which the initial configuration holds, in terms of the given constraint system.

For completeness, it is necessary to explore all final configurations that can be reached by reduction operations from the initial configuration. If this is done, the union of the sets of assignments satisfying the answers contains all possible assignments for variables that could satisfy the initial goal.

Especially, it is necessary to try all (relevant) program clauses for a program atom. (This is quite implicit in the above formulation.) The computation model is nondeterministic. This clause search nondeterminism will necessarily be

made explicit in an implementation. Therefore, the computation model is extended to make clause search nondeterminism explicit.

This is done by grouping the alternative and-boxes by or-boxes.

⟨global goal⟩ ::= ⟨and-box⟩ | ⟨or-box⟩

⟨or-box⟩ ::= **or**(⟨sequence of global goals⟩)

In the context of CLP a *computation rule* will select atomic goals for which all clauses will be tried. This remains in the model. We reify the clause selection nondeterminism that appears when selecting possible clauses for an atomic goal.

The corresponding rewrite rule, called definitionwise reduction, creates an or-box containing all and-boxes that are the result of clausewise reduction operations on a selected program atom.

The *definitionwise reduction* rule

$$\mathbf{and}(R, A, S)^{\theta} \Rightarrow \mathbf{or}(\mathbf{and}(R, B_1, S)^{\theta \wedge \sigma(C1)}, \ldots, \mathbf{and}(R, B_n, S)^{\theta \wedge \sigma(Cn)})$$

unfolds a program atom A using those of its clauses

$$A :- C_1, B_1, \ldots, A :- C_n, B_n$$

for which $\theta \wedge \sigma(C_i)$ is satisfiable (the *candidate clauses*). Again, the actual parameters of A have been substituted for the formal parameters of the program clauses, and the local variables of the clause have been replaced by variables not occurring in the and-box. When there are no candidate clauses, an or-box with no alternative and-boxes **or**(), the empty or-box or **fail**, is produced.

Definitionwise reduction is sufficient to model the behaviour of SLD-resolution and related constraint logic programming models.

8.1.2 Definite Clauses in AKL

We now relate AKL to SLD-resolution, the computation model underlying Prolog, by a translation from definite clauses into AKL and by an interpreter for definite clauses written in AKL.

Translating Definite Clauses

We will show that this basic Prolog functionality is available in AKL by mapping definition clauses and SLD-resolution on corresponding AKL concepts. No formal proof is given, but the discussion should make the correspondence quite clear.

Let *definite* (*program*) *clause*, *definite goal*, *SLD-derivation*, *SLD-refutation*, and related concepts be defined as in Lloyd [1989]. Only the computation rule that selects the leftmost atom in the goal, as in the case of Prolog, will be considered.

For emulation of SLD, we use the constraint system of finite trees.

We further assume that all definitions have at least two clauses. If this is not the case, dummy clauses of the form p ← fail are added. This is necessary only to

get *exactly* the same derivations as in SLD-resolution, and may otherwise be ignored when programming relational programs in AKL.

Let a *definite clause*

$p(T_1, \ldots, T_n) \leftarrow B$

be translated into AKL as

$p(X_1, \ldots, X_n)$:- true ? $X_1=T_1, \ldots, X_n=T_n$, B.

putting equality constraints corresponding to the head arguments in the body, and interpreting the sequence of goals in the body B as AKL composition. Note that the arguments of this AKL clause are different variables, and that the guard is empty.

Similarly, let *definite goals*

$\leftarrow B$

be translated into AKL as

B

As an example, assume in the following the above translation of the definitions of some predicates p, q, and r, which have at least two clauses each. The definition of p is

p :- true ? B_1.
p :- true ? B_2.

The execution of the goal

p, q, r

proceeds as follows. Computation begins by unfolding each of the atoms with its definition, and with further computation within the resulting choice statements. However, as the guards are empty, these are immediately solved.

(true ? B_1 ; true ? B_2), “choice for q”, “choice for r”

Since q and r have at least two clauses (this is the reason for adding dummy clauses to unit clause definitions), the and-box is also stable. Computation may now proceed by trying the alternatives B_1 and B_2.

B_1, “choice for q”, “choice for r”

B_2, “choice for q”, “choice for r”

Computation will then proceed in both alternatives by unfolding the atoms in the bodies B_i, and by telling the equality constraints corresponding to head unification. Failure will be detected before trying alternatives again.

