Logic Programming and Functional Nets
Preliminary version[1]

Alan Mycroft[2]

Computer Laboratory, Cambridge University
New Museums Site, Pembroke Street, Cambridge CB2 3QG, UK

am@cl.cam.ac.uk

Abstract. In general, programming languages and paradigms have each been associated with their own underlying calculus. Thus functional programming has λ-calculus, logic programming has inference systems and concurrent programming has various calculi: Petri nets, π-calculus, CCS, theoretical CSP and the like. Odersky recently showed how a development "Functional Nets" of the Join-calculus can express ideas from Functional, Concurrent and Object-Oriented languages within a single framework in a manner which allows constructs from these disparate languages to interact. Here we examine how logic programming concepts fit into this framework.

1 Introduction

In general, programming languages and paradigms have each been associated with their own underlying calculus. Thus functional programming has λ-calculus, logic programming has inference systems and concurrent programming has various calculi: Petri nets, π-calculus, CCS, theoretical CSP and the like.

Odersky recently showed how a development "Functional Nets" of the Join-calculus can express ideas from Functional, Concurrent and Object-Oriented languages within a single framework in a manner which allows constructs from these disparate languages to interact.

One can also see this paper as an delayed continuation of Jones and Mycroft's [2] work on semantics of Prolog. In that paper, denotational and small-step operational semantics for Prolog were given; these are arguably a functional language implementation and a hardware-implementable machine. Here we give a semantics (or implementation if one wishes to consider it that way!) of Prolog via Odersky's Functional Nets [3].

The use of Functional Nets somewhat generalises Odersky's in that we will feel free to use algebraic datatypes such a lists, whereas Odersky's work on the basis of Functional Nets quite properly explains how they can be constructed *within* the language.

[1] See `http://www.cl.cam.ac.uk/users/am/research/funnel/` for a fuller version of this paper and sample programs.

[2] This work was partly done during sabbatical leave at AT&T Laboratories Cambridge.

2 Theoretical Background

The abstract syntax of logic programs for our purposes is as follows. Let *Var* be a set of variables (ranged over by x), *Func* be a set of function names (ranged over by f) and *Pred* be a set of predicate names (ranged over by p). Predicate and function names each have a given arity (≥ 0), k_p and k_f. Terms (t), atoms (a), goals (g), clauses (c) and programs (p) have syntax:

$$
\begin{aligned}
t &::= x \mid f(t_1, \ldots, t_{k_f}) \\
a &::= p(t_1, \ldots, t_{k_p}) \\
g &::= a_1, \ldots, a_r \\
c &::= a_0 \texttt{ :- } a_1, \ldots, a_m \\
p &::= c_1; \ldots; c_n \texttt{ ?- } g
\end{aligned}
$$

The final goal in a program (following '`?-`' is called the query). The letter b also used to range over atoms in clauses. This language is a subset of Prolog, and will be referred to as Prolog for convenience in the following.

Given two atoms a and b, we have the notion of *most-general unifier*, that is a substitution θ such that $\theta(a) = \theta(b)$ which makes minimum identifications or reports failure when no unifier exists.

A *new variant* of a clause is the clause renamed (bijective substitution) so as to have new variables, distinct from those under consideration.

The semantics of Prolog is often defined in terms of SLD-trees. An SLD-tree is defined as follows. Given a program $c_1; \ldots; c_n$ `?-` g, an SLD-tree has a root node labelled with g. Suppose a node is labelled with goal $a_1, \ldots, a_r$. An SLD-tree identifies a distinguished atom a_i (the *selected atom*) from the goal and moreover has one child for each new clause variant b_0 `:-` $b_1, \ldots, b_m$ of $c_1; \ldots; c_n$ from the program whose head b_0 unifies with a_i. Such a child is labelled with (the *resolvent*) goal

$$\theta(a_1, \ldots, a_{i-1}, b_1, \ldots, b_m, a_{i+1}, \ldots, a_r)$$

where θ is the most-general unifier of a_i and b_0. An SLD tree is determined by its choice strategy for the selected atom. Nodes labelled with the empty goal represent success, and the composition of substitutions (restricted to variables occurring in the root goal) encountered from root to such a node is the answer.

