Heuristics Entwined with Handlers Combined

Tom Schrijvers



with Nicolas Wu, Benoit Desouter and Bart Demoen

Motivation



Tor

Extensible Search with Hookable Disjunction

Tom Schrijvers
Ghent University

Markus Triska
Vienna University of Technology

Bart Demoen
KU Leuven

Abstract

Horn Clause Programs have a natural depth-first procedural semantics. However, for many programs this procedural semantics is unsatisfactory. In order to compute useful solutions, one needs the ability to modify the search method that explores the alternative execution branches.

Tor, a well-defined hook into Prolog disjunction, provides this ability. It is light-weight thanks to its library approach and efficient because it is based on program transformation. Tor is general enough to mimic search-modifying predicates like ECLiPSe's `minof/2`. Moreover, Tor supports modular composition of search methods and other hooks. Our library is already provided and used as an add-on to SWI-Prolog.

Categories and Subject Descriptors *F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs*

General Terms Languages

Keywords Prolog, disjunction, search, modularity

1. Introduction

Kowalski's well-known adage [9] crisply captures the essence of programming in the equation:

ALGORITHM = LOGIC + CONTROL

In Prolog, the LOGIC part is captured in the programmer-supplied rules or clauses that have a first-order logic interpretation. The CONTROL component is supplied by the Prolog engine and essentially consists of search. In order to answer queries, a Prolog engine performs a backward-chaining depth-first tree search.

Prolog's default search strategy is in practice inadequate to effectively scour large search spaces. As a consequence, the programmer often has to complement Prolog's CONTROL with additional hints or heuristics in the form of extra code. This is particularly prevalent in the context of Constraint Logic Programming where it is common practice for the programmer to complement a constraint model with a search specification.

Unfortunately, it is not all that easy to cleanly separate LOGIC and CONTROL when implementing search heuristics in Prolog. When one discovers that Prolog's CONTROL is ineffective, it is often impossible to orthogonally add one's own CONTROL without touching the existing LOGIC. The problem is that syntactically logic and control in Prolog are generally tightly coupled, and adding a different control means cross-cutting existing code.

In this paper we present a novel approach to adding, in an orthogonal manner, CONTROL. Our solution features the following properties:

- It is a light-weight library-based approach that is easily portable to different Prolog systems: it is currently an SWI-Prolog library [22].
- Our approach has all the benefits of modularity: search methods can be composed and the library of these heuristics is (user-)extensible.
- Its overhead is minimal, as we demonstrate on benchmarks: this is achieved through `term_expansion/2`, a feature present in most Prolog systems.

With Tor, we capture all common search methods in CLP(FD)'s libraries such as ECLiPSe's `search/6` [23]. This approach is indeed particularly suitable for Constraint Logic Programming, but also useful for general Prolog programs with a large search space.

2. Problem Statement

We illustrate the heart of the matter on a simple labeling predicate `label/1` written against SWI-Prolog's `clpfd` library [18] (see Fig. 1, left). `label/1` defines a search tree where the branches are created by the disjunction.[1]

Suppose that for a certain call `label([X1,...,Xn])` the search tree is too large to fully explore. In order to get some useful answers, we may decide to leave certain parts of the tree unexplored, conceptually pruning the tree. One particular way in which we can do this is by reaping the low-hanging solutions only, and pruning the subtrees that are below a certain depth. Hence, we impose a *depth bound* on Prolog's depth-first search. This is achieved by modifying the `label/1` code to that of `label/2` (Fig. 1, right) where the second parameter is the depth-bound.

Imposing a depth bound may or may not be a successful approach to getting useful answers. If it turns out to be unsuccessful, we can try other pruning strategies like imposing a node bound or a discrepancy bound. Each of these requires rewriting the `label/1` predicate to incorporate a different pruning technique. In general, we follow an explorative process whereby we write and evaluate several different variants of the labeling code until we find an effective pruning strategy.

[1] `fd_sup/1` returns the smallest value in a variable's finite domain.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'12, September 19–21, 2012, Leuven, Belgium.
Copyright © 2012 ACM 978-1-4503-1522-7/12/09 ... $5.00.

☛ **Tor: extensible search with hookable disjunction**
T.S., M. Triska, B. Demoen. (PPDP'12)

Kowalski:

ALGORITHM

=

LOGIC

+

CONTROL

Prolog:

ALGORITHM

=

LOGIC

+

DEPTH FIRST SEARCH

Prolog:

EFFECTIVE

ALGORITHM

=

LOGIC×HEURISTIC

+

DEPTH FIRST SEARCH

```
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)
  ).
```

+ Depth-Bounded Search

```
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)
  ).
```

Tor: Modular Heuristics

User Facing:

✔ modular definition

```
dbs/2,queens/2
```

✔ trivial composition

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

Tor: Modular Heuristics

User Facing:

✔ modular definition

`dbs/2, queens/2`

✔ trivial composition

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

Implementor Facing:

✘ no semantic grounding

leaky abstraction,
no algebraic properties

✘ limited expressivity

e.g., flip branches

This Paper

User Facing:

✔ modular definition
`dbs`/2, `queens`/2

✔ trivial composition
```
?- dbs(10,queens(8,Qs)).
```

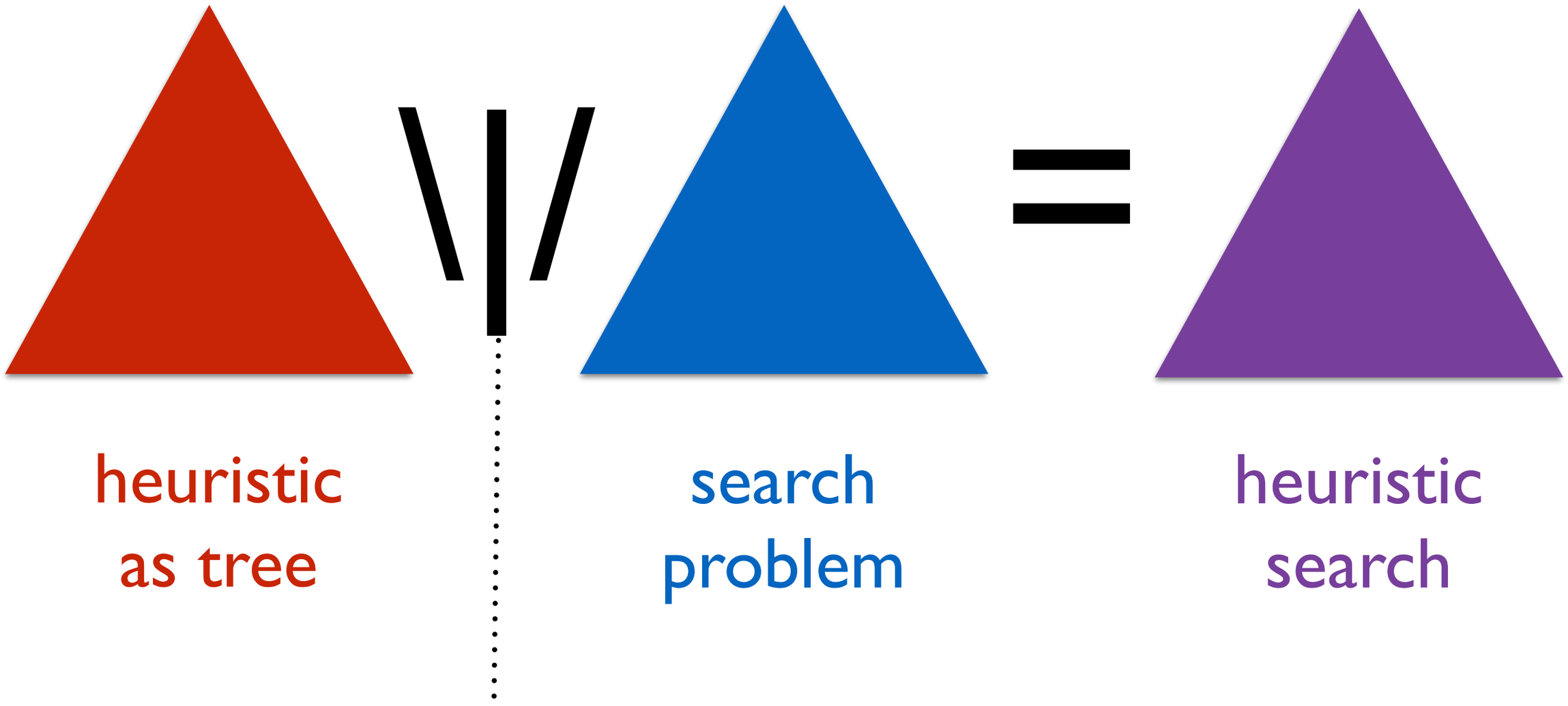
Implementor Facing:

✔ firm semantic grounding
nice algebraic properties

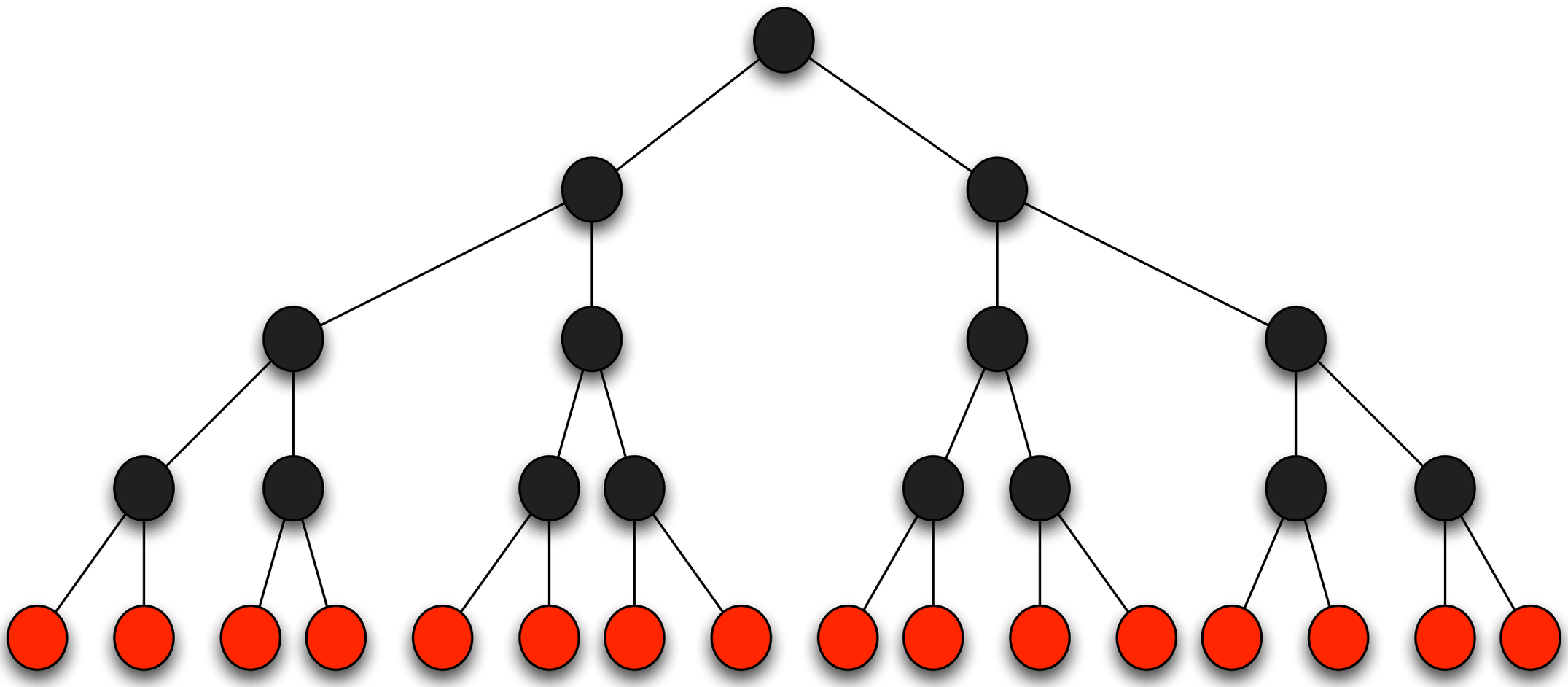
✔ rich expressivity
e.g., flip branches

What I'm promising

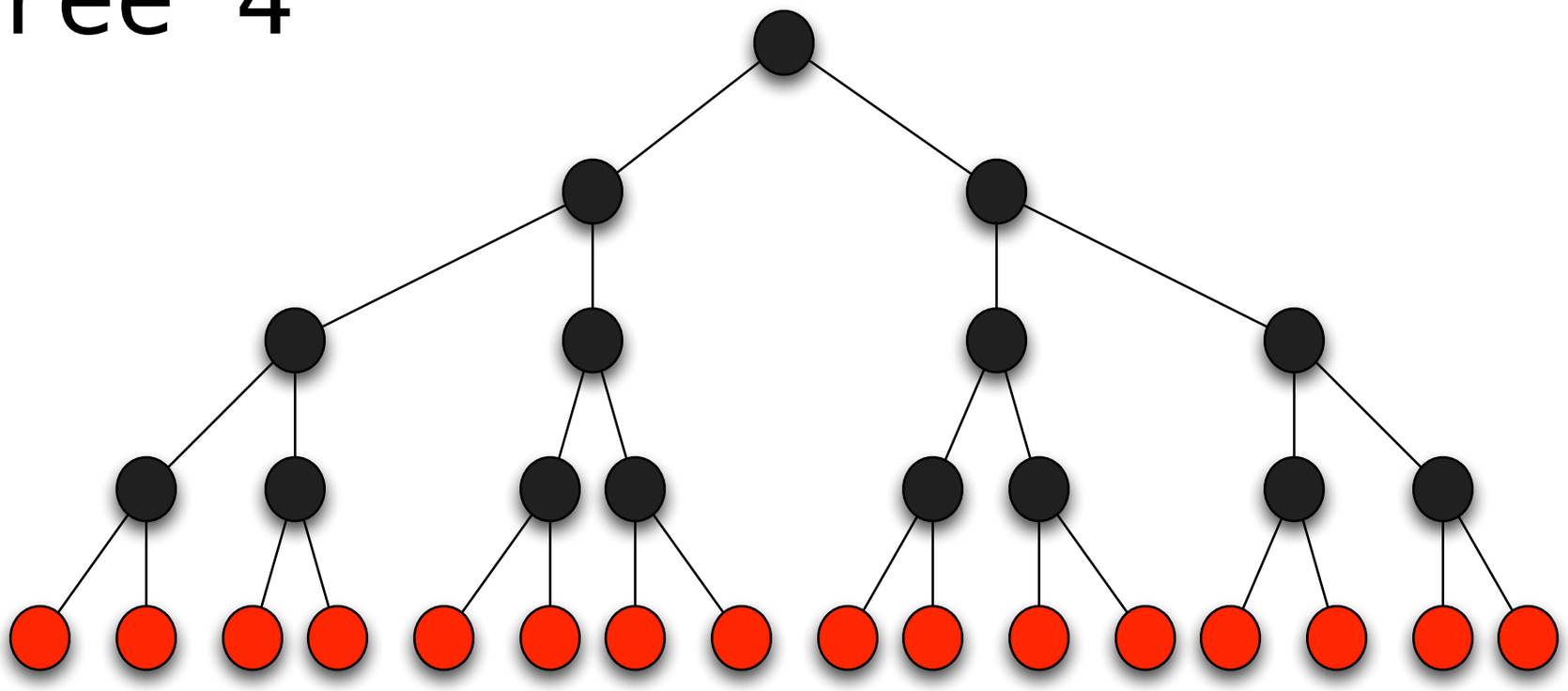
Heuristics as Trees



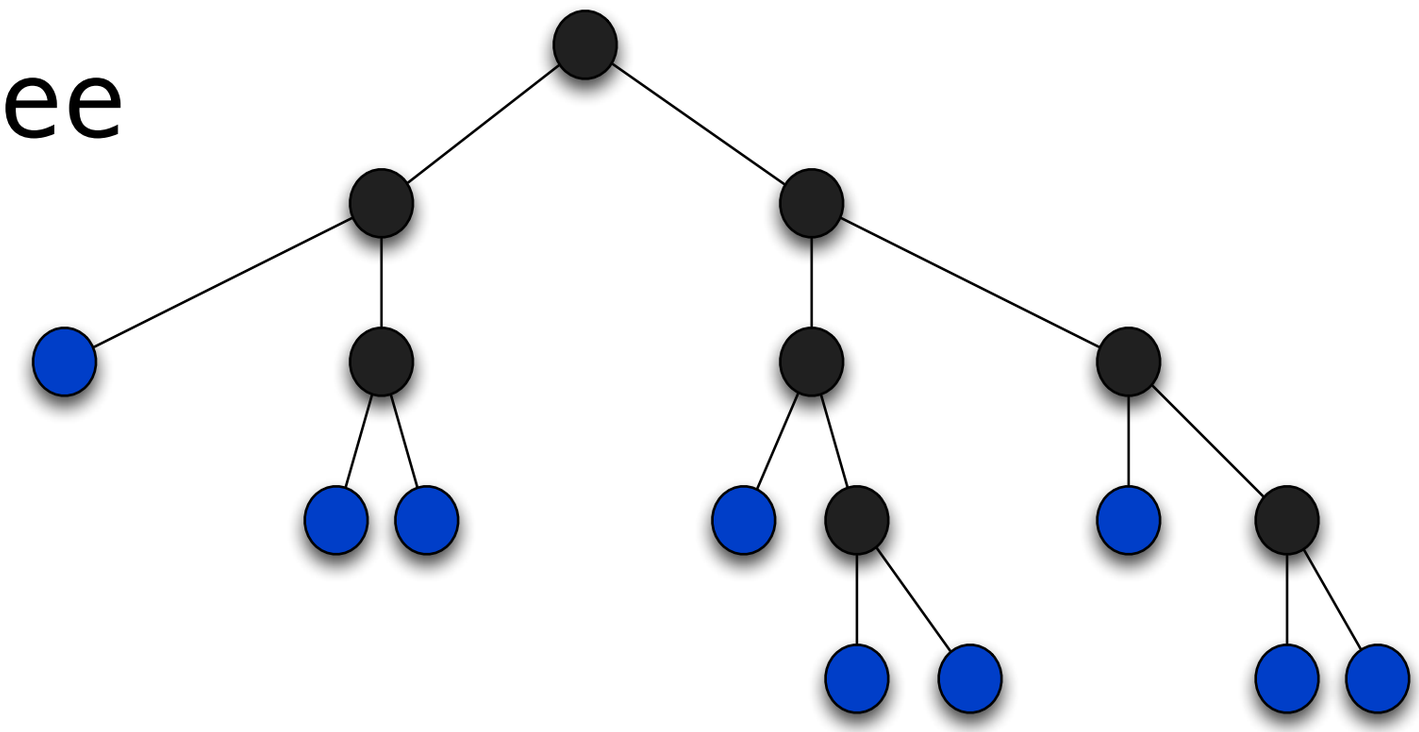
Archetypal DBS Tree

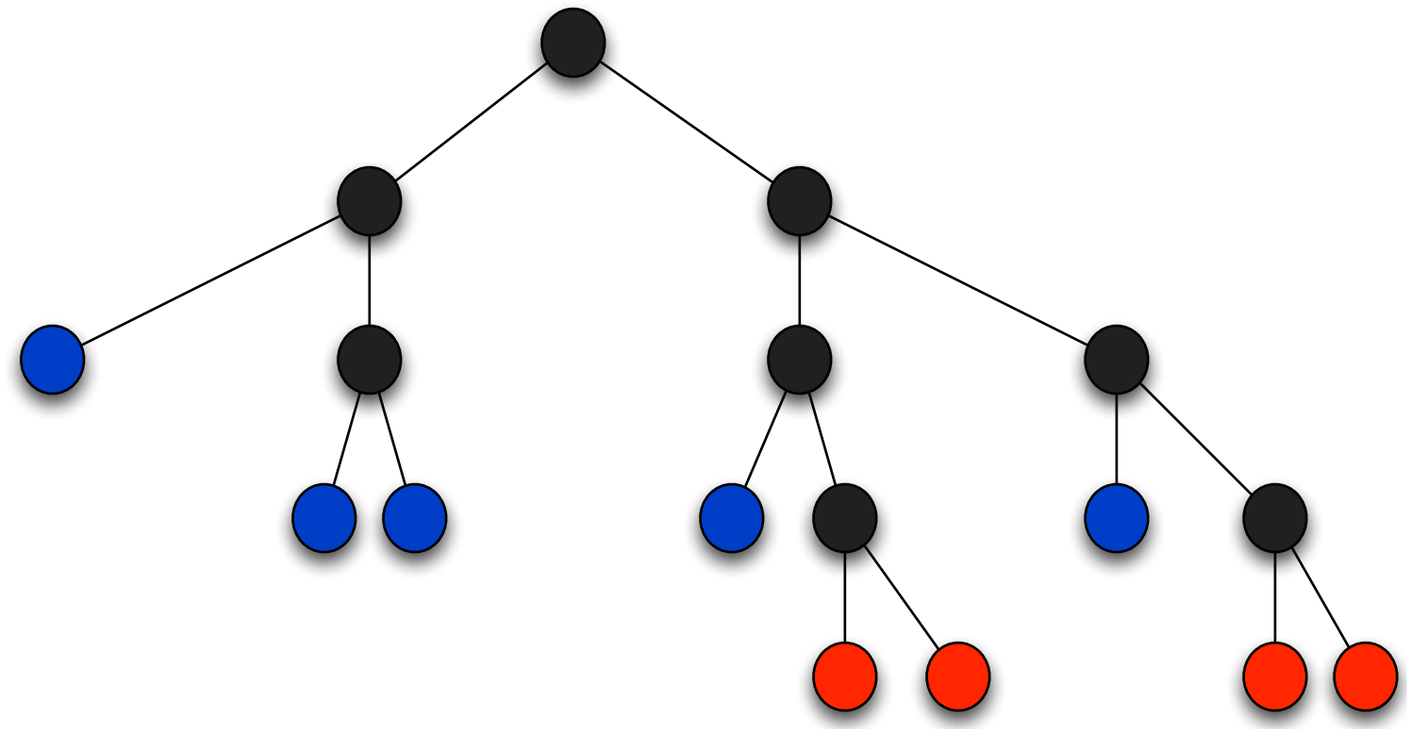


dbsTree 4



problemTree





```
dbsTree 4 \|/ problemTree
```

Nice Algebraic Properties

```
inf \|/ p ≡ p

p \|/ inf ≡ p

p \|/ (q \|/ r) ≡ (p \|/ q) \|/ r
```

```
inf :- inf ; inf.
```