If we analyse the above computation in terms of SLD-resolution, we see that unfolding essentially corresponds to finding variants of program clauses for the atom in question. Nondeterminate choice corresponds to a first (imaginary) stage in the resolution step, where a clause is chosen, and the subsequent telling of the constraints to a second stage, where the substitution is applied.

If we identify AKL computation states of the form

$\theta_1, \ldots, \theta_i$, "choice for p_1", $\theta_{i+1}, \ldots, \theta_j$, "choice for p_k", $\theta_{j+1}, \ldots, \theta_k$

with the definite goals

$(\leftarrow p_1, \ldots, p_k)\theta_1 \cdots \theta_i\theta_{i+1} \cdots \theta_j\theta_{j+1} \cdots \theta_k$

the sequence of *stable* AKL computation states in one alternative branch of the computation corresponds to an *SLD-derivation* using the original definite clauses.

If we further identify a failed AKL computation state

fail

with the empty clause, the failed branches will correspond to *SLD-refutations.*

Of course, the soundness and completeness results for SLD-resolution carry over to this subset of AKL, but stronger results, summarised in Chapter 4, hold for a more interesting logical interpretation of a larger subset of AKL.

Interpreting Definite Clauses

In the translation of definite clauses in the previous section, a nondeterminate choice operation was necessary for each resolution step. Instead, the effect of branching in an SLD-tree can be achieved by recursion in different guards, as shown in the following interpreter, adapted from Ken Kahn's "or-parallel Prolog interpreter in Concurrent Prolog" [Shapiro 1986a].

```
solve([]).
    → true.
solve([A | As]) :-
        clauses(A, Cs)
    → resolve(A, Cs, As).

resolve(A, [(A :- Bs) | Cs], As) :-
        append(Bs, As, ABs),
        solve(ABs)
    ?   true.
resolve(A, [C | Cs], As) :-
        resolve(A, Cs, As)
    ?   true.

append([], Y, Z) :-
    → Y = Z.
append([E | X], Y, Z0) :-
    → Z0 = [E | Z],
        append(X, Y, Z).
```

Definite clauses are represented as follows.

```
clauses(member(_, _), Cs) :-
    → Cs = [ (member(E, [E | _]) :- []),
             (member(E, [_ | R]) :- [member(E,R)]) ].
clauses(ancestor(_,_), Cs) :-
    → Cs = [ (ancestor(X,Y) :- [parent(X,Y)]),
             (ancestor(X,Y) :- [parent(X,Z), ancestor(Z,Y)]) ].
clauses(parent(_,_), Cs) :-
    → Cs = [ (parent(a,ma) :- []),
             (parent(a,fa) :- []),
             (parent(b,mb) :- []),
             (parent(b,fb) :- []),
             (parent(fa,c) :- []),
             (parent(mb,c) :- []) ].
```

At the time of its discovery it was regarded as a "death blow" to Concurrent Prolog, as the then existing or-parallel implementations of Prolog were very inefficient. Since then, efficient or-parallel implementations have appeared, such as Aurora [Lusk et al. 1988; Carlsson 1990] and Muse [Ali and Karlsson 1990; Karlsson 1992].

The technique used for the above interpreter can also be used for programming in AKL, but it is sometimes awkward. For simple programs, such as member, with only one call in the body, the technique is fairly straight-forward, but the resulting program loops in AKL.

```
member(E, [E | R]).
member(E, [_ | R]) :- member(E, R) ? true.
```

The interpreter uses a *success continuation* with the remaining calls in the body, which is passed to each recursive instance of solve. In the member definition, the continuation is empty and can be ignored. The ancestor definition (in the clauses/2 definition) has two calls in its body, and can be translated as follows.

```
ancestor(X, Y, C) :- parent(X, Y, C) ? true.
ancestor(X, Y, C) :- parent(X, Z, ancestor(Z, Y, C)) ? true.

parent(a, ma, C) :- call(C) ? true.
…
```

Ordinary enumeration can be combined with guard nesting, e.g., as follows.

```
ancestor(X, Y) :- parent(X, Y) ? true.
ancestor(X, Y) :- ( parent(X, Z) ? ancestor(Z, Y) ) ? true.
```

To this particular case, with nondeterminate choice being performed without siblings in the guard, correspond particularly efficient implementation techniques, since no copying (or sharing) of active goals is necessary.

8.1.3 Prolog Particulars

Prolog extends SLD-resolution with

- the *pruning* operation cut