Heuristics Entwined with Handlers Combined

From Functional Specification to Logic Programming Implementation

Tom Schrijvers
Ghent University, Belgium
tom.schrijvers@ugent.be

Nicolas Wu
University of Oxford, UK
nicolas.wu@cs.ox.ac.uk

Benoit Desouter
Ghent University, Belgium
benoit.desouter@ugent.be

Bart Demoen
KU Leuven, Belgium
bart.demoen@cs.kuleuven.be

Abstract

A long-standing problem in logic programming is how to cleanly separate logic and control. While solutions exist, they fall short in one of two ways: some are too intrusive, because they require significant changes to Prolog's underlying implementation; others are lacking a clean semantic grounding. We resolve both of these issues in this paper.

We derive a solution that is both lightweight and principled. We do so by starting from a functional specification of Prolog based on monads, and extend this with the effect handlers approach to capture the dynamic search tree as syntax. Effect handlers then express heuristics in terms of tree transformations. Moreover, we can declaratively express many heuristics as trees themselves that are combined with search problems using a generic entwining handler. Our solution is not restricted to a functional model: we show how to implement this technique as a library in Prolog by means of delimited continuations.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Functional Programming; D.1.6 [*Programming Techniques*]: Logic Programming; I.2.8 [*Artificial Intelligence*]: Problem Solving, Control Methods, and Search—Graph and tree search strategies

General Terms Languages

Keywords monads, effect handlers, free monad transformer, delimited continuations, logic programming, heuristic tree search

1. Introduction

One of the long standing problems in logic programming has been to find a modular way of expressing search problems. Kowalski's slogan [21],

$$algorithm = logic + control$$

captures the ideal, which is to achieve effective algorithms through the clean separation of problem logic and control mechanism. The programmer should only focus on the logic, while the system provides the control. In practice, however, the control of Prolog systems is often inadequate. As a work-around, programmers resort to intermixing and obscuring their logic with control heuristics that make finding a solution feasible. Several different solutions have been proposed to this problem. Some offer great flexibility but abandon Prolog's execution model altogether, while those that remain faithful to Prolog offer rather limited expressive power. The TOR approach [31, 34] constitutes the current state of the art in the latter class of solutions: it is a light-weight library-based approach that is easily portable to different Prolog systems. However, it suffers from a major deficiency: it lacks proper semantic grounding, and so requires intimate knowledge of the implementation in order to be understood.

We resolve this deficiency by borrowing techniques from functional programming—*monads* and *effect handlers*—to guide the design of a library that is both modular and based on sound principles. Indeed, exploiting the synergy between the functional programming and logical programming paradigms is essential for this work. Because mastery of both fields is not an easy task, this synergy is rarely exploited. Yet, the cross-pollination of ideas solves problems that are otherwise intractable. In this paper we champion such multi-disciplinary work and present the full development of our solution from its functional specification in Haskell to logic programming implementation in Prolog, in order to bring both communities closer together.

Monads [40] have firmly established their place in functional programming as a practical solution to modelling computations that involve side effects and that have a notion of sequentiality. Instances of monads are abound, and much work has been done on reasoning about monadic programs in terms of the monad's implementation details [16], where the concrete instance is of interest.

An emerging approach is to treat monads as an interface, whose laws form a specification [11], which is closer to the formulation of universal algebras as Lawvere theories [22]. Here the implementation is hidden and the monad is *opaque* or abstract. Programs cannot exploit the monad's implementation details, but are restricted to the exposed interface.

In this paper we develop a technique to extend the opaque monad that represents Prolog, in order to reason carefully about nondeterminism and heuristics. An important ingredient of our solution are *effect handlers* [1, 3, 18, 20, 24, 28]. They provide a new approach to defining monads that cleanly separates syntax from semantics, where a syntax tree is first built, and then interpreted in a semantic domain using a *handler*. We combine these effect handlers with the free monad transformer to model and extend an opaque effectful language expressed in monadic style.

Our technical contributions are as follows:

- We show how to obtain modular search heuristics by means of effect handlers, and, in particular, define an unusual *entwine* handler that allows us to express search heuristics as archetypal search trees much like TOR's `merge/2` combinator.

PPDP '14, September 08–10, 2014, Canterbury, United Kingdom.

ACM 978-1-4503-2947-7/14/09…$15.00.
http://dx.doi.org/10.1145/2643135.2643145

- We formulate a functional model that takes the feature-rich nature of modern Prolog systems into account. Furthermore, we use effect handlers and the free monad transformer to minimize the footprint of both the model and the final solution.
- We explain carefully how to transfer our functional solution to modular search heuristics to Prolog. This transfer involves a non-trivial isomorphism between the free monad transformer and the delimited continuation monad transformer.
- We demonstrate the superiority of the effect handlers approach over TOR: effect handlers can express more search heuristics than TOR. Moreover, while there is no obvious way to generalize TOR from binary to multi-way disjunctions, effect handlers support them with little effort.

Bringing specialist knowledge from one field into another is not an easy task: experts in both domains are few and far between, and so the details of the solution can be bewildering to those that have not yet bridged the divide. However, translating the results across the cognitive gap would have lost an important lesson: it is only when both fields come together that such results are made possible. Nevertheless, we have tried to make our material accessible to readers with a rudimentary understanding of both fields.

2. The Challenge: Modular Search Heuristics

This section explains the need to direct Prolog's control with search heuristics, and formulates the challenge of doing so in a modular fashion.

2.1 Depth-First Search

A Prolog program describes the logic of a problem in terms of the non-deterministic choice operator `(;)/2`. In this way the program implicitly describes a tree of possible paths to a solution, many of which may lead to dead ends denoted by `fail/0`. This tree is called the *search tree* and Prolog implements a *search strategy* that traverses the tree to find the solutions.

In principle Prolog systems could use any search strategy to explore the search tree. In this context, depth-first search (DFS) and breadth-first search (BFS) can be seen as two extremes: DFS can efficiently plumb the depths of complex structure, but might get stuck in a part of the search space that does not lead to a solution; BFS traverses branches evenly, but can be swamped by the complexity of exponentially many options.

In practice Prolog systems are highly tuned for DFS: they exploit special datastructures and algorithms for managing the interaction between backtracking and unification. This is particularly true of Prolog systems based on the Warren abstract machine (WAM) [41].

2.2 Search Heuristics

To compensate for the inflexibility of Prolog's fixed DFS strategy, various techniques have been proposed. This paper is concerned with techniques that prune the search tree into a form that is more suitable for solving with the fixed search strategy: these techniques remove parts of the search tree that are less likely to yield (interesting) solutions and where the search strategy would otherwise dwell for too long. We name these pruning techniques *search heuristics*: they do not fundamentally change the DFS order, but transform the search tree that is being traversed.

The standard practice for adding search heuristics to solve a problem is as follows. First the programmer writes the search problem code without optimisation, such as the `queens/2` program in Figure 1.

Next, if the performance of the search strategy is not adequate, the programmer selects a search heuristic to apply to the search problem. For instance, the programmer decides to apply the depth-bounded search strategy, which prunes parts of the search tree that are too deep. Of course there is no manifest tree datastructure to perform this pruning on; the search tree is only a conceptual representation of the search program's behaviour. Hence, applying the search heuristic really means modifying the program to embody a different tree.

Consequently, the programmer modifies the `queens/2` program to obtain `queens2/3` in Figure 1, where we have highlighted the required changes. This variant threads a bound on depth through the computation. The bound is decremented at every branching, and, when it hits zero, the branching is pruned.

2.3 The Challenge

Unfortunately, there are obvious problems with this approach to search heuristics. Since the heuristic's code is entangled with the problem's code, it is hard to modularly reuse either. Furthermore, this entanglement encourages an error-prone and labour-intensive copy-paste-modify approach. What we really want is a modular approach where problems (i.e., the logic) and heuristics (i.e., the control) can be defined separately and combined effortlessly.

The TOR approach [31, 34] constitutes the current state of the art in solving this problem. With TOR the depth-bounded search heuristic is expressed as an independent predicate `dbs/2` and applied to `queens/2` as follows.

```
?- search(dbs(10,queens(8,Qs))).
```

The approach is based on using a hookable disjunction `tor/2` to be used by `queens/2` instead of Prolog's regular disjunction `(;)/2`. The `dbs/2` heuristic influences `queen/2`'s search by installing appropriate call-backs in the hooks.

However, the hook-based approach lacks proper semantic grounding. As a consequence, the approach lacks elegant algebraic properties that make its use predictable, in other words, its use requires intimate knowledge of the implementation. Moreover, the solution's expressiveness is restricted. For instance, the common technique of visiting the branches of a disjunction in random order cannot be expressed. Finally, TOR cannot be generalized to multi-way disjunctions.

In this work we aim for a more general and elegant solution that is based on proper semantic foundations. For this purpose we start from a functional model of the problem in Haskell and apply established techniques to solve the problem, then subsequently derive a practical implementation for Prolog.

3. From Prolog to Haskell

In this section we present the functional model of Prolog that will drive our developments. This already presents a first dilemma: modern Prolog systems provide a large set of primitive operations, the so-called *built-ins*. Incorporating all of these in our model would be both extremely tedious and onerous. However, we also do not want to oversimplify our model and run the risk of obtaining results that do not work in actual Prolog systems.