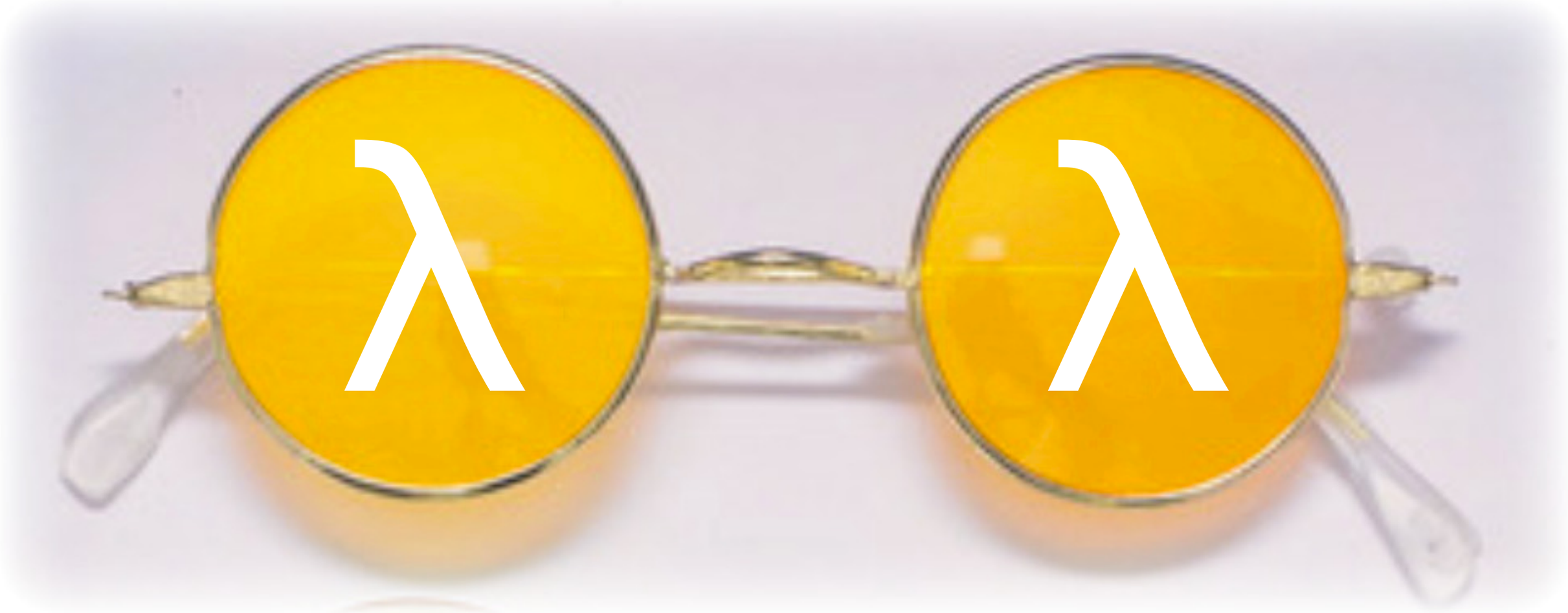
More Expressive Fallback

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

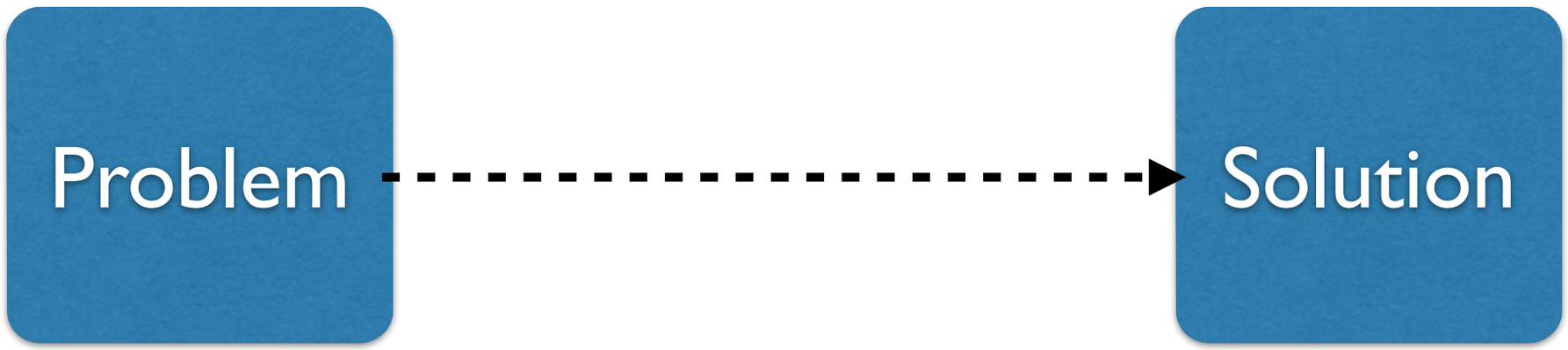
visit branches in random order

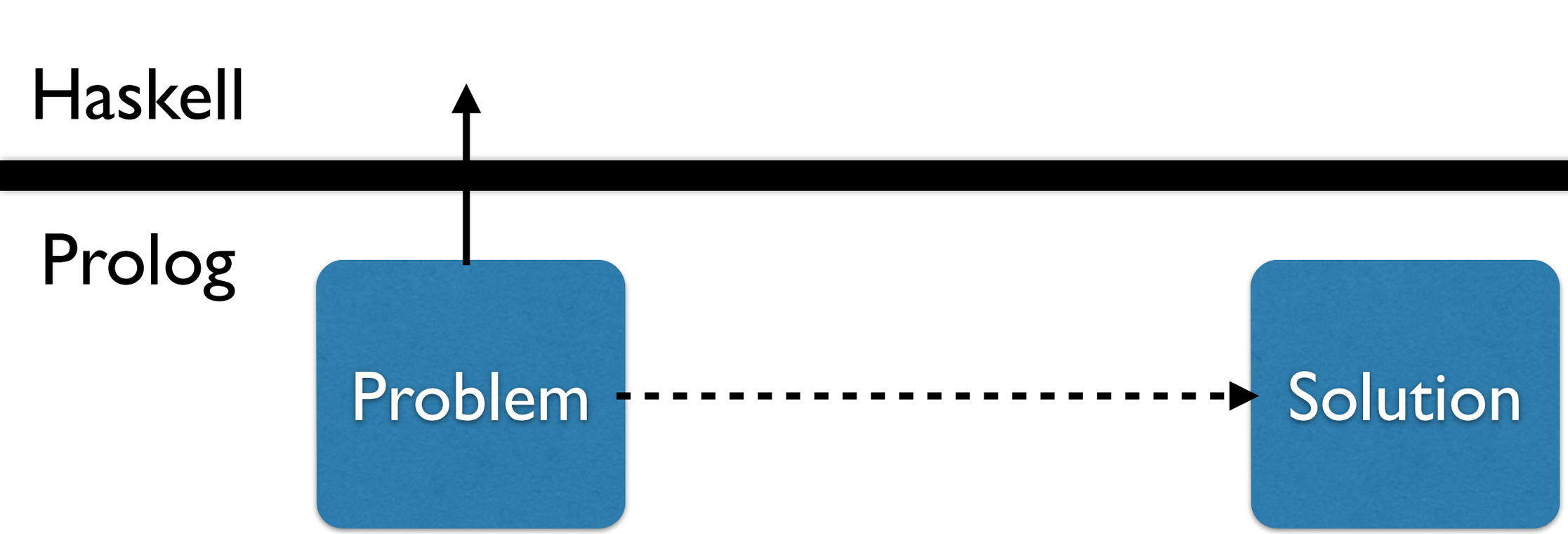
Plan of Attack

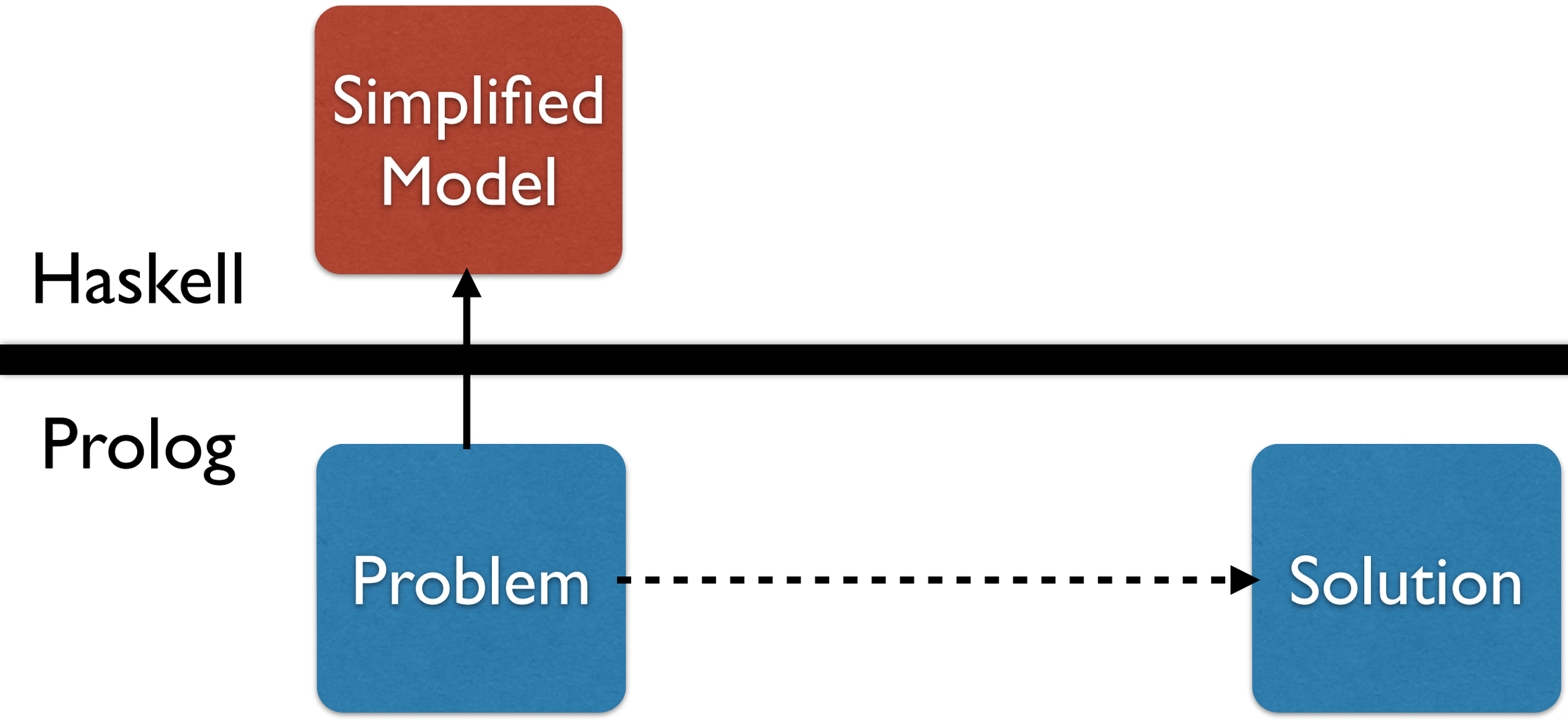


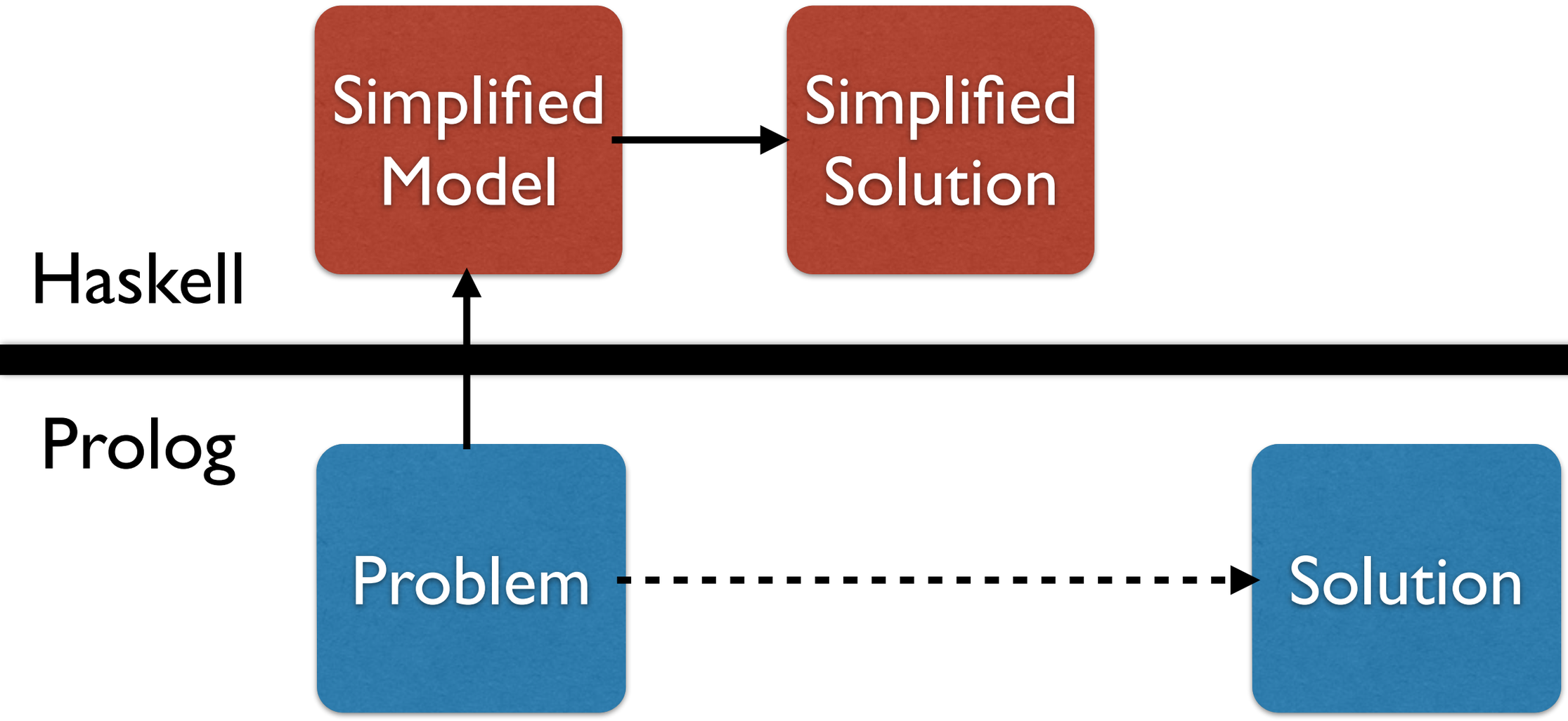


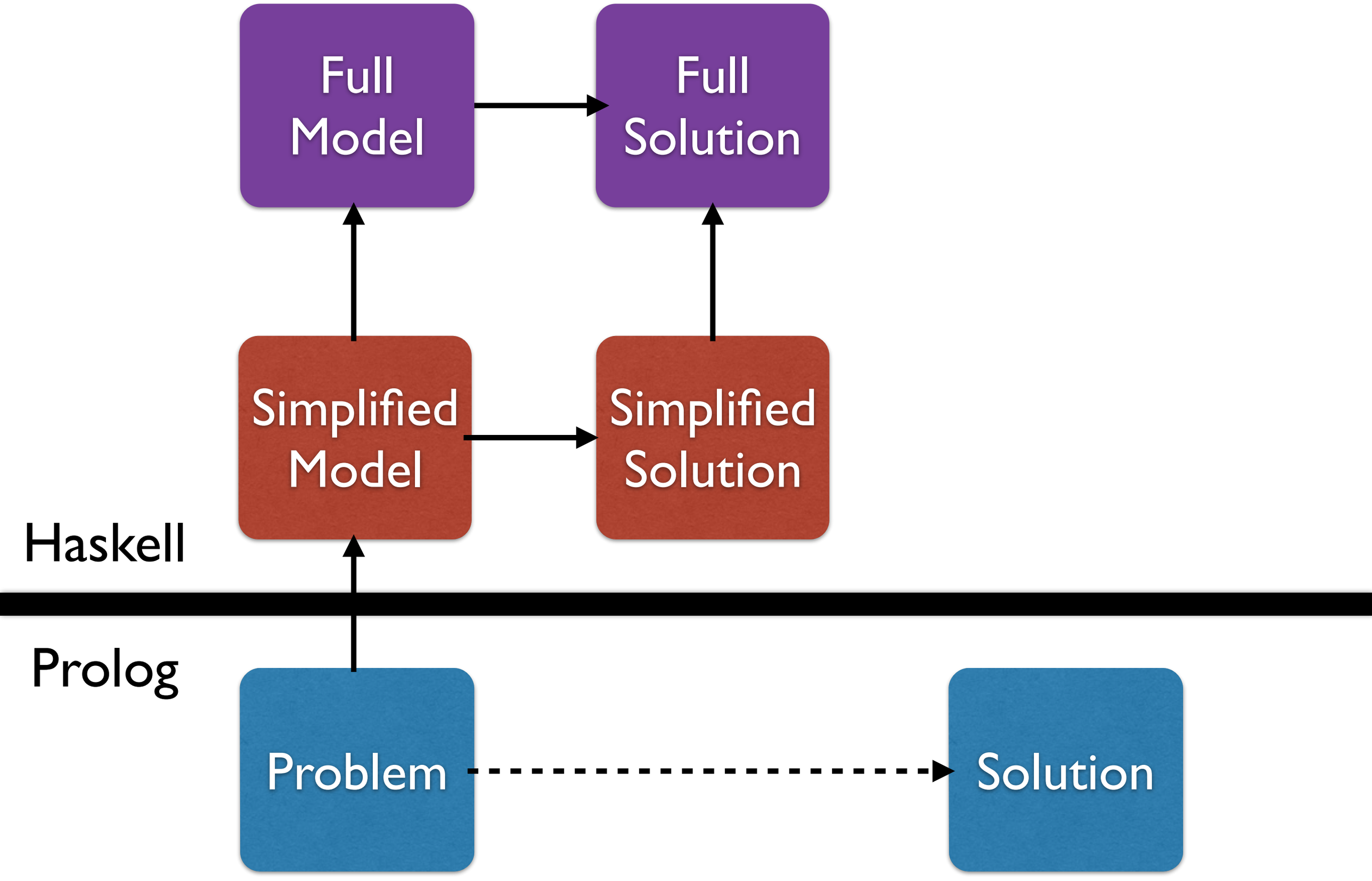
Prolog

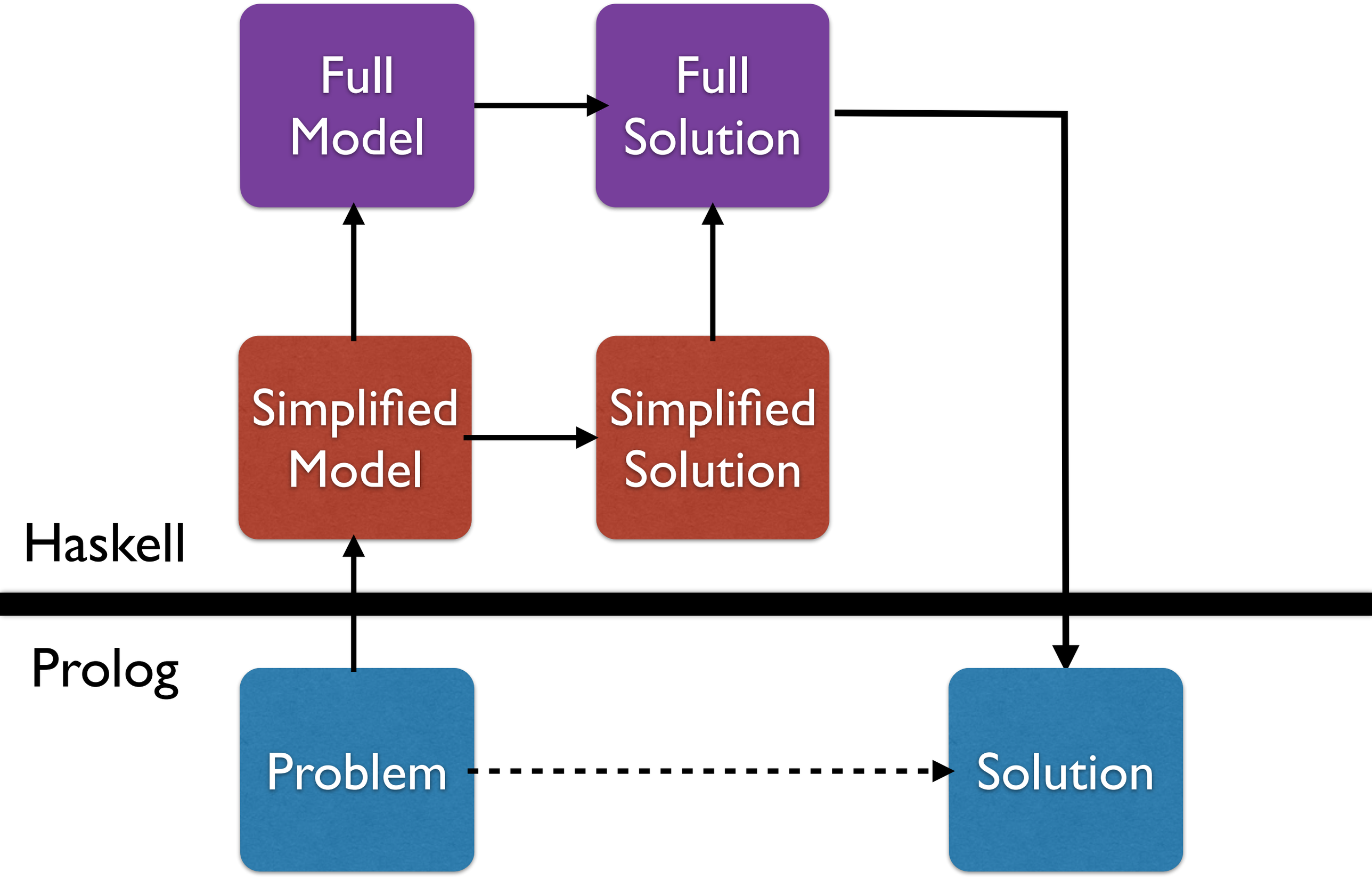



Haskell
Prolog
Problem
Solution


Simplified Model
Haskell
Prolog
Problem
Solution


Simplified Model
Simplified Solution
Haskell
Prolog
Problem
Solution

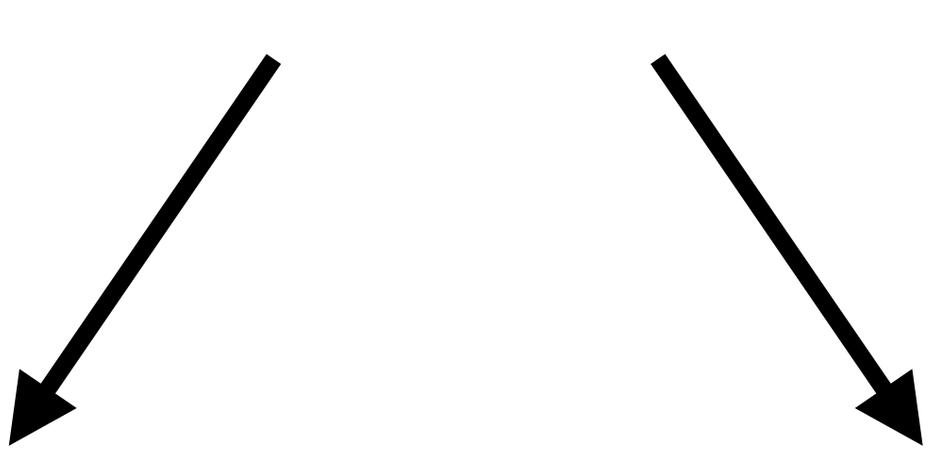

Full Model
Full Solution
Simplified Model
Simplified Solution
Haskell
Prolog
Problem
Solution


Full Model
Full Solution
Simplified Model
Simplified Solution
Haskell
Prolog
Problem
Solution

Simplified Model of Prolog in Haskell

Algebraic Effect Handlers Approach

Monad



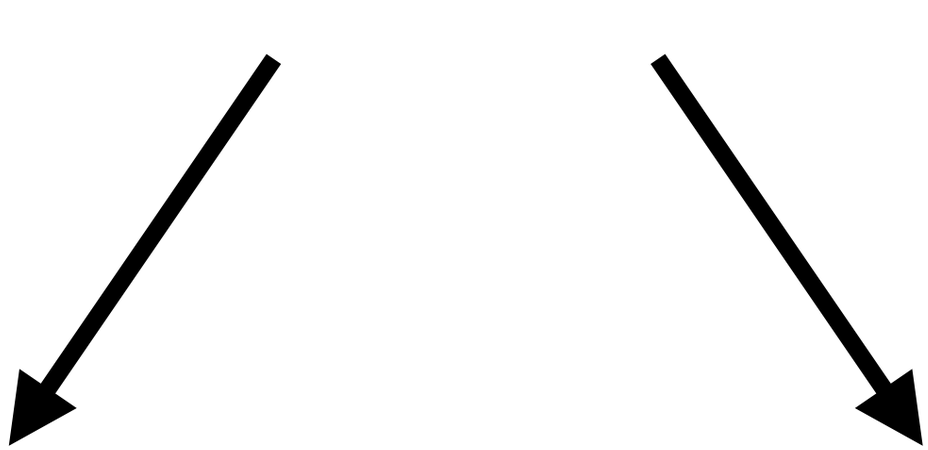
Syntax

Semantics

[Plotkin&Power 2002] [Kiselyov et al. 2013]
[Kammar et al. 2013] [Brady 2013]

Algebraic Effect Handlers Approach

Monad



Syntax

Semantics

Prolog Syntax

```haskell
data Prolog a
  = Return a
  | Fail
  | Or (Prolog a) (Prolog a)
```

Prolog	Haskell
`true`	`Return ()`
`false`	`Fail`
