Statically Allocated Prolog—Hardware Byrd Boxes*

Alan Mycroft

University of Cambridge, Computer Laboratory
JJ Thomson Avenue, Cambridge CB3 0FD, UK
am@cl.cam.ac.uk

Draft version 28/5/02—for updates see
www.cl.cam.ac.uk/users/am/research/sa

Abstract. We show how the Byrd Box model of Prolog execution can be used to identify a subset of Prolog which can be statically allocated. This subset can then be compiled directly to hardware such as FPGAs thus allowing Prolog to be used as a form of behavioural hardware description language.

1 Introduction

Byrd Boxes (or Procedure Boxes) were originally devised by Lawrence Byrd as a mental model for use during Prolog program debugging—see for example Clocksin and Mellish [4]. The behaviour of each invocation of a predicate is seen as a box (see Fig. 1) with four ports named *call*, *exit*, *redo* and *fail*. The idea is that behaviour of such an invocation can be represented by debugging tools by reporting to the user *call* when the invocation starts, *exit* when the predicate succeeds, *redo* when the latest success is rejected by backtracking and *fail* when no more solutions are to be found (finite failure). Note particularly (since we will vary this later) the standard model for this is that each *invocation* creates a new Byrd Box. It can be useful to view each predicate as having (in object-oriented terminology) a *class* of Byrd Boxes and seeing each invocation as providing a new *instance* of this class which interacts with its environment along the four ports.

We wish to take a more implementation-oriented viewpoint, indeed we wish to implement Byrd Boxes as hardware, so it is convenient to clarify the rôle of ports a little. The port *call* will in general have an associated tuple of variables or ($\vec{x}$ in Fig. 1) with which incoming values are unified.[1] In hardware terms (or

* Paper to celebrate the 25th anniversary of the Computer Science Department of the Universidad Politécnica de Madrid.

[1] Since a predicate may have several clauses, in the Byrd Box model it is more natural to see the initial *predicate* invocation as always succeeding, but that unification with individual *clause heads* as being determined after the box has been created and before the box responds with *exit* or *fail*.

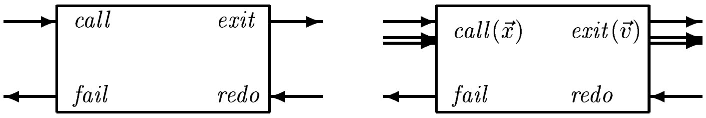

Fig. 1. A Byrd Box and the bundled-data view of a Byrd Box

indeed in value-passing process calculi terms, see later) it is convenient to see $call(\vec{x})$ as a single wire *call* on which a pulse is applied to start computation and to clock in the so-called bundled-data from a separate bus, $\vec{x}$, of wires. Similarly, $exit(\vec{v})$ will be seen as a single wire *exit* which transmits a 'data available' pulse and bundled-data, $\vec{v}$, on an associated bus. By contrast the ports *redo* and *fail* are by nature pure signalling wires.

1.1 Static Allocation

A system is *statically allocated* if its storage can be allocated statically on a once-per-procedure basis, rather than dynamically on a once-per-invocation or once-per-instance basis. This is explicit in languages like Java where in the definition

```
class A { int m; static int n; };
```

where there is a distinct copy of member `m` for each instance of `A` while member `n` is shared between all instances.

Static allocatability is a useful criterion for systems to be implemented in hardware, e.g. on FPGAs or as Systems-on-a-Chip (SoC), in that it allows data to be stored [on buses or in registers] along with functions [functional units] that manipulate them. In particular it avoids the "von Neumann bottleneck" whereby dynamically allocated values need to be accessed via a narrow "address-decode" interface using a multiplexer.

1.2 Byrd Boxes as a Semantic Model for Prolog

One other effect of showing how to implement Prolog in terms of Byrd Boxes is that provides an immediate translation into process-calculi. In particular, the statically allocated subset of Prolog maps to a finite state subset of CCS, whereas full Prolog can be implemented in full CCS. (It requires essential use of renaming, hiding and recursion operators of CCS, but otherwise does not require the higher-order features of the Pi-calculus.)

In this framework a Prolog predicate r would be represented by a CCS behaviour Q_r. Each dynamic occurrence of a predicate would create Byrd Box whose behaviour is an initial subsequence of the following extended regular expression:

$$c(\vec{x}).\,[\overline{e}(\vec{v}).r]_0^\infty\,.\overline{f}.\mathrm{nil}$$

Here c, e, r and f represent *call*, *exit*, *redo* and *fail*. Allowing the infinite sequence

$$c(\vec{x}).\,(\overline{e}(\vec{v}).r)^\infty$$

is necessary for predicates which succeed with infinitely many potentially different success substitutions.