We solve this dilemma with a standard functional programming technique, *abstract types*. Instead of specifying a concrete representation for Prolog computations, we assume the existence of an abstract type:

data *Prolog a*

Values of this type represent Prolog computations, also known as *goals*. As a small admission to functional programming, we will assume that goals have a return value which is reflected in the type variable a. Goals that return values of a particular type instantiate a accordingly. Proper Prolog goals are of type *Prolog* $()$: they do not return any value of interest, which is modelled by instantiating a with the unit type $()$.

```
queens(N, Qs) :-
  findall(C,between(1,N,C),L),
  go(L,N,[],Qs).

go([],N,Qs,Qs).
go([X|Xs],N,Acc,Qs) :-
  select(Y,[X|Xs],Ys),
  noThreat(Acc,N,1),
  go(Ys,N,[Y|Acc],Qs).

select(Y,[X|Xs],Ys) :-
  (  Y = X,
     Ys = Xs
  ;  Ys = [X|Zs],
     select(Y,Xs,Zs)
  ).
```

```
queens2(N, Qs, DB) :-
  findall(C,between(1,N,C),L),
  go(L,N,[],Qs, DB).

go([],N,Qs,Qs, _).
go([X|Xs],N,Acc,Qs, DB) :-
  select(Y,[X|Xs],Ys, DB, NDB),
  noThreat(Acc,N,1),
  go(Ys,N,[Y|Acc],Qs, NDB).

select(Y,[X|Xs],Ys, DB, NDB) :-
  DB > 0,
  (  Y = X,
     Ys = Xs,
     NDB is DB - 1
  ;  Ys = [X|Zs],
     DB1 is DB - 1,
     select(Y,Xs,Zs, DB1, NDB)
  ).
```

```
noThreat([],_,_).
noThreat([M|Ms],R,C) :- abs(M-R) =/= C, NC is C + 1, noThreat(Ms,R,NC).
```

Figure 1. The n-queens search problem: plain (left) and with depth bound (right, with changes highlighted in black).

We will assume that there are numerous ways to construct values of type *Prolog a*. However, we deliberately do not attempt to model them all and make no assumptions about how a Prolog goal is constructed. For our own purposes, we restrict our vocabulary to no more than four constructors (two primitives and two combinators) to build new goals.

The first two of these can be summarized by saying that *Prolog* has an instance of the monad class:

class *Monad m* **where**
$return :: a \to m\ a$
$(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$
instance *Monad Prolog*

This means that *Prolog* supports a sequential composition operator $(\gg\!=)$ and matching unit operator *return*. This notion of sequentiality closely corresponds to Prolog's conjunction operation `p , q`, which is satisfied when a solution exists for both `p` and `q`. In this setting order matters, and expressions are executed from left to right. More specifically `p , q` corresponds to $p \gg q$, which is an instance of $\gg\!=$ where the return value of p is of no interest to q.

$$p \gg q = p \gg\!= (\lambda x \to q)$$

A computation can end in one of two ways: either the search for a solution ends in success where the result is `true`, or in failure in which case the result is `false`. The behaviour of `true` interacts with the conjunction operator in the same way that *return* $()$ behaves with $(\gg)$, as described by the left and right unit laws of a monad:

$$return\ () \gg p = p = p \gg return\ () \tag{1}$$

Given this relationship, we identify `true` with *return* $()$.

To model `false`, we introduce the operation

$$fail :: Prolog\ a$$

This operation comes equipped with the *left-zero* law, which dictates how *fail* interacts with the monadic bind:

$$fail \gg\!= q = fail \tag{2}$$

This is a perfect fit for `false` since in the setting of Prolog there is no right-zero of conjunction: side-effects performed before failure cannot be undone in general.

Haskell	Prolog
return $()$	`true`
fail	`false`
$p \gg q$	`p , q`
$p \mid\mid\mid q$	`p ; q`

Table 1. Prolog model in Haskell

To model the disjunction `p ; q`, where either `p` or `q` must be satisfied, we introduce the operation:

$$(\mid\mid\mid) :: Prolog\ a \to Prolog\ a \to Prolog\ a$$

that satisfies the *left-distributivity* property:

$$(m \mid\mid\mid n) \gg\!= f = (m \gg\!= f) \mid\mid\mid (n \gg\!= f) \tag{3}$$

Moreover, $\langle Prolog\ a, (\mid\mid\mid), fail \rangle$ forms a monoid, where the following laws must hold:

$$x \mid\mid\mid (y \mid\mid\mid z) = (x \mid\mid\mid y) \mid\mid\mid z \tag{4a}$$

$$fail \mid\mid\mid x = x = x \mid\mid\mid fail \tag{4b}$$

The relationship between our model and the syntax of logic programming we are interested in is summarized in Table 1.

4. Background: Handlers and Transformers

Search heuristics are naturally expressed as transformations of search trees. For instance, the depth-bounded search prunes all subtrees below a given depth. Unfortunately, this view does not fit well with our monadic model of Prolog as the *Prolog* monad is opaque and we cannot observe the search tree structure of goals. A more effective technique for observing the syntactic structure of monadic programs is the *effect handlers* approach. This approach will be key to expressing heuristics in a modular way.

Effect handlers decouple the syntax and the semantics of side-effect primitives, which we call *operations* in the rest of the paper. The syntactic operations themselves live in an abstract syntax tree, which is modelled by the free monad. The semantics are captured in so-called handler functions, or handlers for short, and we focus on those that can be expressed as folds over the abstract syntax tree.

The decoupling has a number of advantages: it facilitates both the modular definition of monads in terms of separately defined operations, and also the assignment of different semantics to the same syntax. In this paper we will add one more advantage to that list: it allows us to extend opaque monads that have not necessarily been defined in terms of the effect handlers approach with new capabilities.

The well-known free monad $f^\star$ denotes abstract syntax trees where the shape of the nodes is captured in the functor f.

$$\textbf{data}\ f^\star\ a = \mathit{Return}\ a \mid \mathit{Op}\ (f\ (f^\star\ a))$$

A new node of shape f and subtrees of type $f^\star\ a$ can be built with the *Op* constructor. The *Return* constructor marks a non-terminal, and $(\gg\!=)$ performs simultaneous substitution on all non-terminals in the tree.

$$\begin{aligned}&\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Monad}\ (f^\star)\ \textbf{where}\\ &\quad \mathit{return}\ a \qquad\quad = \mathit{Return}\ a\\ &\quad \mathit{Return}\ a \gg\!= f = f\ a\\ &\quad \mathit{Op}\ n \qquad \gg\!= f = \mathit{Op}\ (\mathit{fmap}\ (\gg\!= f)\ n)\end{aligned}$$

An important convention when using the free monad for modelling syntax trees is that each node represents an operation and its subtrees denote the possible continuations from that operation. In this way, the free monad's notion of substitution and the operational interpretation of sequential composition both coincide in $(\gg\!=)$.

As an example, let's consider an abstract syntax tree for expressions involving only the operations in *MonadState s m*, where the monad *m* uniquely determines the state *s*.

$$\begin{aligned}&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadState}\ s\ m \mid m \to s\ \textbf{where}\\ &\quad \mathit{get} :: \mathit{Monad}\ m \Rightarrow m\ s\\ &\quad \mathit{put} :: \mathit{Monad}\ m \Rightarrow s \to m\ ()\end{aligned}$$

This has two primitive operations, and the shape of the corresponding syntax is given by the functor *State s r*.

$$\begin{aligned}&\textbf{data}\ \mathit{State}\ s\ r\\ &\quad = \mathit{Get}\ (s \to r)\\ &\quad \mid \mathit{Put}\ s\ (() \to r)\\ &\textbf{instance}\ \mathit{Functor}\ (\mathit{State}\ s)\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ (\mathit{Get}\ k) \;= \mathit{Get}\ (f \cdot k)\\ &\quad \mathit{fmap}\ f\ (\mathit{Put}\ s\ k) = \mathit{Put}\ s\ (f \cdot k)\end{aligned}$$

Thus, $\mathit{Op}\ (\mathit{Get}\ k)$ is the syntactic representation of $\mathit{get} \gg\!= k$ and $\mathit{Op}\ (\mathit{Put}\ s\ k)$ for $\mathit{put}\ s \gg\!= k$. If we want to represent the operations by themselves without any relevant continuation, we just instantiate k to *return*.

$$\begin{aligned}&\textbf{instance}\ \mathit{MonadState}\ s\ ((\mathit{State}\ s)^\star)\ \textbf{where}\\ &\quad \mathit{get} \;= \mathit{Op}\ (\mathit{Get}\ \mathit{return})\\ &\quad \mathit{put}\ s = \mathit{Op}\ (\mathit{Put}\ s\ \mathit{return})\end{aligned}$$

These operations construct syntax trees that represent actions.

4.1 The Free Monad Transformer

While the free monad approach forms an attractive basis for solving our search heuristics problem, it is not very appealing to replace the opaque *Prolog* monad with the free monad. After all, that would be akin to throwing away our existing Prolog system and engineering a new one from the ground up. Instead of this prodigious effort we would much prefer a more lightweight solution.