`G1 ; G2`	`Or g1 g2`

Prolog Syntax

```haskell
data Prolog a
  = Return a
  | Fail
  | Or (Prolog a) (Prolog a)
```

Prolog	Haskell
true	Return ()
false	Fail
G1 ; G2	Or g1 g2
G1 , G2	

Prolog Syntax

```haskell
data Prolog a
  = Return a
  | Fail
  | Or (Prolog a) (Prolog a)

instance Monad Prolog where
  return x  =  Return x

  Return x >>= f  =  f x
  Fail >>= f      =  Fail
  Or l r >>= f    =  Or (l >>= f) (r >>= f)

p >> q  =  p >>= \_ -> q
```

Prolog	Haskell
true	Return ()
false	Fail
G1 ; G2	Or g1 g2
G1 , G2	

Prolog Syntax

```haskell
data Prolog a
  = Return a
  | Fail
  | Or (Prolog a) (Prolog a)

instance Monad Prolog where
  return x  =  Return x

  Return x >>= f  =  f x
  Fail >>= f      =  Fail
  Or l r >>= f    =  Or (l >>= f) (r >>= f)

p >> q  =  p >>= \_ -> q
```

Prolog	Haskell
true	Return ()
false	Fail
G1 ; G2	Or g1 g2
G1 , G2	g1 >> g2

Prolog Semantics

```
sem :: Prolog a -> [a]
sem (Return x)  =  [x]
sem Fail        =  []
sem (Or l r)    =  sem l ++ sem r
```

Prolog Semantics

```
sem :: Prolog a -> [a]
sem (Return x)  =  [x]
sem Fail        =  []
sem (Or l r)    =  sem l ++ sem r
```

depth-first search

Simplified Solution

Heuristics

```
type Heuristic a =
  Prolog a -> Prolog a
```

Heuristics

```