1.3 Paper Structure and Related Work

After this introduction, the paper first considers static allocation in pure control-flow Prolog (Section 3) and then the effect of adding data (Section 4). Section 5 considers a process calculus view of our Byrd Box model and Section 6 concludes.

Related work includes that of Mycroft and Sharp [6,7] on statically allocated functional languages, Jahier, Ducasse and Ridoux [5] on the semantics of Byrd Boxes and Charatonik, Gordon and Talbot [3] on the more general case of ambients with finite control.

2 Notation

We use the usual notation for Prolog with abstract syntax as follows. Let *Var* be a set of variables (ranged over by x), *Func* be a set of function symbols (ranged over by f) and *Pred* be a set of predicate symbols (ranged over by p). Predicate and function symbols each have a given arity (≥ 0), k_p and k_f. Terms (t), atoms (a), clauses (c) and programs (π) have syntax:

$$
\begin{aligned}
t &::= x \mid f(t_1, \ldots, t_{k_f}) \\
a &::= p(t_1, \ldots, t_{k_p}) \\
c &::= a_0 \texttt{ :- } a_1, \ldots, a_m. \quad (m \geq 0) \\
\pi &::= c_1 \cdots c_n \quad (n > 0)
\end{aligned}
$$

A program π has a distinguished predicate symbol (the one appearing in the head of c_n, say) which defines an 'external world' interface to π. In examples we follow the usual Prolog stropping conventions and write variables in upper case, and function and predicate symbols in lower case.

3 Control Flow

In this section we construct Byrd Boxes for predicate symbols, without considering data values. This is equivalent to restricting to nullary predicate symbols, but the techniques used will also be valid for a larger subset of Prolog.

Our construction will be *compositional*, in other words the Byrd Box for a predicate will be constructed in terms of the Byrd Boxes of predicates it directly calls, and will not depend on the internal structure of these Byrd Boxes, save as provided by their *call/exit/redo/fail* interface.

Because there are no data, we can assume that ports for *call* and *exit* are merely single wires; a request is transmitted as an event on this wire (for synchronous digital logic this might be achieved by holding the wire high while a single upward-going clock edges occurs; for asynchronous logic it might simply be an upward-going transition). Often we need to represent an output event which can never happen; this is achieved by tying the output wire to 0. Inputs which are ignored are simply left unconnected.

Having done this, there are three primitive predicates for which we need to define Byrd Boxes. These are `true` (which always succeeds) `false` (which immediately fails) and `loop` which computes forever without ever signalling *exit* or *fail*. Their internal structure is given in Fig. 2. In passing we observe that extending these boxes to data-passing is easy; only `true` requires any real work, and that merely to connect the incoming data from *call* to the outgoing data to *exit*.

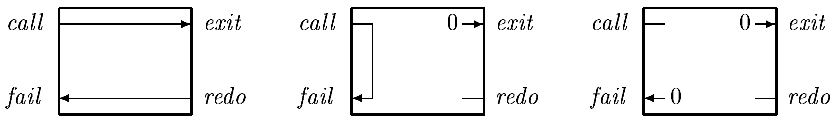

Fig. 2. Structure of Byrd Boxes for `true`, `false` and `loop`

Now we turn to the problem of constructing a Byrd Box for a user-defined predicate based on the Byrd Boxes of predicates which it calls. We do this by exhibiting Byrd Boxes which represent (sequential) conjunction and disjunction of their constituents. These are given graphically in Fig. 3 Note that sequential

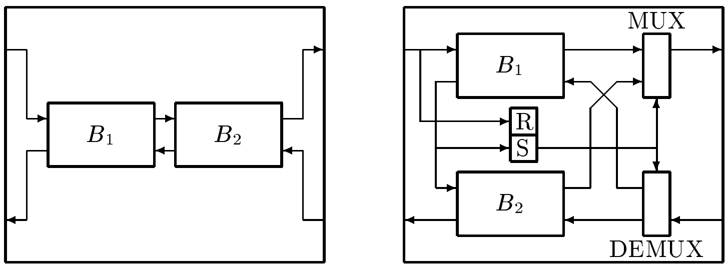

Fig. 3. Byrd Box constructions for conjunction (left) and disjunction (right)

conjunction B_1, B_2 merely connects the (success) *exit* of B_1 with the *call* input of B_2 and similarly exhaustion of backtracking possibilities of B_2 signals its *fail* which is connected to the *redo* line of B_1.