This solution comes in the form of the free monad transformer $f^\star_m$, which combines an existing monad m with the free monad $f^\star$ by interleaving computations in m with syntactic operations from f.

$$\begin{aligned}&\textbf{newtype}\ f^\star_m\ a = \mathit{Free}_T\ \{\mathit{unFree}_T :: m\ (\mathit{Free}\ f\ a\ (f^\star_m\ a))\}\\ &\textbf{data}\ \mathit{Free}\ f\ a\ x = \mathit{ReturnF}\ a \mid \mathit{OpF}\ (f\ x)\\ &\textbf{instance}\ \mathit{Functor}\ f \Rightarrow \mathit{Functor}\ (\mathit{Free}\ f\ a)\ \textbf{where}\\ &\quad \mathit{fmap}\ \_\ (\mathit{ReturnF}\ a) = \mathit{ReturnF}\ a\\ &\quad \mathit{fmap}\ f\ (\mathit{OpF}\ x) \qquad = \mathit{OpF}\ (\mathit{fmap}\ f\ x)\end{aligned}$$

When the monad is *Id* there is no additional structure so:

$$f^\star\ a = f^\star_{Id}\ a$$

In other words, $f^\star\ a$ is the fixpoint of $\mathit{Free}\ f\ a$.

The structure of $f^\star_m$ requires m to be a monad, and f to be a functor. We use this fact so often that we will use the following convenient notation for the required type constraint synonym:

$$\textbf{type}\ \vdash f^\star_m = (\mathit{Functor}\ f, \mathit{Functor}\ m, \mathit{Monad}\ m)$$

In other words, this constraint expresses that $f^\star_m$ is a well-formed free monad transformer.

We can define the fold over this structure:

$$\begin{aligned}&\mathit{fold} :: \vdash f^\star_m \Rightarrow (m\ (\mathit{Free}\ f\ a\ b) \to b) \to (f^\star_m\ a \to b)\\ &\mathit{fold}\ \mathit{alg} = \mathit{alg} \cdot \mathit{fmap}\ (\mathit{fmap}\ (\mathit{fold}\ \mathit{alg})) \cdot \mathit{unFree}_T\end{aligned}$$

This can be used to give a definition of $(\gg\!=)$ in the instance that shows that $f^\star_m$ is a monad.

$$\begin{aligned}&\textbf{instance}\ \vdash f^\star_m \Rightarrow \mathit{Monad}\ (f^\star_m)\ \textbf{where}\\ &\quad \mathit{return} \;= \mathit{Free}_T \cdot \mathit{return} \cdot \mathit{ReturnF}\\ &\quad m \gg\!= f = \mathit{fold}\ (\mathit{Free}_T \cdot \mathit{join} \cdot \mathit{fmap}\ (\mathit{unFree}_T \cdot \mathit{alg}))\ m\\ &\qquad \textbf{where}\ \mathit{alg}\ (\mathit{ReturnF}\ a) = f\ a\\ &\qquad\qquad \mathit{alg}\ (\mathit{OpF}\ \mathit{op}) = \mathit{opF}\ \mathit{op}\\ &\mathit{opF}\ \mathit{op} = \mathit{Free}_T\ (\mathit{return}\ (\mathit{OpF}\ \mathit{op}))\end{aligned}$$

The following definition of *runStateF* is an example of a handler that turns eliminates the *State* syntax by interpreting it in terms of a state that is threaded through the computation.

$$\begin{aligned}&\mathit{runStateF} :: \vdash (\mathit{State}\ s)^\star_m \Rightarrow (\mathit{State}\ s)^\star_m\ a \to s \to m\ (a,s)\\ &\mathit{runStateF}\ p\ s0 = \mathit{runFreeT}\ (\mathit{alg}\ s0)\ p\ \textbf{where}\\ &\quad \mathit{alg}\ s\ (\mathit{ReturnF}\ x) \qquad = \mathit{return}\ (x,s)\\ &\quad \mathit{alg}\ s\ (\mathit{OpF}\ (\mathit{Get}\ k)) \quad = \mathit{runStateF}\ (k\ s)\ \ s\\ &\quad \mathit{alg}\ s\ (\mathit{OpF}\ (\mathit{Put}\ s'\ k)) = \mathit{runStateF}\ (k\ ())\ s'\end{aligned}$$

The above definition makes use of the following auxiliary definition:

$$\begin{aligned}&\mathit{runFreeT} :: \vdash f^\star_m \Rightarrow (\mathit{Free}\ f\ a\ (f^\star_m\ a) \to m\ b) \to (f^\star_m\ a \to m\ b)\\ &\mathit{runFreeT}\ \mathit{alg}\ p = \mathit{unFree}_T\ p \gg\!= \mathit{alg}\end{aligned}$$

which runs the given computation upto the first syntactic operation, which it delegates to *alg*. In the case of *runStateF*, this *alg* handles the state operation appropriatley, and at the end of the computation returns the result together with the final state.

In the remainder of the paper we will also make use of the following variant of *runFreeT*:

$$\begin{aligned}&\mathit{step} :: \vdash f^\star_m \Rightarrow f^\star_m\ a \to (f\ (f^\star_m\ a) \to f^\star_m\ a) \to f^\star_m\ a\\ &\mathit{step}\ t\ \mathit{alg} = \mathit{Free}_T\ (\mathit{runFreeT}\ \mathit{alg}'\ t)\ \textbf{where}\\ &\quad \mathit{alg}'\ (\mathit{ReturnF}\ x) = \mathit{return}\ (\mathit{ReturnF}\ x)\\ &\quad \mathit{alg}'\ (\mathit{OpF}\ \mathit{op}) \quad = \mathit{unFree}_T\ (\mathit{alg}\ \mathit{op})\end{aligned}$$

The function *step* differs in two ways from *runFreeT*. Firstly, it does not promise to eliminate the syntactic operations altogether. Instead, it can be used to eliminate only some operations or to replace them by others. Secondly, *step* preserves *ReturnF*. As a consequence, the *alg* parameter only needs to concern itself with the operations.

5. Heuristics as Handlers in Haskell

The machinery of effect handlers gives us the tools we need to describe heuristics in a modular way. Our solution will be developed in four steps.

5.1 Step 1: Overloading

Our first step is to overload the operations of *Prolog*. Here we accomplish this with the *MonadProlog* type class:

$$\begin{aligned}&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadProlog}\ m\ \textbf{where}\\&\quad \mathit{fail} :: m\ a\\&\quad (|||) :: m\ a \to m\ a \to m\ a\end{aligned}$$

which inherits the left-zero and left-distributivity laws of *Prolog*,

$$\mathit{fail} \gg\!= q = \mathit{fail} \tag{5a}$$

$$(m \,|||\, n) \gg\!= f = (m \gg\!= f) \,|||\, (n \gg\!= f) \tag{5b}$$

However, importantly, we do not require that $\langle m\ a, (|||), \mathit{fail}\rangle$ forms a monoid. This relaxation is crucial to support search heuristics. After all, the monoid laws require that the shape of the search tree is irrelevant. For instance, according to the associativity law, the following two trees should be indistinguishable:

$$\begin{aligned}t_1 &= \mathit{return}\ x \,|||\, (\mathit{return}\ y \,|||\, \mathit{return}\ z)\\t_2 &= (\mathit{return}\ x \,|||\, \mathit{return}\ y) \,|||\, \mathit{return}\ z\end{aligned}$$

In contrast, the shape of the search tree is essential for search heuristics. Different shapes of trees will be affected differently by the same heuristic. For instance, the depth-bounded search heuristic may prune the solutions y and z in t_1, while it prunes x and y in t_2.

5.2 Step 2: Introducing Syntax

We proceed in the second step by capturing the relevant operations as syntax using the free monad transformer. While *MonadProlog* provides two operations, *fail* and $(|||)$, we will see in the next step that we can get away with capturing only $(|||)$ in syntax. This comes at the cost of somewhat more complicated handlers. However, Section 6 will bear out that keeping the syntactic footprint as small as possible is a good idea.

Hence, the functor *Or* captures only $p \,|||\, q$ with the syntax $\mathit{Or}\ p\ q$.

$$\begin{aligned}&\textbf{data}\ \mathit{Or}\ x = \mathit{Or}\ x\ x\\&\mathit{orF} :: \mathit{Monad}\ m \Rightarrow \mathit{Or}^\star_m\ a \to \mathit{Or}^\star_m\ a \to \mathit{Or}^\star_m\ a\\&\mathit{orF}\ p\ q = \mathit{opF}\ (\mathit{Or}\ p\ q)\\&\textbf{instance}\ \mathit{Functor}\ \mathit{Or}\ \textbf{where}\\&\quad \mathit{fmap}\ f\ (\mathit{Or}\ p\ q) = \mathit{Or}\ (f\ p)\ (f\ q)\end{aligned}$$

Observe that unlike *Get* the constructor *Or* does not require a separate field for the continuation. Thanks to the left-distributivity property, we can express $(p \,|||\, q) \gg\!= k$ also as $(p \gg\!= k) \,|||\, (q \gg\!= k)$.