type Heuristic a =
  Prolog a -> Prolog a

dbs :: Int -> Heuristic a
dbs 0 p                 =  Fail
dbs n (Return x)        =  Return x
dbs n Fail              =  Fail
dbs n (Or l r)          =
  Or (dbs (n-1) l) (dbs (n-1) r)
```

Compositional Heuristics

```
sem                    :: Prolog a -> [a]
```

Compositional Heuristics

```
sem                    :: Prolog a -> [a]

        dbs d :: Heuristic a
```

Compositional Heuristics

```
sem                    :: Prolog a -> [a]

dbs d :: Heuristic a

sem . dbs d            :: Prolog a -> [a]
```

Compositional Heuristics

```
sem                    :: Prolog a -> [a]
```

```
dbs d :: Heuristic a
```

```
sem . dbs d            :: Prolog a -> [a]
```

```
muddle :: Heuristic a
```

Compositional Heuristics

```
sem                    :: Prolog a -> [a]

dbs d :: Heuristic a

sem . dbs d            :: Prolog a -> [a]

muddle :: Heuristic a

sem . muddle           :: Prolog a -> [a]
```

Compositional Heuristics

```
sem                    :: Prolog a -> [a]

        dbs d :: Heuristic a

sem . dbs d            :: Prolog a -> [a]

        muddle :: Heuristic a

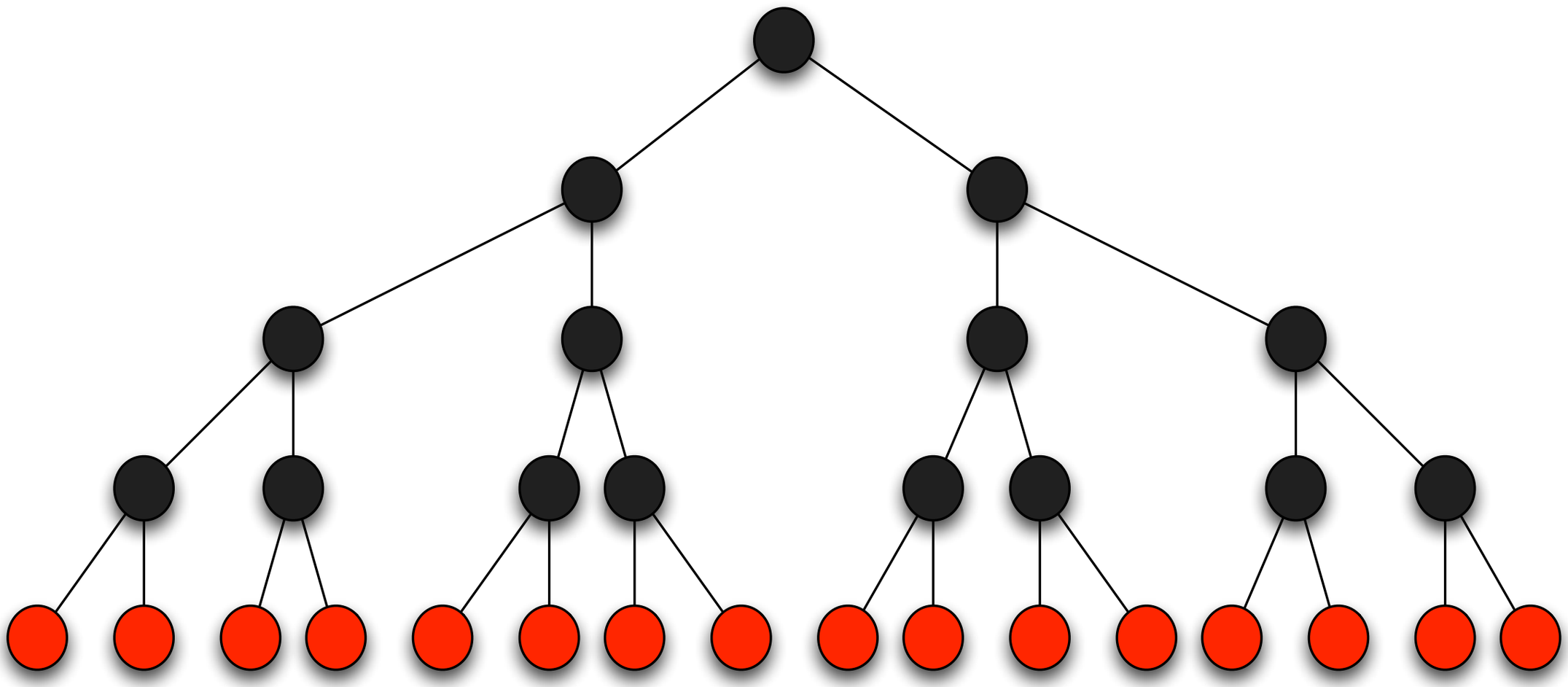
sem . muddle           :: Prolog a -> [a]

sem . dbs d . muddle :: Prolog a -> [a]
```