Prolog's sequential disjunction is a little more complicated and requires additional state in the guise of a flip-flop. For a clocked logic implementation this would be a synchronous RS flip-flop. Sequential disjunction, written $B_1; B_2$, represents the sequential backtracking behaviour obtained by having multiple clauses defining a predicate symbol. When *call* is invoked on $B_1; B_2$ then we reset the flip-flop (representing the fact that B_1 is active) and pass successful *exit* signals from B_1 to the caller. The caller's *redo* signal is likely routed back to B_1's *redo* port using a demultiplexer. When (and if) B_1's success *exit*s are exhausted and B_1 signals *fail* this causes the routing flip-flop to be set and B_2 to be called. Now success *exit*s of B_2 are routed to the output port and *redo* from a called procedure is routed back to B_2's *redo* port. Finally, when (and if) B_2's *fail* port is activated, the composition signals *fail*.

Note that $(B_1, B_2), B_3$ is identical to $B_1, (B_2, B_3)$. Note also that $(B_1; B_2); B_3$ differs textually, but not behaviourally, from $B_1; (B_2; B_3)$. Indeed we could implement an n-way sequential disjunction by extending the binary disjunction in an obvious way and replacing the routing multiplexer/demultiplexer the controller's flip-flop with an n-state register.

In the absence of direct or indirect recursion this implementation of Prolog in Byrd Boxes accurately models the semantics of data-free Prolog. (When we consider data values we need to be a little more careful).

3.1 Cut

The cut (`!`) predicate can also be expressed in this framework—not as a Byrd Box (a partial justification being that it is extra-logical) but as an operator (like conjunction or disjunction above) which causes the appropriate wiring together of its neighbours. This is shown graphically for conjunction in Fig. 4.

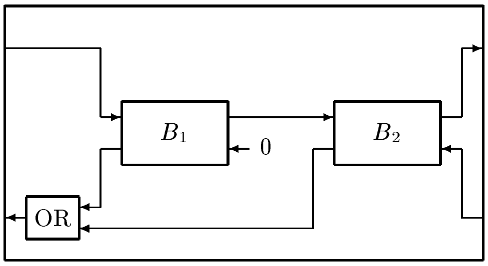

Fig. 4. Byrd Box construction for $B_1, !, B_2$

Note that the semantics of *cut* is such that failing to a *cut* fails the whole predicate in which it occurs. Thus when, as is typical, a cut occurs (perhaps within a conjunction) inside a disjunction, the OR-gate shown at the bottom left of Fig. 4 will instead appear at the *fail* port of the whole predicate, not just the clause (not shown).

3.2 Recursion

In general recursive programs may not be statically allocated; indeed statically allocated programs are equivalent to Finite State Automata (just as even the most powerful computer today is a Finite State Automaton) whereas general recursion is Turing-powerful.

We therefore now turn to consider when a recursive program can be implemented using static allocation. Note that we are seeking syntactic rules, and wish to avoid the slippery slope of undecidability represented by arguments of the form “well, a rather deep bit of mathematical reasoning implies that XXX holds”. In functional languages, a well-known rule (Mycroft and Sharp [6] give more details) is that when recursion calls are limited to being tail-recursive then they may be implemented as loops rather than using a stack of calls.

For Prolog a particularly fruitful example to consider (this unusual formulation of a down-counter is chosen to to illustrate the conjunction and disjunction operators given above) is that of

```
p(X,Z) :- X=Z.
p(X,Z) :- X>0, Y is X-1,  p(Y,Z).
```

Using B_1 to refer to the Byrd Box for ‘`X=Z`’ and B_2 to that of ‘`X>0, Y is X-1`’ results in the problem demonstrated in Fig. 5—we wish the Byrd Box P of predicate p to have behaviour satisfying the indicated equation. In general, to

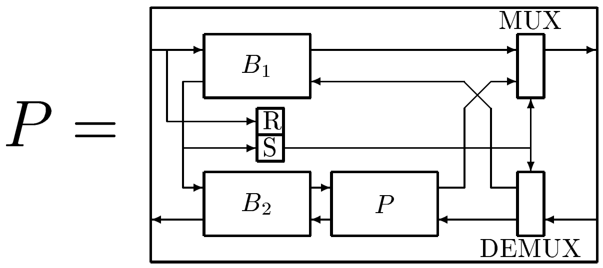

Fig. 5. Byrd Box for simple recursive procedure

implement Prolog correctly, we *require* the inner instance of Byrd Box P to be distinct from the outer one.