This data acts as a syntactic construction that gives us an instance of *MonadProlog* in terms of the free monad transformer:

$$\begin{aligned}&\textbf{instance}\ \mathit{MonadProlog}\ m \Rightarrow \mathit{MonadProlog}\ (\mathit{Or}^\star_m)\ \textbf{where}\\&\quad \mathit{fail} \quad = \mathit{lift}\ \mathit{fail}\\&\quad p \,|||\, q = \mathit{orF}\ p\ q\end{aligned}$$

5.3 Step 3: Adding Heuristics

The syntactic *Or* gives substance to the search tree that is implicitly embodied by a computation. Now we can truly write search heuristics as functions that transform this search tree.

$$\textbf{type}\ \mathit{Heuristic}\ m\ a = \mathit{Or}^\star_m\ a \to \mathit{Or}^\star_m\ a$$

Below we capture a number of well-known heuristics in this form.

Depth-Bounded Search The depth-bounded search heuristic bounds the search tree to a given depth, pruning everything underneath.

$$\begin{aligned}&\mathit{dbs} :: \mathit{MonadProlog}\ m \Rightarrow \mathit{Int} \to \mathit{Heuristic}\ m\ a\\&\mathit{dbs}\ 0\ t = \mathit{fail}\\&\mathit{dbs}\ n\ t = \mathit{step}\ t\ \mathit{alg}\ \textbf{where}\\&\quad \mathit{alg}\ (\mathit{Or}\ x\ y) = (\mathit{dbs}\ (n-1)\ x) \,|||\, (\mathit{dbs}\ (n-1)\ y)\end{aligned}$$

Here we see our first use of the *step* function. It is used to decrement the depth bound at every Or. When the limit is exceeded, the whole remaining computation is replaced by failure.

Discrepancy-Bounded Search Discrepancy-bounded search is a minor variant of depth-bounded search that bounds the number of right turns.

$$\begin{aligned}&\mathit{dibs} :: \mathit{MonadProlog}\ m \Rightarrow \mathit{Int} \to \mathit{Heuristic}\ m\ a\\&\mathit{dibs}\ 0\ t = \mathit{fail}\\&\mathit{dibs}\ n\ t = \mathit{step}\ t\ \mathit{alg}\ \textbf{where}\\&\quad \mathit{alg}\ (\mathit{Or}\ x\ y) = (\mathit{dibs}\ n\ x) \,|||\, (\mathit{dibs}\ (n-1)\ y)\end{aligned}$$

Node-Bounded Search The node-bounded search heuristic limits the number of nodes that are visited during the search, pruning any further nodes that come up. Like many other search heuristics, it maintains some state across the different branches of the search tree: the maximum number of nodes that it can still visit before starting to prune.

The implementation of node-bounded search in Prolog requires the use of mutable references.[1] We can capture such references in our model by using the type class *MonadRef*. This supports three operations: *newRef* creates a new reference $r\ a$ within the monadic context m, *readRef* extracts a value from a reference into the context, and *writeRef* takes a reference and a value a, and writes the value into the reference.

$$\begin{aligned}&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadRef}\ r\ m \mid m \to r\ \textbf{where}\\&\quad \mathit{newRef}\ \ :: a \to m\ (r\ a)\\&\quad \mathit{readRef}\ \ :: r\ a \to m\ a\\&\quad \mathit{writeRef} :: r\ a \to a \to m\ ()\end{aligned}$$

To illustrate the use of this interface, we implement *modifyRef*, which simply modifies the current value in some cell by applying a function f to its contents, and then returns the original value.

$$\begin{aligned}&\mathit{modifyRef} :: \mathit{MonadRef}\ r\ m \Rightarrow r\ a \to (a \to a) \to m\ a\\&\mathit{modifyRef}\ r\ f = \textbf{do}\ x \leftarrow \mathit{readRef}\ r\\&\qquad\qquad\qquad\quad \mathit{writeRef}\ r\ (f\ x)\\&\qquad\qquad\qquad\quad \mathit{return}\ x\end{aligned}$$

This works by first reading the value x contained in the reference, writing the new value $f\ x$ to the reference, and then returning x.

In addition to the usual properties of mutable references, we also explicitly stipulate the interaction with backtracking: the writes are not backtracked over.

$$\mathit{writeRef}\ \mathit{ref}\ x \gg (p \,|||\, q) = (\mathit{writeRef}\ \mathit{ref}\ x \gg p) \,|||\, q \tag{6}$$

Read from left to right this law expresses that writes in the left branch are also seen by the right branch. Contrast this with the characterization of backtracking behaviour:

$$\begin{aligned}&\mathit{writeRef}\ \mathit{ref}\ x \gg (p \,|||\, q)\\&\qquad =\\&(\mathit{writeRef}\ \mathit{ref}\ x \gg p) \,|||\, (\mathit{writeRef}\ \mathit{ref}\ x \gg q)\end{aligned}$$

which expresses that writes in one branch are not seen by the other branch.

The support for references can be lifted straightforwardly from m to $\mathit{Or}^\star_m$.

$$\begin{aligned}&\textbf{instance}\ \mathit{MonadRef}\ r\ m \Rightarrow \mathit{MonadRef}\ r\ (\mathit{Or}^\star_m)\ \textbf{where}\\&\quad \mathit{newRef}\ x \quad = \mathit{lift}\ (\mathit{newRef}\ x)\\&\quad \mathit{readRef}\ r \quad = \mathit{lift}\ (\mathit{readRef}\ r)\\&\quad \mathit{writeRef}\ r\ x = \mathit{lift}\ (\mathit{writeRef}\ r\ x)\end{aligned}$$

[1] See Schrijvers et al. [34, Appendix A] for a Prolog implementation of mutable references.

This support enables us to express a handler for node-bounded search.

```
nbs :: (MonadProlog m, MonadRef r m) ⇒ Int → Heuristic m a
nbs n t = newRef n >>= go t where
  go t' ref = step t' alg where
    alg (Or x y) = do n ← modifyRef ref pred
                      guard (n > 0)
                      go x ref ||| go y ref
```

Failure-Bounded Search Failure-bounded search terminates the search when too many paths in the tree lead to dead ends. It may actually seem surprising that we can write this heuristic without being able to explicitly observe failure. Nevertheless, with a clever trick that relies on the underlying DFS we can observe failure indirectly.

```
fbs :: (MonadProlog m, MonadRef r m) ⇒
       Int → Heuristic m a
fbs n t = do ref ← newRef n
             fref ← newRef False                    -- (a)
             x ← go ref fref t
             writeRef fref True                     -- (b)
             return x where
  go ref' fref' t' = step t' alg
    where alg (Or x y) = x |||
            (do b ← modifyRef fref' (const False)   -- (c)
                when (¬ b)                          -- (d)
                  (do n ← modifyRef ref' pred
                      guard (n > 0))
                y)
```

The mutable reference *fref* expresses whether the last explored path has terminated successfully. At the start of the search (a), we are on the first path but have not completed it yet. Hence initially *fref* holds the value *False*. Later, when a solution is found (b), the value *True* is written into the reference. After completing a path successfully or unsuccessfully, the search backtracks into the right branch of an *or*. At the start of the right branch (c) we can observe in *fref* whether the previous path was successful or not. We also write *False* into *fref* to capture the status of the new path we are on. If the previous path has failed (d), we subtract one from the failure bound and prune if we have failed too often already.

5.4 Step 3′: Adding Heuristics as Trees

The effect handlers approach distinguishes syntactic operations and handlers. Syntactic operations offer the flexibility of a deeply embedded domain-specific language (DSL); they can be freely analyzed, manipulated and interpreted in different ways. This is exactly the property we have put to good use with the definition of heuristics over search trees. In contrast, handlers like our heuristics are akin to a shallow embedding of a DSL: they can be used in one way only, by function application to a computation.

In this section we show how to recover much of the flexibility of deep embeddings, while simultaneously providing a more structured approach to defining search heuristics. This approach is based on two ingredients:

1. We capture the essence of a heuristic in an archetypal search tree. For instance, the archetypal search tree for depth-bounded search is a perfect binary tree of depth n with failures at its leaves.

   ```
   dbsTree 0 = fail
   dbsTree n = dbsTree (n − 1) ||| dbsTree (n − 1)
   ```

2. We apply the heuristic to a search problem by means of an operator $(\curlyvee)$ called *entwine*, that combines two search trees: in this case, one given by the logic to solve the problem, and the other given by the heuristic.

   For instance, we recover *dbs* as follows:

   ```
   dbs' :: MonadProlog m ⇒ Int → Heuristic m a
   dbs' db t = dbsTree db ⋎ t
   ```

In summary, this approach refines Kowalski's slogan to:

$$algorithm = control \curlyvee logic$$

where both logic and control are expressed in a declarative rather than an operational style. Moreover, they are expressed in the same language of search trees.

The Entwining Operator The $(\curlyvee)$ operator is similar to the definition of the *zip* operation: zipping two lists truncates the longer one when their structure disagrees. Similarly, entwining two trees truncates the larger one when their structure disagrees. The truncation of trees is essentially the pruning of the search space.

```
(⋎) :: (MonadProlog m) ⇒ Or*_m a → Or*_m a → Or*_m a
p ⋎ q = p `step` (λ(Or x1 x2) →
          q `step` (λ(Or y1 y2) →
            (x1 ⋎ y1) ||| (x2 ⋎ y2)))
```

This operation steps into the first tree, and inspects its structure for an *Or* constructor. When this is found, it steps into the second tree where it again inspects its structure for an *Or* constructor. If both are found, then a new tree is constructed, where children of the trees are entwined together.