Entwine Operator

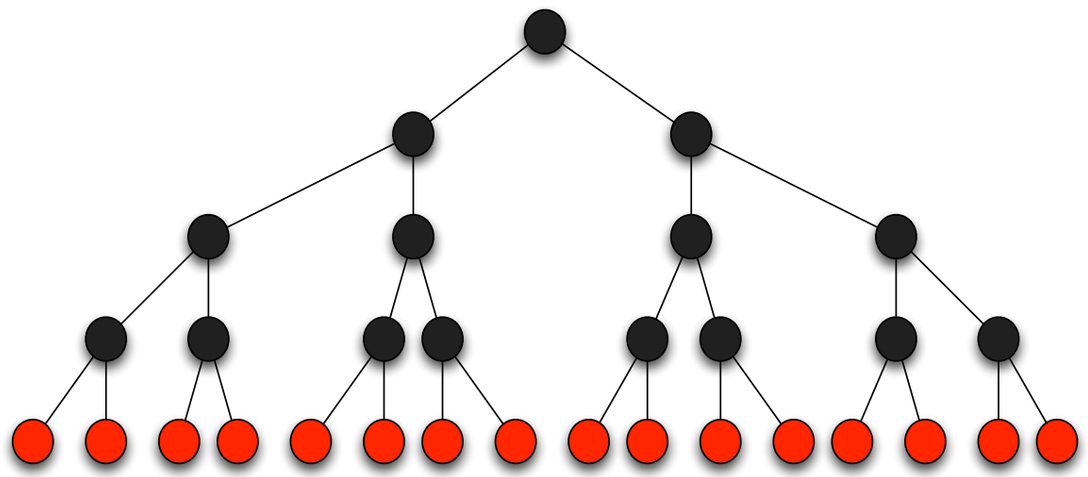
```
(\|/)
  :: Prolog a -> Prolog a -> Prolog a
(Return x) \|/  t                  = Return x
Fail       \|/  t                  = Fail
(Or l1 r1) \|/ (Or l2 r2)  =
    Or (l1 \|/ l2) (r1 \|/ r2)
(Or l1 r1) \|/ t                   = t
```

Archetypal DBS Tree

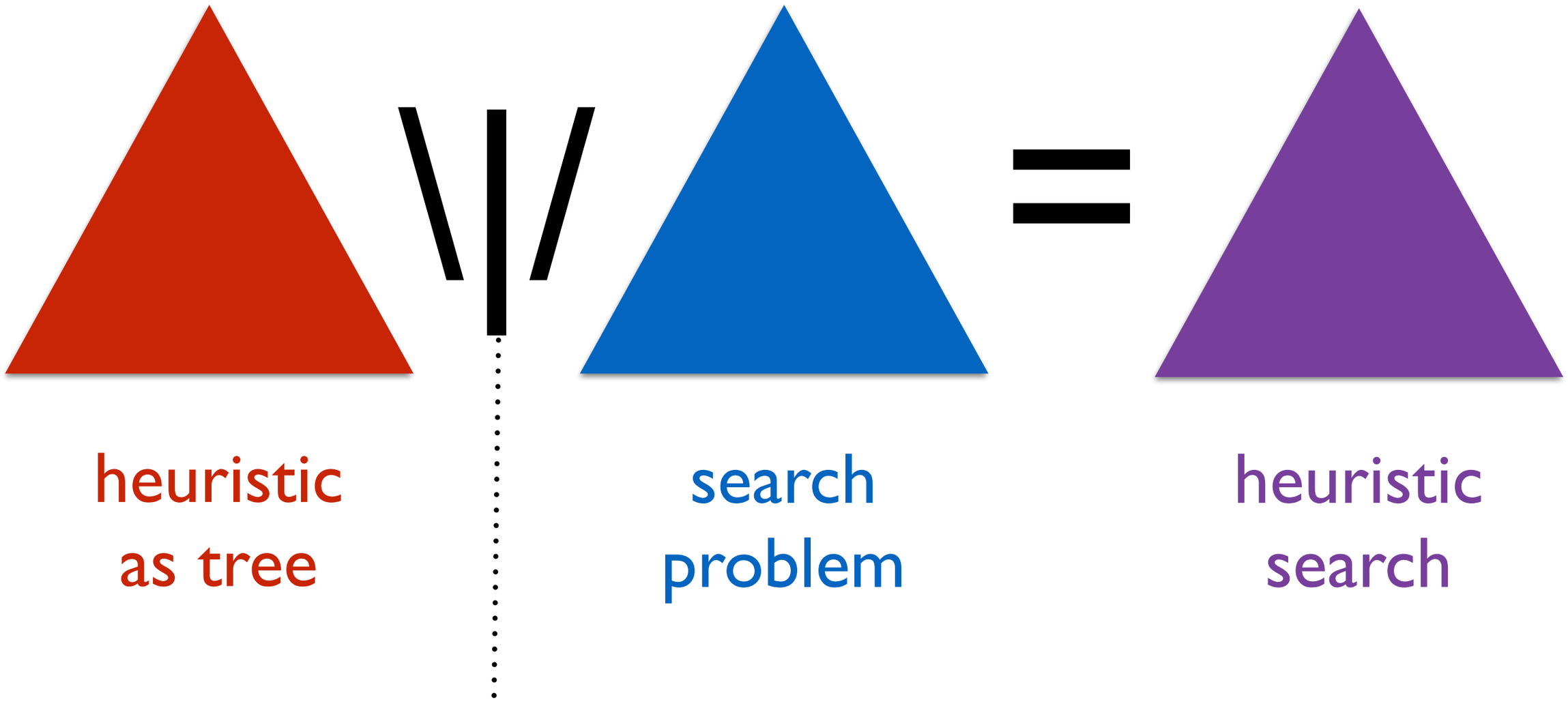


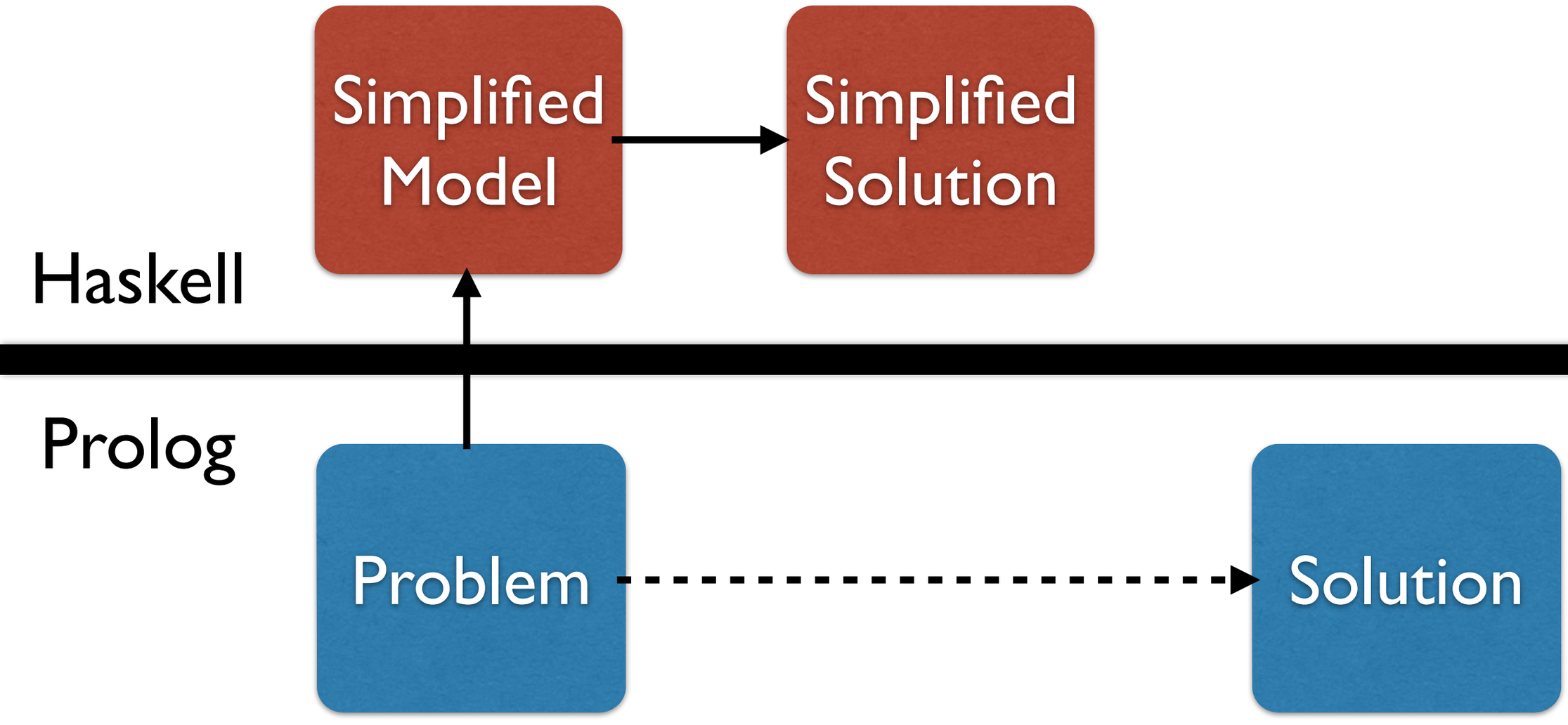
Archetypal DBS Tree

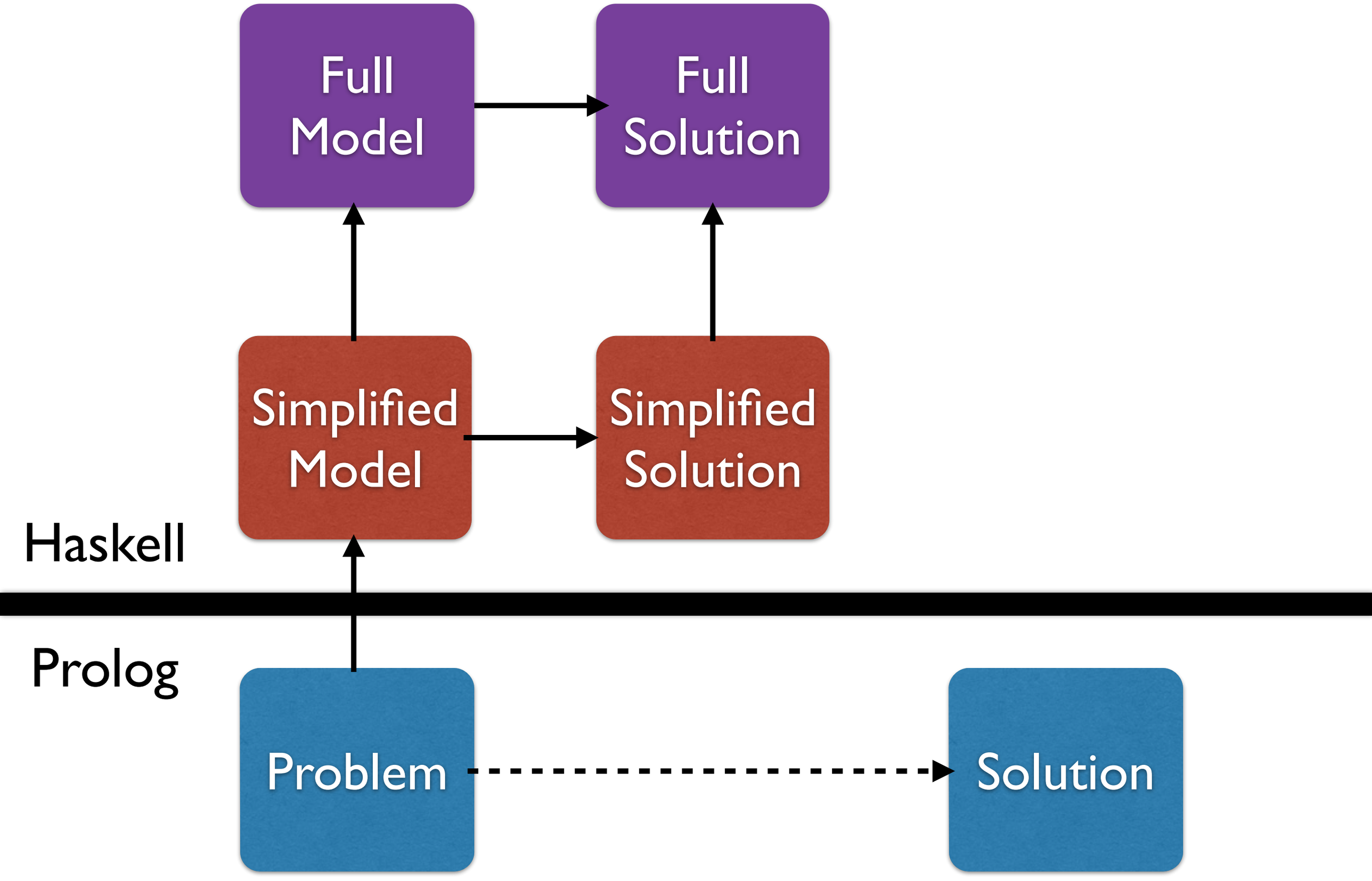
```
dbsTree 0 =
  Fail
dbsTree n =
  Or (dbsTree (n-1))
     (dbsTree (n-1))
```