However, the special case of Prolog tail recursion can be seen here in a new way. Suppose (as is the case in the example) that B_2 is a partial function, i.e. if it issues an *exit* event then all *redo* events will cause a *fail* event.[2] This can also be expressed that the circuit for B_2 has a simple wire (possibly via a router whose path is activated after B_2's initial *exit*) connecting its *redo* port to its *fail* port. Then all *fail* events from the inner P will cause *fail* events on the outer P.

[2] There is a subtlety here, in order to preserve termination behaviour when optimising tail recursion we cannot regard predicates as partial functions if they succeed once and then loop forever (fail infinitely) instead of issuing a *fail* event.

It is also clear that all *exit* events of the inner P will be routed as *exit* events of the outer P and indeed all will arrive from an instance of B_1.

Continuing in this way, we can reason that the *call* to the inner P can be safely replaced by a loop to re-activate the outer P (see Fig. 6).

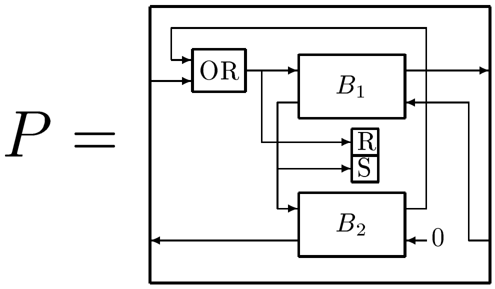

Fig. 6. Byrd Box for tail-recursive implementation of P

There are several things worth noting. Firstly, we have exploited the partial-function nature of B_2 in the transformation and hence 0 is supplied for its *redo* port—in essence Byrd Box B_2 is abandoned when it first succeeds, so that it becomes re-invoked by another *call* without first having completed itself by issuing a *fail*. It turns out that this just works in the hardware version, but we have to be a little more careful when translating to statically allocated process calculus expressions (see later).

Secondly, the RS flip-flop is now unused and may be removed. Instead of the original program (with two clauses for P) needing a single flip-flop to remember which of the two clauses is active (for use by *redo*), we now only have a single state. However, now consider a slightly more general tail-recursive program:

```
p(X,Z) :- q(X,Z)
p(X,Z) :- r(X,Z).
p(X,Z) :- X>0, Y is X-1,  p(Y,Z).
```

With three clauses, this modified predicate `p` requires three-states (two flip-flops) before tail-recursion removal, but still requires two states (one flip-flop) after tail-recursion removal.

3.3 Static Allocation Criterion

For pure control flow, then the static allocatability criterion (at most one instance of a Byrd Box for a given procedure is ever active at any time) is that recursive calls must occur in tail-context. In the framework of Byrd Boxes this simply means that the *exit* port of such calls is always routed to the *exit* port of the predicate being defined (i.e. no conjunct follows such a call) and the *fail* port of such calls is always routed to the *fail* port of the predicate being define (i.e. no disjunct follows such a call *and* any conjunct preceding it are partial functions). The conditions can be appropriately weakened for predicates containing *cut*.

Note that, following Mycroft and Sharp's [7] treatment of the functional language SAFL, the compilation to Byrd Boxes presented here is *resource aware*. By this we mean, given a program satisfying the static allocation rules, we translate each predicate into a separate hardware functional unit with four ports. Multiple calls to a predicate are implemented by multiplexing date from call sites into the predicate's *call* port, and result data routed back appropriately.

4 Data Flow

So far we have said very little about data, instead preferring to present the control-flow issues. Clearly, to be able to implement Prolog computation with statically allocated storage space, we need to remove constructors—this leads us to considering a form of Datalog [2] which forbids all but nullary function symbols. Note that there is no problem in allowing (bounded) arithmetic, and indeed we used this in the tail-recursion example above.

There is one more problem to be addressed which arises from the nature of logic programming—that of unbound variables. Consider the Prolog program:

```
p(X,Y).
q(A,B) :- p(1,A), p(2,B).
```

The predicate `p` seems to need to create a potentially unlimited number of 'fresh' unbound variables. Accordingly adopting the unification-free discipline of Apt and Etalle [1] or a scheme which requires the Program to be well-moded (arguments are marked as input (`ground` $\Rightarrow$ `ground`) or output (`var` $\Rightarrow$ `ground`).

In such a moded system values are ground; input variables are bound to them on *call* and output variables are bound to them on *exit*.