Note that $\langle Or^{\star}_m\ a, (\curlyvee), inf \rangle$ forms a monoid, because $(\curlyvee)$ is associative and the infinitely branching tree *inf* is its identity:

```
inf :: Monad m ⇒ Or*_m a
inf = opF (Or inf inf)
```

Archetypal Trees and Handlers We can establish that *dbsTree n* captures the essence of the *dbs* handler in an archetypal tree by showing that *dbs n* and *dbs′ n* are equivalent. Our proof proceeds by induction on n.

For $n = 0$ we have that:

```
    dbs 0 t
=   { def. of dbs }
    fail
=   { ∀alg.step fail alg = fail }
    fail `step` (λ(Or x1 x2) →
      t `step` (λ(Or y1 y2) → (x1 ⋎ y1) ||| (x2 ⋎ y2)))
=   { def. of (⋎) }
    fail ⋎ t
=   { def. of dbsTree }
    dsbTree 0 ⋎ t
=   { def. of dbs' }
    dbs' 0 t
```

```
dibs' :: (MonadProlog m) ⇒ Int → Heuristic m a
dibs' db t = dibsTree db ⅄ t where
  dibsTree 0 = fail
  dibsTree n = dibsTree n ⅄ dibsTree (n − 1)

nbs' :: (MonadProlog m, MonadRef r m) ⇒
        Int → Heuristic m a
nbs' n t = do ref ← newRef n
              t ⅄ nbsTree ref where
  nbsTree ref = do n ← modifyRef ref pred
                   guard (n > 0)
                   nbsTree ref ||| nbsTree ref
```

```
fbs' :: (MonadProlog m, MonadRef r m) ⇒ Int → Heuristic m a
fbs' n t = do ref ← newRef n
              fref ← newRef False
              x ← t ⅄ fbsTree ref fref
              writeRef fref True
              return x where
  fbsTree ref fref = fbsTree ref fref |||
    do b ← modifyRef fref (const False)
       when (¬ b) (do n ← modifyRef ref pred
                      guard (n > 0))
       fbsTree ref fref
```

Figure 2. Search heuristics expressed as entwined trees

Also, for $n = m+1$ and induction hypothesis $dbs\ m = dbs'\ m$, we can show that:

```
    dbs (m+1) t
=   { def. of dbs }
    step t (λ(Or y1 y2) → (dbs m y1) ||| (dbs m y2))
=   { induction hypothesis }
    step t (λ(Or y1 y2) → (dbs' m y1) ||| (dbs' m y2))
=   { def. of dbs' }
    step t (λ(Or y1 y2) → (dbsTree m ⅄ y1) ||| (dbsTree m ⅄ y2))
=   { ∀alg t1 t2.alg (Or t1 t2) = step (t1 ||| t2) alg }
    (dbsTree m ||| dbsTree m) `step` (λ(Or x1 x2) →
     t `step` (λ(Or y1 y2) → (x1 ⅄ y1) ||| (x2 ⅄ y2)))
=   { def. of ⅄ }
    (dbsTree m ||| dbsTree m) ⅄ t
=   { def. of dbsTree }
    dbsTree (m+1) ⅄ t
=   { def. of dbs' }
    dbs' (m+1) t
```

Modular Definition of Heuristics Figure 2 shows that there is an archetypal search tree hidden in all the heuristics of Section 5.4. Yet, the main advantage of archetypal search trees is that we can define them in a convenient modular style.

For instance, we can define *dbsTree n* as *delay n* ≫ *fail*, where *delay* returns *return* () after a given depth:

```
delay :: MonadProlog m ⇒ Int → Or*_m ()
delay 0 = return ()
delay n = delay (n − 1) ||| delay (n − 1)
```

Here, *return* () is a placeholder for another heuristic that can be plugged in with (≫) and becomes active at depth n in the search tree.

This allows us to define iterative deepening as follows:

```
itd :: (MonadProlog m, MonadRef r m) ⇒ Heuristic m a
itd t = do  newRef False >>= go t 0 where
  go t n ref = (t ⅄ (delay n ≫ prune ref)) |||
               do b ← readRef ref
                  if b then do writeRef ref False
                               go t (n+1) ref
                       else fail
  prune ref = writeRef ref True ≫ fail
```

Iterative deepening *itd* repeatedly runs a depth-bounded search, incrementing the depth bound on each iteration. The iterative process stops when no pruning happened in the last iteration. The heuristic *prune ref* performs immediate pruning and records this in the mutable reference to remember it across backtracking. With *delay n* ≫ *prune ref* the immediate pruning is delayed to depth n.

While TOR [34] provides an operator `merge/2` similar to (⅄), that operator does not satisfy the same elegant algebraic properties (e.g, forming a monoid) and, as consequence, cannot express *delay*ed heuristics in this modular fashion.

Limitation While (⅄) is very convenient and captures a large class of search heuristics, not all heuristics can be expressed in this way. In particular, consider the random reordering of branches,

```
muddle :: (MonadRandom m, MonadProlog m) ⇒ Heuristic m a
muddle t = step t alg where
  alg (Or x y) = do b ← getRandom
                    if b then muddle x ||| muddle y
                         else muddle y ||| muddle x
```

which is a popular heuristic to randomize the search tree. This heuristic cannot be expressed with (⅄) because it always keeps left branches on the left and right branches on the right.

5.5 Step 4: Reflecting Syntax Back into Semantics

Finally, the *semOr* handler reflects the syntactic *Or* back into the semantic (|||) of the underlying monad m.

```
semOr :: MonadProlog m ⇒ Or*_m a → m a
semOr = runFreeT alg where
  alg (ReturnF a)     = return a
  alg (OpF (Or x y)) = semOr x ||| semOr y
```

We can now recover *queens'* as:

```
queens'' n db = semOr (dbs db (queens n))
```

provided that *queens* is written against the type class *MonadProlog* rather than the opaque monad *Prolog*.

In this scheme, we start with the original tree generated by *queens*, but interpreted under the $Or^{\star}_{m}$ monad. The ensuing tree is then pruned by the function *dbs* before it is finally reflected back into the underlying monad m.

6. From Haskell to Prolog

This section sets up the means to transfer our Haskell-based solution for modular search heuristics to Prolog.

On the outset, there are several compelling reasons why basing our approach on the free monad transformer would make it well-suited for implementation in Prolog:

1. In the $Or^{\star}_{m}$ type, we can choose m to be the complex (but implicit) monad that underpins Prolog.
2. The approach allows us to conveniently reuse Prolog's primitive implementations for *return* $()$ and *fail* by lifting.
3. We can lift individual feature extensions that we have modelled as additional class constraints, such as mutable references and random number generation. This could be applied to other extensions we have not covered explicitly in our model: predicates (i.e., goal abstractions), mutable databases, I/O, and many more.

However, implementing the free monad transformer itself in Prolog is challenging. As the *Prolog* monad is implicit in the Prolog language, it does not lend itself to transformation. So we now direct our efforts to overcoming this obstacle.

6.1 Meta-Interpreter

Meta-circular interpreters, or just meta-interpreters, are the most common way of modelling Prolog language extensions in Prolog. The signature of a plain Prolog meta-interpreter is `eval/1`, where `eval(Goal)` conceptually denotes the type $m\ ()$. However, our meta-interpreter needs to capture computations of type $m\ (\mathit{Free}\ \mathit{Or}\ ()\ (\mathit{Or}^{\star}_{m}\ ()))$. Hence, we extend the interpreter's signature with an extra (output) argument: `eval(Goal,Flag)`. `Flag` is either `return` (corresponding to *Return* $()$) or `or(Goal1,Goal2)` (corresponding to $\mathit{opF}\ (\mathit{Or}\ g_1\ g_2)$).

With this signature it is straightforward to port the free monad transformer implementation to Prolog (see Figure 3). However, this meta-interpreter requires us to reify *all* of the syntax in Prolog which we are interested in. For our small fragment this is very manageable, but there are many other features in Prolog, such as built-ins and user-defined predicate calls, and with a growing list, this approach will soon become tedious and unmaintainable. We clearly need an approach that is orthogonal to the existing language features.

```
semOr (Goal) :-
  eval(Goal,Flag),
  ( Flag = return ->
      true
  ; Flag = or(G1,G2) ->
      ( runT(G1)
      ; runT(G2)
      )
  ).

eval(Goal,Flag) :-
  ( Goal = fail ->
      fail
  ; Goal = true ->
      Flag = return
  ; Goal = (Goal1,Goal2) ->
      eval(Goal1,Flag1),
      ( Flag1 = return ->
          eval(Goal2,Flag)
      ; Flag1 = or(Left,Right) ->
          Flag = or((Left,Goal2),(Right,Goal2))
      )
  ; Goal = (Goal1;Goal2) ->
      Flag = or(Goal1,Goal2)
  ; Goal = entwine(Goal1,Goal2) ->
      eval(Goal1,Flag1),
      ( Flag1 = return ->
          Flag = return
      ; Flag1 = or(Left1,Right1) ->
          eval(Goal2,Flag2),
          ( Flag2 = return ->
              Flag = return
          ; Flag2 = or(Left2,Right2) ->
              eval((entwine(Left1,Left2)
                   ;entwine(Right1,Right2)
                   ),Flag)
          )
      )
  ).
```

Figure 3. Prolog meta-interpreter

6.2 Delimited Continuations

We do not have to look very far for an alternative approach. *Delimited continuations* provide an isomorphic replacement of the free monad transformer. An approach based on delimited continuations is more Prolog-friendly because recent work [32] has made them available natively in the hProlog [7] system.