Heuristics as Trees




Simplified Model
Simplified Solution
Haskell
Prolog
Problem
Solution


Full Model
Full Solution
Simplified Model
Simplified Solution
Haskell
Prolog
Problem
Solution

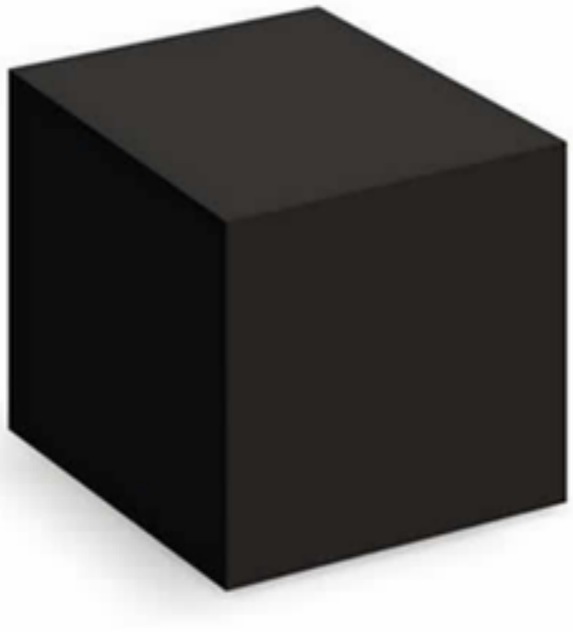
Full Model

Full Model

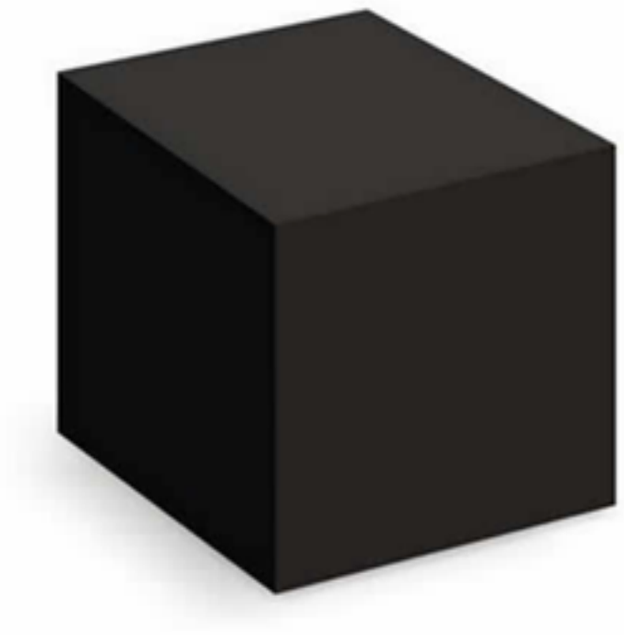
★ full-scale Prolog
lots of Prolog features

★ performance
WAM-level efficiency

Black Box Model of WAM



Black Box Model of WAM

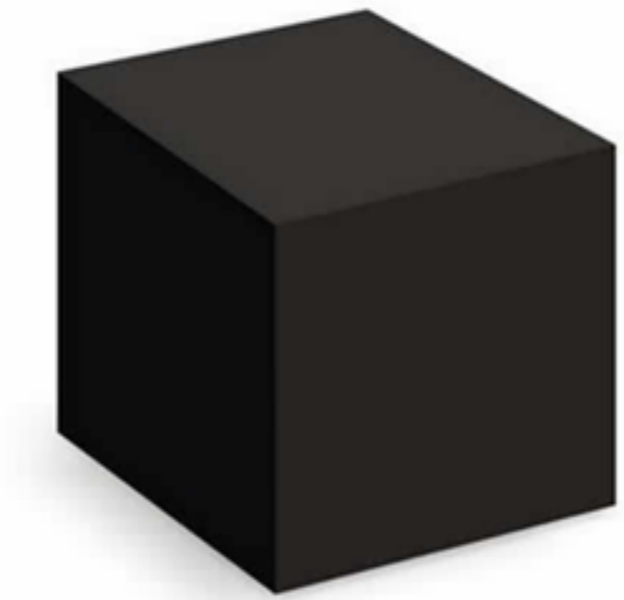


```
data Prolog a -- abstract

instance Monad Prolog

instance MonadProlog Prolog
```

Black Box Model of WAM



```
data Prolog a -- abstract

instance Monad Prolog

instance MonadProlog Prolog
```

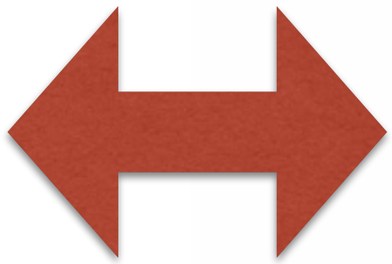
```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

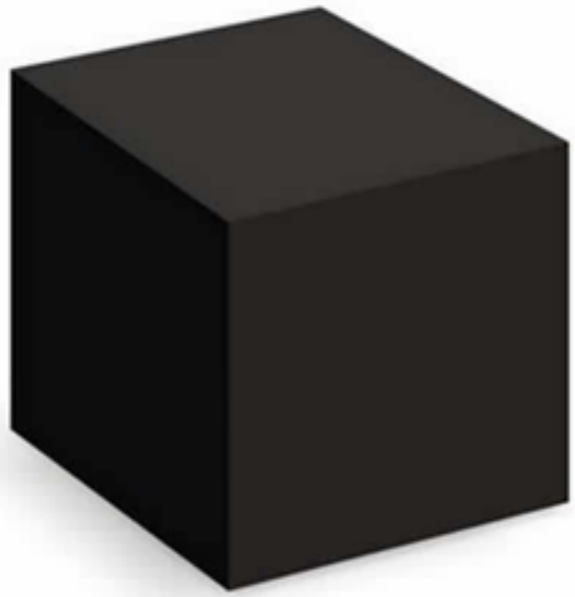
class Monad m => MonadProlog m where
  fail  :: m a
  (|||) :: m a -> m a -> m a
```

Full Solution

How to reconcile?

```
data Prolog a
   = Return a
   | Fail
   | Or (Prolog a)
        (Prolog a)
```

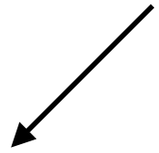




Abstract Syntax Tree

Opaque Monad

Interleaving Data & Effects

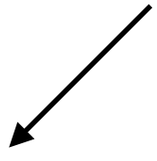


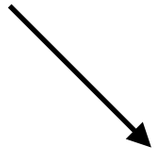


```
data OrF r = OrF r r
```

```
MonadProlog m
```

Interleaving Data & Effects





```
data OrF r = OrF r r
```

```
MonadProlog m
```

with

```
data FreeT f m a = FreeT (m (FreeT_ f m a))

data FreeT_ f m a
  = ReturnT a
  | OpT (f (FreeT f m a))
```

Free Monad Transformer

Interleaving Data & Effects



```
data OrF r = OrF r r
```

```
MonadProlog m
```

with

```
data FreeT f m a = FreeT (m (FreeT_ f m a))

data FreeT_ f m a
  = ReturnT a
  | OpT (f (FreeT f m a))
```

Free Monad Transformer

see also Patricia Johann's invited talk at Haskell 2014

Simple Solution

```
sem ::
  Prolog a -> [a]

type Heuristic a =
  Prolog a -> Prolog a

dbs ::
  Int -> Heuristic a

(\|/) ::
  Prolog a -> Prolog a -> Prolog a
```