5 A Process-Calculus View

We have already remarked that a Byrd Box P_r for predicate r can be seen as having behaviour Q_r which refines a specification of the form

$$Q_r = c(\vec{x}).\,[\overline{e}(\vec{v}).r]_0^\infty\,.\overline{f}.\mathrm{nil}$$

where c, $\overline{e}$, r, $\overline{f}$ are ports private to the invocation of r representing respectively *call*, *exit*, *redo* and *fail*.

In particular, the behaviours of *single calls* to the three predicates `true`, `false` and `loop` can be represented as

$$\begin{aligned} Q_{\mathtt{true}} &= c.\overline{e}.f.\overline{f}.\mathrm{nil} \\ Q_{\mathtt{false}} &= c.\overline{f}.\mathrm{nil} \\ Q_{\mathtt{loop}} &= c.\mathrm{nil} \end{aligned}$$

The use of 'nil' reflects the fact that the boxes take no more part in the computation after their work is complete. Invocation of such processes in general will be done by creating a new instance of the process, i.e. embedding the current behaviour b to give a form such as:

$$(Q[c'/c, e'/e, r'/r, f'/f] \mid b) \setminus \{c', e', r', f'\}.$$

Recall that if a program is statically allocatable (i.e. each predicate in it is statically allocatable) then instead of creating a new instance of a process Q on each call then a single instance of each will suffice. However, we must then modify the predicates so that they are *serially reusable*:

$$\begin{aligned} Q_{\texttt{true}} &= c.\overline{e}.r.\overline{f}.Q_{\texttt{true}} \\ Q_{\texttt{false}} &= c.\overline{f}.Q_{\texttt{false}} \\ Q_{\texttt{loop}} &= c.\mathrm{nil} \end{aligned}$$

Each Byrd Box can similarly be translated into a CCS expression.

A statically allocatable Prolog program can thus be translated into CCS expression consisting of one parallel component for each predicate definition. An external interaction $\overline{c}(\vec{v})$ to the *call* channel of the process name Q_q corresponding to the distinguished 'external world' predicate symbol q then initiates the computation; the observer sees results by listening on channels e (*exit*) and f (*fail*) belonging to Q_q.

Charatonik, Gordon and Talbot [3] examine "finite control" (a form of static allocatability) in the more general framework of the ambient calculus.

6 Conclusions

We have shown that Byrd Boxes can be seen as providing an execution model for Prolog. Moreover, we have identified a *statically allocatable* subset of Prolog which can be executed by associating a single Byrd Box with each predicate symbol. Each Byrd Box has only finite control and can thus be implemented directly in hardware. A finite state CCS model of statically allocated Prolog falls out directly.

One issue perhaps meriting further work is that of evaluation order. We have adhered to the Prolog ordering (left-right sequential both for selected atom and selected clause) which enabled us to handle *cut*. However, hardware is naturally parallel and it would be interesting to investigate whether the parallelism inherent in and- and or-resolution choice can be exploited in (finite-state) hardware.

Acknowledgements

This work was supported by (UK) EPSRC grant GR/N64256 "A Resource-Aware Functional Language for Hardware Synthesis"; travel to and from UPM Madrid was provided by Iberia.

References

1. Apt K.R. and Etalle, S. On the Unification Free Prolog Programs. Lecture Notes in Computer Science: Proc. 18th MFCS. vol. 711, Springer-Verlag, 2000.
2. Chandra, A.K. and Harel, D. Horn Clause Programs and Generalizations, Journal of Logic Programming, vol. 2(1), 1985.
3. Charatonik, W., Gordon, A.D. and Talbot, J.-M. Finite-Control Mobile Ambients. Lecture Notes in Computer Science: Proc. ESOP 2002. vol. 2305, Springer-Verlag, 2002.
4. Clocksin, W.F. and Mellish, C.S. Programming in Prolog. Springer-Verlag, 1981.
5. Jahier, E., Ducasse, M. and Ridoux, O. Specifying Prolog Trace Models with a Continuation Semantics. Lecture Notes in Computer Science: LOPSTR Selected Papers, vol. 2042, Springer-Verlag, 2000.
6. Mycroft, A. and Sharp, R.W. A Statically Allocated Parallel Functional Language. Lecture Notes in Computer Science: Proc. 27th ICALP, vol. 1853, Springer-Verlag, 2000.
7. Mycroft, A. and Sharp, R.W. Hardware Synthesis using SAFL and Application to Processor Design. Lecture Notes in Computer Science: Proc. CHARME'01, vol. 2144, Springer-Verlag, 2001.