Prolog provides an idiosyncratic interface of two operators for capturing delimited continuations: `shift/1` and `reset/3` that generalize exceptions, which are conventionally modelled by:[2]

$$\begin{aligned}&\textbf{class}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadDelCont}\ f\ m \mid m \to f\ \textbf{where}\\ &\quad \mathit{shift}_P :: f\ b \to m\ b\\ &\quad \mathit{reset}_P :: m\ a \to (a \to m\ b) \to\\ &\qquad\qquad (\mathit{Susp}\ f\ (m\ a) \to m\ b) \to m\ b\end{aligned}$$

These operations can be seen as a generalization of *throw* and *catch* from the well-known error monad. Like *throw*, the shift_P operation terminates the ongoing computation abruptly, with a value that indicates the reason. Like *catch*, the reset_P operation makes it possible to observe whether a subcomputation terminates normally or abruptly.

The big difference between both interfaces is that *catch* only exposes the reason for the abrupt termination. In contrast, reset_P gives us, nicely packaged up in a *Susp*(ension), both the reason (as a value of type $f\ a$), and the unfinished part (the continuation given by $a \to r$) of the subcomputation.

$$\begin{aligned}&\textbf{data}\ \mathit{Susp}\ f\ r\ \textbf{where}\\ &\quad S :: f\ a \to (a \to r) \to \mathit{Susp}\ f\ r\\ &\textbf{instance}\ \mathit{Functor}\ (\mathit{Susp}\ f)\ \textbf{where}\\ &\quad \mathit{fmap}\ f\ (S\ d\ r) = S\ d\ (f \cdot r)\end{aligned}$$

Note that the type of the reason $f\ a$ is indexed by the type a expected by the continuation $a \to r$.

The reset_P and shift_P control operations satisfy a number of laws that regulate their interaction.

$$\mathit{reset}_P\ (\mathit{shift}_P\ d \gg\!= f)\ r\ h = h\ (S\ d\ f) \tag{7a}$$

$$\mathit{reset}_P\ (\mathit{return}\ x)\ r\ h = r\ x \tag{7b}$$

The first law shows that a shift under a reset is handled by h, which has access to the suspended computation. The second law shows that the result of a successful computation under a reset is handled by r, which has access to the result.

6.3 The Delimited Continuations Transformer

An instance of *MonadDelCont* can be obtained from any monad m by making use of the delimited continuations monad transformer, written $f^{\dagger}_{m}\ a$, and this will serve as our replacement for the free monad transformer that fits the functionality exposed by Prolog.

[2] Note that these control operators are have different signatures and semantics than those originally introduced by Danvy and Filinski under those names [6]. They are more closely related to Sitaram's *fcontrol* and *run* operators [36].

$$\textbf{newtype}\ f_m^\dagger\ a = DC_T\ \{\mathit{runDC_T} :: \forall r.(a \to m\ r) \to (\mathit{Susp}\ f\ (f_m^\dagger\ a) \to m\ r) \to m\ r\}$$

Its representation takes two continuations, the return continuation of type $a \to m\ r$, and the handler continuation of type $\mathit{Susp}\ f\ (f_m^\dagger\ a) \to m\ r$.

The transformed monad's *return* method invokes the return continuation, and its $(\gg\!=)$ extends both continuations:

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{Monad}\ (f_m^\dagger)\ \textbf{where}\\ \quad \mathit{return}\ x = DC_T\ (\lambda r\ h \to r\ x)\\ \quad m \gg\!= f = DC_T\ (\lambda r\ h \to\\ \quad\quad \mathit{runDC_T}\ m\ (\lambda x \to \mathit{runDC_T}\ (f\ x)\ r\ h)\ (h \cdot \mathit{fmap}\ (\gg\!= f)))\end{array}$$

The reset_P method sets the handler continuation and the shift_P method invokes it.

$$\begin{array}{l}\textbf{instance}\ \mathit{Monad}\ m \Rightarrow \mathit{MonadDelCont}\ f\ (f_m^\dagger)\ \textbf{where}\\ \quad \mathit{reset}_P\ m\ r\ h = DC_T\ (\lambda r'\ h' \to\\ \quad\quad \mathit{runDC_T}\ m\ (\lambda x \to \mathit{runDC_T}\ (r\ x)\ r'\ h')\\ \quad\quad\quad\quad (\lambda p \to \mathit{runDC_T}\ (h\ p)\ r'\ h'))\\ \quad \mathit{shift}_P\ d = DC_T\ (\lambda r\ h \to h\ (S\ d\ \mathit{return}))\end{array}$$

6.4 The Isomorphism

We will now establish the relationship between the free monad transformer and the delimited continuations monad transformer. This will enable us to adapt our existing infrastructure formulated in terms of the former to Prolog-compatible infrastructure in terms of the latter.

For all intents and purposes the two transformers are isomorphic, but there are two significant technical wrinkles that must be ironed out before formally establishing this isomorphism.

First, we must enforce that the base functor f of the delimited continuation transformer is only applied to monadic values. These values are after all meant to be the continuations of the syntactic operations. We can impose this restriction by pre-composing f with the monad and thus use $f_m^\ddagger$ instead of $f_m^\dagger$.

$$\textbf{data}\ f_m^\ddagger\ a = L\ \{\mathit{runL} :: (f \circ f_m^\ddagger)_m^\dagger\ a\}$$

where $(\circ)$ is the well-known functor composition:

$$\textbf{data}\ (f \circ g)\ a = \mathit{Comp}\ \{\mathit{runComp} :: f\ (g\ a)\}$$

Second, the suspension $S\ s\ k$ breaks up a continuation into two parts: one part that sits under the syntactic construction s, and another part k that represents the following execution. Consequently, continuations that have been broken up at different points are distinguishable. However, if we are careful to always treat these parts as one atomic continuation, then this does not pose a problem. We can enforce this atomic treatment by *normalize*-ing all $f_m^\ddagger\ a$ computations.

$$\begin{array}{l}\mathit{normalize} :: \vdash f_m^\star \Rightarrow f_m^\ddagger\ a \to f_m^\ddagger\ a\\ \mathit{normalize}\ m =\\ \quad L\ (DC_T\ (\lambda r\ h \to \mathit{runDC_T}\ (\mathit{runL}\ m)\ r\ (h \cdot \mathit{norm})))\\ \quad \textbf{where}\\ \quad\quad \mathit{norm}\ (S\ x\ k) =\\ \quad\quad\quad S\ ((\mathit{Comp} \cdot \mathit{fmap}\ \mathit{join} \cdot \mathit{runComp} \cdot \mathit{fmap}\ (L \cdot k))\ x)\ \mathit{return}\end{array}$$

Taking the above two points into consideration we can formulate the two witnesses of the isomorphism:

$$\begin{array}{l}\mathit{to} :: \vdash f_m^\star \Rightarrow f_m^\star\ a \to f_m^\ddagger\ a\\ \mathit{to}\ m = L\ (DC_T\ (\lambda r\ h \to\\ \quad \textbf{do}\ x \leftarrow \mathit{unFree_T}\ m\\ \quad\quad \textbf{case}\ x\ \textbf{of}\\ \quad\quad\quad \mathit{ReturnF}\ a \to r\ a\\ \quad\quad\quad \mathit{OpF}\ s\ \ \ \to h\ (S\ (\mathit{Comp}\ (\mathit{fmap}\ \mathit{to}\ s))\ \mathit{return})))\end{array}$$

$$\begin{array}{l}\mathit{from} :: \vdash f_m^\star \Rightarrow f_m^\ddagger\ a \to f_m^\star\ a\\ \mathit{from}\ m\ = \mathit{Free_T}\ (\mathit{runDC_T}\ (\mathit{runL}\ m)\ r\ h)\ \textbf{where}\\ \quad r = \ \mathit{return} \cdot \mathit{ReturnF}\\ \quad h = \ \mathit{return} \cdot \mathit{OpF} \cdot \mathit{fmap}\ \mathit{from} \cdot \mathit{collapse}\\ \quad \mathit{collapse}\ (S\ s\ k) = \mathit{fmap}\ (\gg\!= L \cdot k)\ (\mathit{runComp}\ s)\end{array}$$

These two functions indeed witness the isomorphism when quotiented by *normalize*:

$$\mathit{from} \cdot \mathit{to} = \mathit{id} \tag{8}$$

$$\mathit{normalize} \cdot \mathit{to} \cdot \mathit{from} = \mathit{normalize} \tag{9}$$

Finally, using the isomorphism it is possible to derive that the $f_m^\ddagger$ equivalent of *opF* can be defined simply as:

$$\mathit{opF}'\ s = \mathit{shift}_P\ s$$

In other words, a syntactic operation can be modelled directly in Prolog using `shift/1`.

Similarly, we can derive the counterpart of *step* as:

$$\mathit{step}'\ m\ h = \mathit{reset}_P\ m\ \mathit{return}\ h$$

With opF' and step' we have all we need to make the transition from Haskell to Prolog.