The usual behaviour of Prolog is to consider a particular SLD-tree, the one in which the selected atom from $a_1, \ldots, a_r$ is always a_1. Moreover this SLD-tree is searched in a depth-first left-right (i.e. textual order of clauses) manner.

Depth-first SLD-tree searching defines an extremely sequential execution mechanism for Prolog. One alternative dimension is to search a given SLD-tree in a more breadth-first fashion (this is generally called OR-parallelism). Another evaluation choice is to arrange that, given a goal $a_1, \ldots, a_r$, the atom-selection strategy selects goals $a_1, \ldots, a_r$ in some order (or even concurrently) before selecting goals b_j which occur in the resolvent of atom a_i and a given clause:

$$\theta(a_1, \ldots, a_{i-1}, b_1, \ldots, b_m, a_{i+1}, \ldots, a_r).$$

This is generally called AND-parallelism. Note that in general this is more problematic than OR-parallelism due to contention during unification—we may find that two concurrent resolutions are attempting to unify a variable with two incompatible terms—and a way of dealing with this is required.

In general it is often useful to consider Prolog execution as a search of an AND-OR tree.

Petri Nets are another formalism whose elaboration also explores AND-OR graphs. See Fig. 1 for an pictorial example. When playing the "token game" (defined below), the token on the leftmost s_1 place can either fire transition t_1 or transition t_2 (this is an OR-choice). If transition t_1 fires then the original token on s_1 is replaced by two tokens, one each on s_2 and s_3 (this corresponds to an AND-choice).

Formally[1] a Petri net N is a quadruple (S, T, F, M_0) where S is a set of *places* (drawn with circles, corresponding to states), T is a set of *transitions* (drawn with boxes), $F \subseteq S \times T \cup T \times S$ is the *flow relation* and $M_0 \subseteq S$ is the *initial marking*. The marking evolves by repeated occurrences of the *firing rule*. Let $t \in T$ be a transition, we write $\cdot t = \{s | sFt\}$ (the pre-set of t) and $t\cdot = \{s | tFs\}$ (the post-set of t); then the firing rule says that a marking M can evolve to a marking $M \setminus \cdot t \cup t\cdot$ whenever $\cdot t \subseteq M$. Moreover, if $G \subseteq T$ is a non-empty set of transitions then the members of G can fire *concurrently* providing their pre-sets are disjoint, this is often called a *step*. Note that Petri nets capture finite automata as a special case, these are just finite nets with $(\forall t \in T)|\cdot t| = |t\cdot| = 1$.

Note that there is no requirement for a Petri net to be finite. Indeed Nielsen, Plotkin and Winskel defined the notion of *occurrence net*[2] which is a form of unfolding of a given net which captures all its AND-OR behaviour. Such occurrence nets are acyclic (and moreover tree-like in that they allow path merges only to represent AND-joins and not OR-joins). Of course occurrence nets are infinite for any net which admits unlimited firing sequences.

There is a link between such unfoldings and SLD-trees, via the notion of *coloured Petri nets*. In these tokens carry additional state information rather than being indistinguishable, similarly transitions can choose whether or not to fire based on the values carried by their input tokens and moreover can write new values into their output tokens. Using this model (with token values being goals) a non-deterministic Prolog interpreter can can be written as in Fig. 2 where there is one transition t_i for each clause in the program (transition t_i is enabled when the goal is non-empty and its selected atom unifies with the clause i) plus one transition t_{accept} which is enabled when the goal is empty. Unfoldings of this net are now isomorphic to SLD-trees (there is a family of both determined by the strategy choosing the selected atom).