Haskell	Prolog
$\mathit{Left}\ ()$	`K = 0`
$\mathit{Right}\ (S\ d\ k)$	`K = [[k]], D = [[d]]`

Table 2. Interpretation of delimited continuations in Prolog

7. Heuristics as Handlers in Prolog

Finally, we have a Prolog-friendly approach that is both light-weight and enables a mostly native execution of search problems.

7.1 Delimited Continuations

The actual Prolog interface to delimited continuations is as follows: The built-in predicate `shift(D)` corresponds to $\mathit{shift}_P\ t$. The reset_P operation has signature `reset(P,K,D)`, which corresponds more or less to

$$\begin{array}{l}\mathit{reset}_P :: M\ () \to M\ (\mathit{Either}\ ()\ (\mathit{Susp}\ f\ (M\ ())))\\ \mathit{reset}_P\ p\ (\mathit{return} \cdot \mathit{Left})\ (\mathit{return} \cdot \mathit{Right})\end{array}$$

The input argument is the computation `P` and the other two arguments encode in an untyped way the output, as is shown in Table 2.

7.2 The `entwine/2` Infrastructure

With the help of the delimited continuation primitives, we can implement the infrastructure for `entwine/2` in Prolog, which corresponds to the $(\curlyvee)$ operation.

Since Prolog does not allow us to override the disjunction primitive `(;)/2`, we are forced to use a new name, `or/2`, for expressing its syntactic form.

```
or(G1, G2) :- shift(or(G1, G2)).
```

Prolog's plain disjunction `(;)/2` remains available, allowing programmers to choose between disjunction that can be observed by our framework, and that which cannot. Capturing and handling of syntactic disjunctions is implemented with `reset/3`.

```
step(G, Pattern, Handler) :-
  reset(G, K, D),
  ( K = 0
```

```
    ->  true
    ;   D = or(G1, G2),
        Pattern = or((G1, K), (G2, K)),
        call(Handler)
    ).
```

This enables a straightforward port of the $(\curlyvee)$ implementation:

```
entwine(G1, G2) :-
  step(G1, or(GL1, GR1),
  step(G2, or(GL2, GR2),
  or(entwine(GL1, GL2), entwine(GR1, GR2)))).
```

Finally, the reflection of toplevel syntactic disjunctions into Prolog's original disjunction is handled by `semOr/1`:

```
semOr(G) :-
  step(G, or(G1, G2), (semOr(G1) ; semOr(G2))).
```

While the meta-interpreter must cater for all features in the languages, this delimited continuations-based approach is nicely orthogonal to other features. The code is substantially shorter and clearly requires less maintenance. Moreover, even though raw performance is not the main objective, this approach is almost 3 times faster than the meta-interpreter on search intensive code that does not use $(\curlyvee)$.

A small caveat is in order: In our Haskell model we expect that the following property holds for $p :: \mathit{MonadProlog}\ m \Rightarrow m\ a$:

$$p = \mathit{semOr}\ p$$

The Prolog equivalent of this statement is only true if the programmer avoids calling `or/2` inside a small subset of Prolog's control operations like Prolog's special `catch/3`. Fortunately, this requirement is generally not a heavy burden when solving search problems.

7.3 Search Heuristics

Now that we implemented the entwining infrastructure in Prolog, it is possible to define well-known search heuristics in the same concise and high-level style as in Haskell. As an example, the following Prolog code implements the depth-bounded search heuristic:

```
dbs(Depth,Goal) :- entwine(Goal,dbs(Depth)).

dbs(Depth) :-
  Depth > 0,
  Depth1 is Depth - 1,
  ( dbs(Depth1) or dbs(Depth1) ).
```

We have also implemented other heuristics such as discrepancy-bounded, node-bounded and failure-bounded search, as well as limited discrepancy search, iterative deepening, and branch-and-bound.[3]

As we have remarked in Section 5.4, not all search heuristics can be expressed in terms of `entwine/2`. Fortunately, we can still write custom handlers. One such handler is `muddle/1`:

```
muddle(G) :-
  step(G, or(GL, GR),
    ( random(2) > 0
    ->  or(muddle(GL),muddle(GR))
    ;   or(muddle(GR),muddle(GL))
    )).
```

This handler cannot be expressed with TOR's hookable disjunction [34], because its hooks only manipulate the branches individually and not the disjunction as a whole.

[3] `http://users.ugent.be/~bdsouter/heuristics.html`

7.4 Multi-Way Disjunctions

A final limitation of TOR [34] is that it does not support multi-way (i.e., n-ary) disjunctions. These are useful to express for instance that all the alternatives generated by a call to `select/3` are at the same level in the search tree and thus should be treated equally by depth-bounded search.

With the effect handlers approach, multi-way disjunction can easily be expressed as a generalization of binary disjunctions: the multi-way disjunction predicate `mor/1` takes a list of goals rather than two goals.

```
mor(Gs) :- shift(mor(Gs)).

mstep(G, Pattern, Handler) :-
  reset(G, K, D),
  ( K = 0
  ->  true
  ;   D = mor(Gs),
      maplist(extend(K),Gs,EGs),
      Pattern = mor(EGs),
      call(Handler)
  ).

extend(K,G,(G,K)).
```

A multi-way disjunction can be interpreted in terms of Prolog's binary disjunction.

```
semMor(G) :- mstep(G, mor(Gs), alts(Gs)).

alts([])     :-  fail.
alts([G|Gs])  :-  (G ; alts(Gs)).
```

However, first we can apply heuristics, like depth-bounded search. While it is not obvious how to extend `entwine/2` to multi-way disjunctions, it is easy enough to write regular handlers.

```
mdbs(DB,G) :-
  mstep(G,mor(Gs),
    (DB > 0,
     NDB is DB - 1,
     maplist(mdbs_rec(NDB),Gs,NGs),
     mor(NGs))).

mdbs_rec(DB,G,mdbs(DB,G)).
```

8. Related Work

8.1 Search

FP Models of LP Spivey's algebraic model of logic programming's combinatorial search [37] is very similar to *MonadProlog*. The model was first described by Seres et al. [35] as a way to allow both depth-first and breadth first strategies.

It has long been known that Prolog-like features can be embedded in Haskell using monads and monad transformers. For instance, Hinze [14] provides the equivalent of *MonadProlog* as well as a pruning primitive *once*. We can implement this using $(\curlyvee)$.

Hinze [15] has also derived a backtracking monad transformer using the techniques of term and context passing. Both are systematic ways to derive a program implementation from its specification. The technique thus builds on the laws one imposes on the monad at hand to eliminate the need for a deus ex machina.

Kiselyov et al. [19] derive two implementations of a backtracking monad transformer. The first manages continuations explicitly, while the second does this implicitly using delimited control operations. Unlike our work, their monad transformer provides several extra operations, among which are fair conjunctions and disjunctions, and allows selecting an arbitrary number of answers.

Erwig [9] compares Prolog and Haskell-style approaches to solving search problems. He argues that the Haskell style (which comprises lazy evaluation, static typing and multi-parameter type classes) is better suited. However, search heuristics do not feature.

Functional Logic Programming Typically, Functional Logic Programming (FLP) systems support nondeterminism in the same way as Prolog, with a fixed depth-first search strategy. In order to provide more flexibility, various FLP researchers [4, 23] have investigated *encapsulated search*. Encapsulated search reifies the search tree of a nondeterministic computation in a datastructure similar to $Or^{\star}_{m}$. This reified tree can be explored by programmer-supplied search strategies instead of the default depth-first search.

Given the tree-based interface of FLP encapsulated search, it is a perfect platform for the ideas of this paper: the declarative definition of search heuristics as archetypal search trees, and the modular composition of search trees with the $(\curlyvee)$ operator.

Constraint Programming The constraint logic programming libraries of many Prolog systems [5, 8, 13, 42] provide search heuristics that offer limited reusability: they are hardwired in a generic labelling predicate that can be used to solve particular classes of problems. The one exception is the branch-and-bound heuristic of ECLiPSe [29], which is not tied to a labelling predicate.

Schrijvers et al. [30] present Monadic Constraint Programming, a monadic model of Constraint Programming. This model features an explicit search tree datastructure that is manipulated by search heuristics. Compositionality of search heuristics is achieved by defining them in terms of a set of hooks. This approach is more complex and operational in nature than the one in this paper, which makes it harder for the programmer to define new heuristics and reason about the behaviour of compositions. The hook-based approach is further explored in C++ [33] and Prolog [34] settings, where it suffers from similar problems.

Nordin and Tolmach [25] describe a lazy functional framework for solving constraint satisfaction problems. As in our approach, it is straightforward to express and combine algorithms to prune the search space, using both fixed and dynamic variable ordering. They note an imperative implementation of several combinations of these algorithms is known to be tricky. However, they stress the importance of being able to experiment with them, since the best combination of features tends to depend on the particular problem.

Continuations The explicit use and manipulation of continuations in continuation passing style programs for implementing search is folklore. In the late 1980's, Felleisen [10] and Danvy & Filinski [6] independently proposed operators for delimited continuations in direct style programs. The latter is the reset/shift approach we have adopted in this article, which has a simple static interpretation in terms of continuations.

The CP language *Comet* [38] is a particularly interesting application of this technique: it features fully programmable search [39] based on continuations that make it easy to capture the state of the solver and write non-deterministic code.

8.2 Algebraic Effect Handlers

Plotkin and Pretnar [28] have introduced the concept of handlers for algebraic effects as a generalization of exception handlers. Their approach applies handlers on the free monad. Based on this idea, two entirely new programming languages, Frank [24] and Eff [1], have been created from the ground up around algebraic effect handlers; in these languages the computation monad is implicit.

More recently, three proposals show how to implement algebraic effect handlers on top of existing functional programming languages: Kiselyov et al. [20] provide a Haskell implementation in terms of the free monad, in combination with the codensity transformer to obtain better performance for $(\gg\!=)$. Brady [3] provides a layered implementation: a syntactic monad is interpreted into what is essentially the delimited continuation-based approach of Section 6. Finally, underneath it all is an arbitrary monad m; while Brady only uses this underlying monad in the handler definitions, his handler infrastructure is in fact a monad transformer. Kammar et al. [18] present several different implementations in Haskell, OCaml, SML and Racket. For Haskell, the free monad and a continuation-based approach are considered. For the other languages, the delimited continuation approach is taken.

8.3 Monads

Monadic Zip The literature covers a number of *zip*-like monadic operations similar to our $(\curlyvee)$: Giorgidze et al. [12] introduce a monadic zip operator *mzip* to support parallel monad comprehensions, a generalization of parallel list comprehensions. Their *mzip* is subject to two laws: it must have a partial inverse *munzip*, and it must be associative.

As part of their *Joinad* concept, Petricek et al. [27] define monadic operations similar to ours, including a monadic *zip*. However, the laws associated with their operations make them different in important ways from ours. Notably, while our $(\curlyvee)$ and $(|||)$ commute, for *Joinad*s the *zip* operator left-distributes over *or*.

Monad Laws Gibbons and Hinze [11] promote reasoning about code that is polymorphic in the monad by means of laws, which is the starting point of this paper. They illustrate law-based reasoning on several related monadic effects: failure, nondeterministic choice and probabilistic choice.

Free Monad Transformer The free monad (transformer) is also known by various other names, emphasizing different aspects: coroutine monad [2], resumption monad [26] and step monad [17]. The coroutine aspects is very relevant in our setting: in essence, the $(\curlyvee)$ operation treats two searches as coroutines that are synchronized at corresponding occurrences of $(|||)$.

9. Conclusion

This paper has exploited the synergy between two declarative paradigms to tackle a challenging problem in logic programming with functional programming techniques. First it has shown how to cleanly separate logic and search heuristics in a functional model of Prolog by means of effect handlers and the free monad transformer. Then it has derived an actual Prolog implementation from this functional specification.

We are keen to use effect handlers to further extend Prolog with control operations that interact with the ones presented in this work. Of particular interest is Spivey's *wrap* [37] that groups multiple binary disjunctions into a single multi-way disjunction.

Acknowledgments

We are grateful to the members of IFIP WG 2.1 for feedback on this work, to Olivier Danvy for discussions about delimited continuations, and to Ralf Hinze for his detailed comments on an early draft of this paper. This work has been funded by EPSRC grant number EP/J010995/1, on Unifying Theories of Generic Programming, and by the Flemish Fund for Scientific Research.

References

[1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers, 2012. `arXiv:1203.1539`.

[2] M. Blazevic. monad-coroutine: Coroutine monad transformer for suspending and resuming monadic computations, 2010. `http://hackage.haskell.org/package/monad-coroutine`.

[3] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, ICFP '13, pages 133–144. ACM, 2013.

[4] B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.

[5] M. Carlsson and P. Mildner. SICStus Prolog - The first 25 years. *Theory and Practice of Logic Programming*, 12(1-2):35–66, 2012.

[6] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, LFP '90, pages 151–160. ACM, 1990.

[7] B. Demoen and P.-L. Nguyen. So many WAM Variations, so little Time. In *Computational Logic - CL2000, First International Conference, Proceedings*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1240–1254. ALP, Springer, 2000.

[8] D. Diaz, S. Abreu, and P. Codognet. On the implementation of GNU-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):253–282, 2012.

[9] M. Erwig. Escape from Zurg: an exercise in logic programming. *J. Funct. Program.*, 14(3):253–261, May 2004.

[10] M. Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 180–190. ACM, 1988.

[11] J. Gibbons and R. Hinze. Just do it: simple monadic equational reasoning. In *Proc. ICFP*, pages 2–14. ACM, 2011.

[12] G. Giorgidze, T. Grust, N. Schweinsberg, and J. Weijers. Bringing back monad comprehensions. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 13–22. ACM, 2011.

[13] M. V. Hermenegildo, F. Bueno, M. Carro, P. López-García, E. Mera, J. F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[14] R. Hinze. Prological Features In A Functional Setting Axioms And Implementations. In *Third Fuji Int. Symp. on Functional and Logic Programming*, pages 98–122, 1998.

[15] R. Hinze. Deriving backtracking monad transformers. *SIGPLAN Not.*, 35(9):186–197, Sept. 2000.

[16] G. Hutton and D. Fulger. Reasoning About Effects: Seeing the Wood Through the Trees. In *Pre-proceedings of the Symposium on Trends in Functional Programming*, 2008. Unpublished, available at `http://www.cs.nott.ac.uk/~gmh/effects.pdf`.

[17] M. Jaskelioff and E. Moggi. Monad transformers as monoid transformers. *Theor. Comput. Sci.*, 411(51-52):4441–4466, Dec. 2010.

[18] O. Kammar, S. Lindley, and N. Oury. Handlers in action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional programming*, ICFP '13, pages 145–158. ACM, 2013.

[19] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers. In *Proc. ICFP'05*, pages 192–203. ACM, 2005.

[20] O. Kiselyov, A. Sabry, and C. Swords. Extensible effects: an alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN symposium on Haskell*, Haskell '13, pages 59–70. ACM, 2013.

[21] R. Kowalski. Algorithm = logic + control. *Commun. ACM*, 22(7): 424–436, July 1979.

[22] B. Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences of the United States of America*, 50(1):869–872, 1963.

[23] W. Lux. Implementing encapsulated search for a lazy functional logic language. In *Fuji International Symposium on Functional and Logic Programming*, volume 1722 of *Lecture Notes in Computer Science*, pages 100–113. Springer, 1999.

[24] C. McBride. The Frank manual, May 2012. `https://personal.cis.strath.ac.uk/conor.mcbride/pub/Frank/TFM.pdf`.

[25] T. Nordin and A. Tolmach. Modular lazy search for constraint satisfaction problems. *J. Funct. Program.*, 11(5):557–587, Sept. 2001.

[26] N. S. Papaspyrou. A Resumption Monad Transformer and its Applications in the Semantics of Concurrency. Technical Report CSD-SW-TR-2-01, National Technical University of Athens, 2001.

[27] T. Petricek, A. Mycroft, and D. Syme. Extending monads with pattern matching. In *Proc. Haskell'11*, pages 1–12. ACM, 2011.

[28] G. D. Plotkin and M. Pretnar. Handlers of algebraic effects. In *ESOP*, volume 5502 of *Lecture Notes in Computer Science*, pages 80–94. Springer, 2009.

[29] J. Schimpf and K. Shen. ECLiPSe – From LP to CLP. *Theory and Practice of Logic Programming*, 12(1-2):127–156, 2012.

[30] T. Schrijvers, P. Stuckey, and P. Wadler. Monadic constraint programming. *Journal of Functional Programming*, 19(6):663–697, Nov. 2009.

[31] T. Schrijvers, M. Triska, and B. Demoen. Tor: Extensible search with hookable disjunction. In A. King, editor, *Principles and Practice of Declarative Programming, 14th International ACM SIGPLAN Symposium, Proceedings*, pages 103–114. ACM, 2012.

[32] T. Schrijvers, B. Demoen, B. Desouter, and J. Wielemaker. Delimited continuations for Prolog. *Theory and Practice of Logic Programming (TPLP)*, 13(4-5):533–546, 2013. Proceedings of the International Conference on Logic Programming (ICLP).

[33] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, and P. Stuckey. Search combinators. *Constraints*, 18(2):269–305, 2013.

[34] T. Schrijvers, B. Demoen, M. Triska, and B. Desouter. Tor: Modular search with hookable disjunction. *Sci. Comput. Program.*, 84:101–120, 2014.

[35] S. Seres, M. Spivey, and T. Hoare. Algebra of logic programming. In *International Conference on Logic Programming*, pages 184–199. Palgrave MacMillan, 1999.

[36] D. Sitaram. Handling control. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 147–155, New York, NY, USA, 1993. ACM. ISBN 0-89791-598-4. . URL `http://doi.acm.org/10.1145/155090.155104`.

[37] J. M. Spivey. Algebras for combinatorial search. *J. Funct. Program.*, 19(3-4):469–487, July 2009.

[38] P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press, 2005.

[39] P. Van Hentenryck and L. Michel. Nondeterministic control for hybrid search. *Constraints*, 11(4):353–373, 2006.

[40] P. Wadler. The essence of functional programming. In *POPL'92: 19th Symposium on Principles of Programming Languages*, pages 1–14, 1992.

[41] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical report, SRI International, 1983.

[42] N.-F. Zhou. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218, 2012.