In general, mapping OR-choice into non-determinism is not very useful for programming—we would have to keep executing a program and hope that the

[1] Many presentations allow for multiple tokens to occur at a single place, but I have rather avoided this here.

[2] Beware: there is more than one definition of this term.

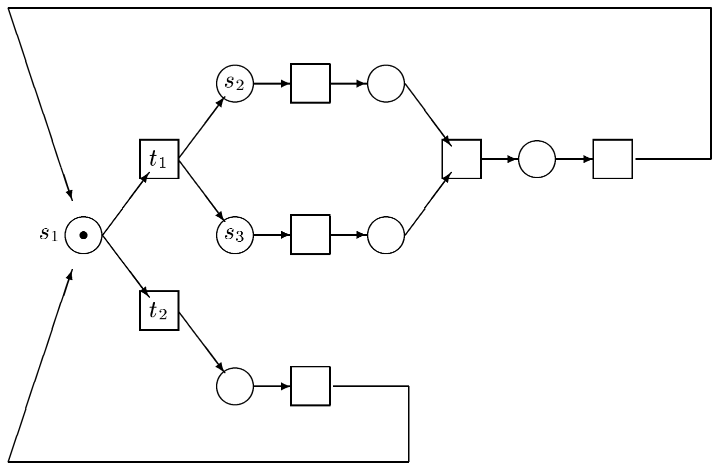

Fig. 1. A simple Petri Net

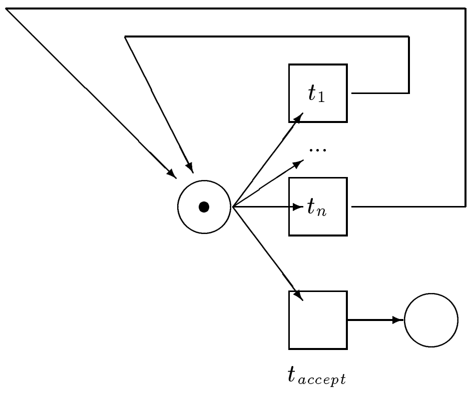

Fig. 2. A Coloured Petri Net Prolog Interpreter

hoped-for answer arises. Thus instead we map OR-choice to search, either sequential or concurrent. We now study this in more detail.

3 Implementing Prolog in Functional Nets

The standard implementation of substitution when implementing Prolog is to implement variables as state-containing objects, as opposed to carrying a substitution as a value and (eagerly or lazily) applying it to a state-less term. Instantiating a variable to a term is implemented as assignment to a private field within the variable. In such implementations `unify` returns a boolean merely indicating whether the unification succeeded or not. Because of the need to undo this assignment when backtracking to explore an (OR-) alternative, it is common to record such instantiations in a data-structure called a *trail*. In the sequential case the trail can be implemented as a stack: a note of the stack depth is taken before performing resolution, unification pushes descriptors of variables as they are instantiated and backtracking pops variable from the stack restoring their state to “uninstantiated”.

Thus a traditional Prolog interpreter looks as follows (in Pseudo-C++):

```
void solve(Goal *g, ClauseList *p, VarToNameMapping *map)
{
    if (g == NULL) map->showanswer();
    else for (Clause *ci in p)
    {   Trail *t = Trail::Note();
        Clause *c = ci->copy();
        Trail::Undo(t);
        if (unify(g->car, c->head))
            solve(append(g->cdr, c->body), p, map);
        Trail::Undo(t);
    }
}
void start(Goal *g, ClauseList *p, VarToNameMapping *map)
{
    solve(g, p, map);
    printf("No more solutions\n");
}
```

I.e. if the Goal is empty then print the corresponding substitution and prompt the user as to whether alternatives are required; otherwise attempt to unify the left-hand-side of a variant `c = ci->copy()` with the first member of the goal `g->car`. If unification succeeds then append the right-hand-side of the variant to the remainder `g->cdr` of the goal. After unification and the recursive call to `solve` it is necessary to restore instantiated variables by calling `Trail::undo(t)`. It is convenient to allow `unify` to fail untidily (in an way which may leave some variables instantiated) and to allow the call `Trail::undo(t)` also to restore these also. Finally, it is sometimes convenient to allow `copy()` to temporarily update variables present in clauses of the original program (e.g. to make `copy()` an $O(n)$ process where n is the size of the clause being copied). The first call to `Trail::undo(t)` can therefore serve to restore the original program too. Of course various tricks are used, append is coded as $O(1)$ and the recursive call is replaced by an explicit stack, but the principle holds.

3.1 Finite Failure

Finite failure (the search of the SLD-tree has been completed, with or without success solutions) is important in that it indicates that the computation is finished as opposed to a possible additional answers being produced in the future. It is also important as a form of negation, but this is rather outwith the scope of this paper.

Sequentially, finite failure just reflects backtracking to the origin. With OR-parallelism we have to keep track of which computations are extant, and we now turn to a Functional Net solution.

4 Compiling Prolog to Functional Nets

The word 'compile' in this section represents forms of evaluation mechanism where the source program structure may be exploited, as opposed to an interpreter. The presentation does not aim at high efficiency, but it is clear that efficiency-gaining adjustments (at the expense of clarity) would be possible. Consider the following, assuming that the program is `cl1`; ...; `cln`. (Here we represent substitutions `s` explicitly rather than by updating the state of the object representing a variable.)

```
def solve(g: List[Atom], ans: List[String*Term], ff: ()->()) =
{
    if (g == []) (print ans; ff());
    else
    {   def f1() & ... & fn() = ff();
         (let (head:-body) = copy(cl1)
          let s = unify(hd(a), head)
          if (s == fail) f1()
          else solve(s(tl(g)) @ s(body), s(ans), f1)
         &
         ...
         &
         (let (head:-body) = copy(cln)
          let s = unify(hd(a), head)
          if (s == fail) fn()
          else solve(s(tl(g)) @ s(body), s(ans), fn)
    }
}
```

```
def start(g: List[Atom], ans: List[String*Term]) =
{
    def finitefail() = print "No more solutions";
    solve(g, ans, finitefail);
}
```

This captures OR-parallel evaluation in a manner which lets us simultaneously understand the semantics and see possible implementation optimisations more clearly. As an example of the latter, note that it would be simple to produce one variant of solve for each predicate symbol p occurring in the program. Then the Functional Net program would have one procedure for each predicate symbol containing actions determined solely by the clauses defining p. This reduces unnecessary tests in a manner characteristic of a compiler, but these actions are seen as semantically justified rather than based on low-level reasoning.

5 Conclusions and Further Work

This paper confirms that Functional Nets are a promising formalism for describing Logic Programming in additional to their expressiveness at capturing functional and object-oriented concepts (and of course in expressing concurrency from their basis in the Join calculus). The nearness of Functional Nets and Join calculus means that the descriptions can easily be seen from both underlying semantic and implementation viewpoints.

Clearly much remains to be done beyond these introductory notes.

Acknowledgments

I am indebted to Martin Odersky for discussions about Functional Nets.

References

1. Jensen, K. Coloured Petri Nets. Basic Concepts. EATCS Monographs of Theoretical Computer Science, Springer-Verlag, 1992.
2. Jones, N.D. and Mycroft, A. Stepwise development of operational and denotational semantics for Prolog. Proc. IEEE intl. symp. on Logic Programming, Atlantic City, 1984.
3. Odersky, M. Functional Nets. Proc. ESOP'2000 Berlin, Springer-Verlag LNCS vol. 1782, 2000. See also `http://lampwww.epfl.ch/funnel/`
4. Shen, K. Studies of AND/OR parallelism in Prolog. PhD thesis, Computer Laboratory, University of Cambridge, 1992.