Full Solution

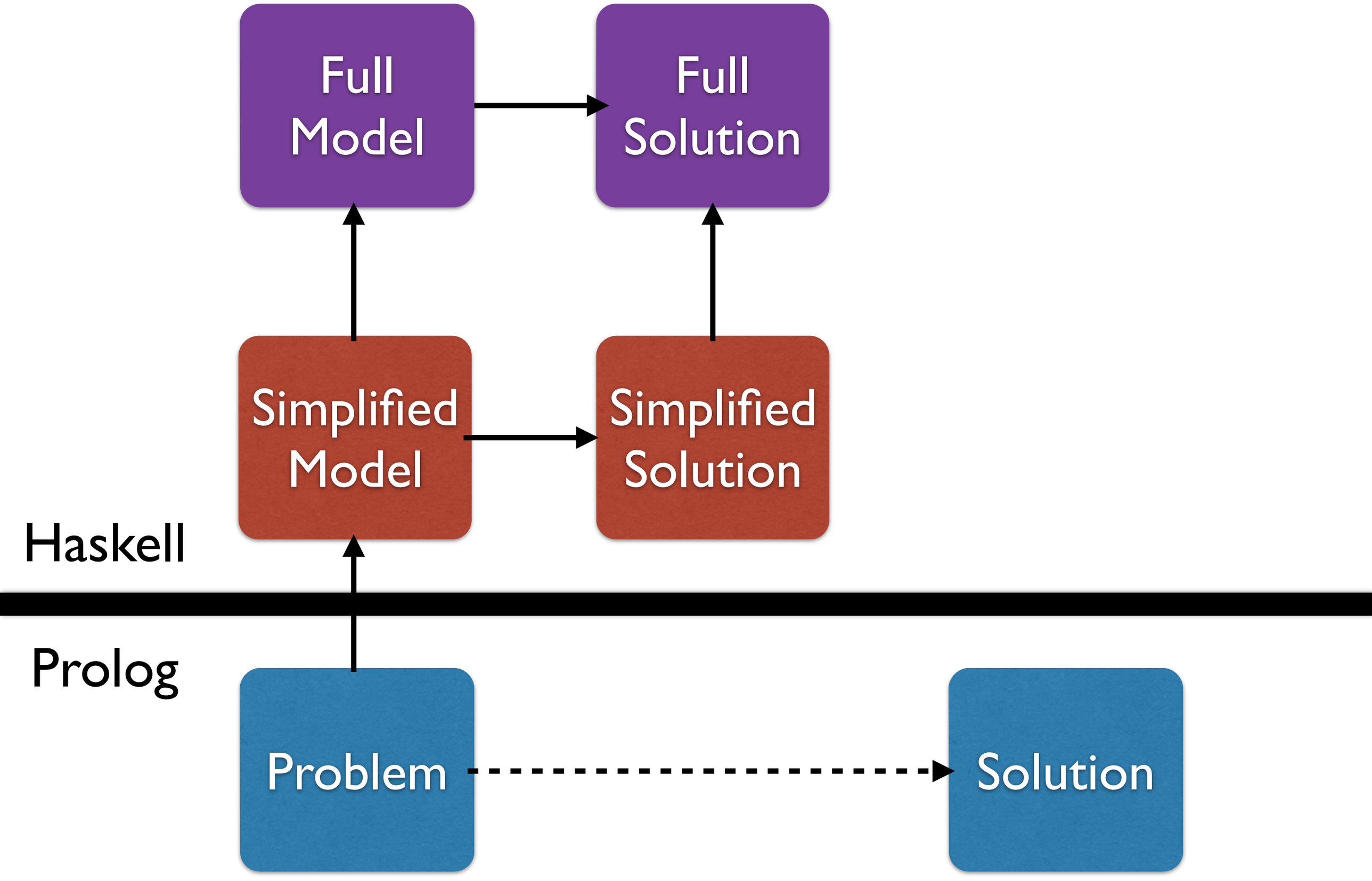
```
type Or*m a = FreeT OrF m a

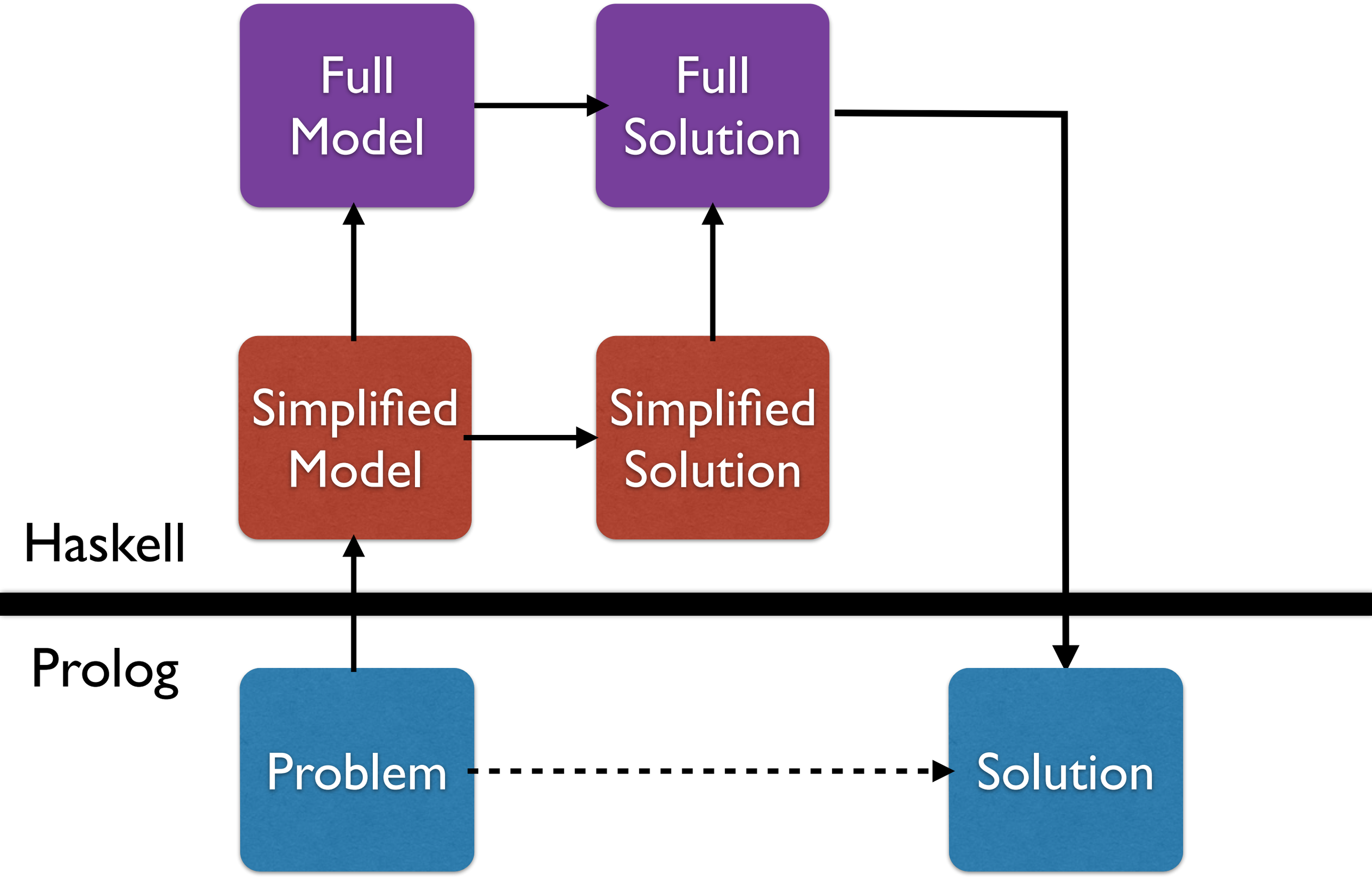
sem :: MonadProlog m =>
  Or*m a -> m a

type Heuristic m a =
  Or*m a -> Or*m a

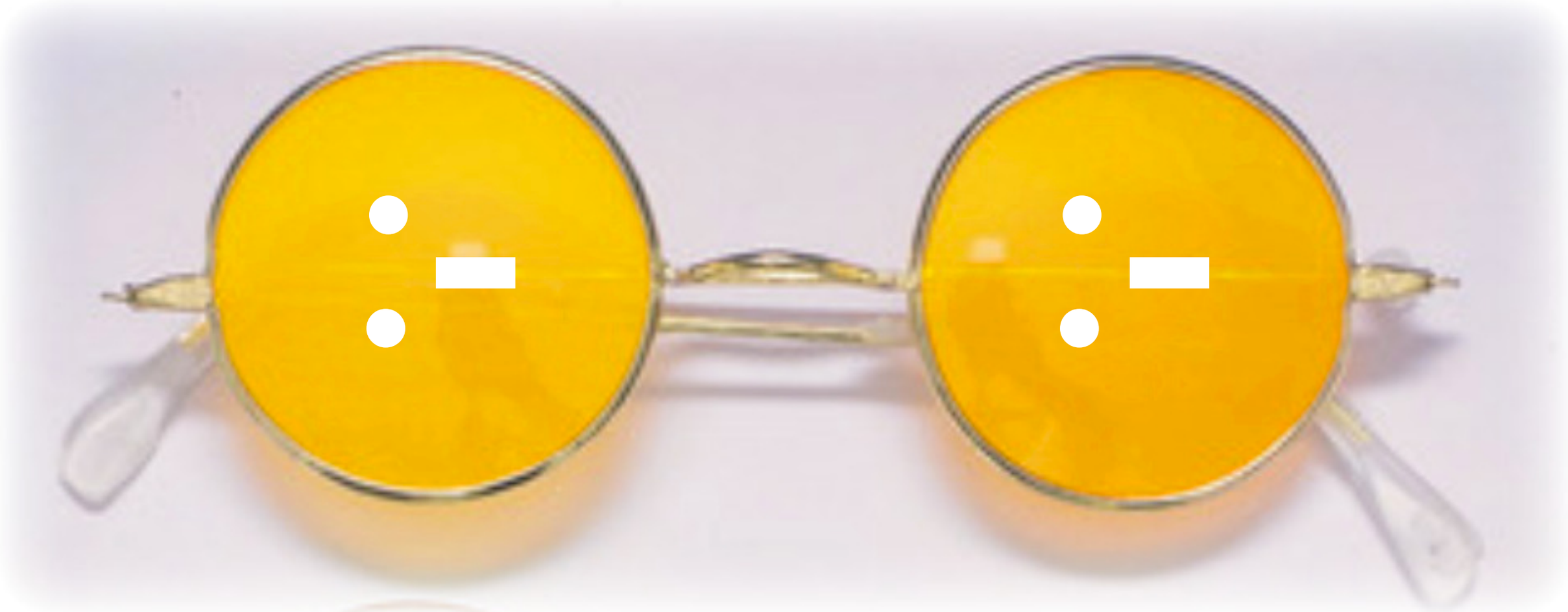
dbs :: MonadProlog m =>
  Int -> Heuristic m a

(\|/) :: MonadProlog m =>
  Or*m a -> Or*m a -> Or*m a
```


Full Model
Full Solution
Simplified Model
Simplified Solution
Haskell
Prolog
Problem
Solution


Full Model
Full Solution
Simplified Model
Simplified Solution
Haskell
Prolog
Problem
Solution

Prolog Solution

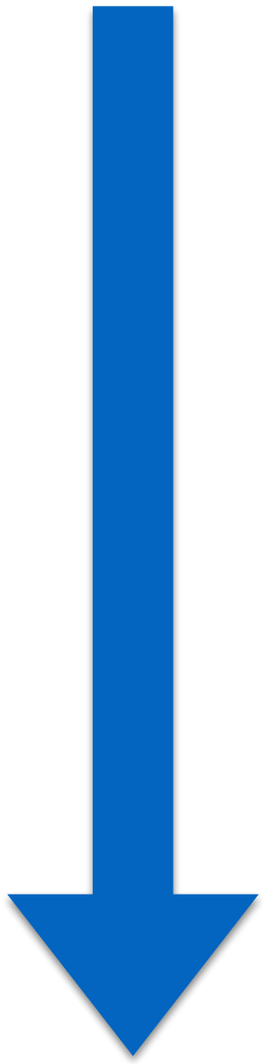


Free Monad Transformer



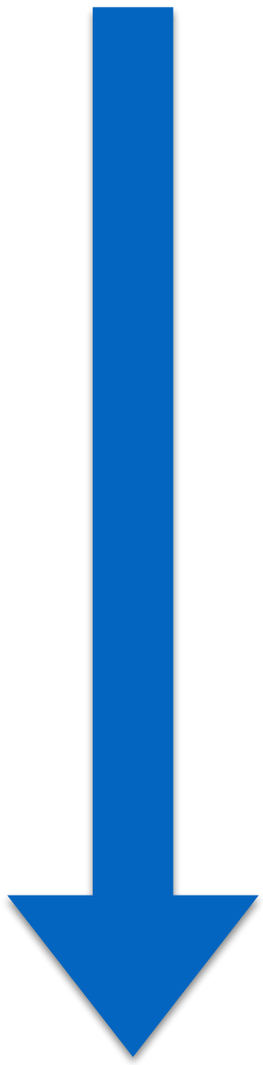
Prolog

Free Monad Transformer



Prolog Meta-Interpreter

Free Monad Transformer



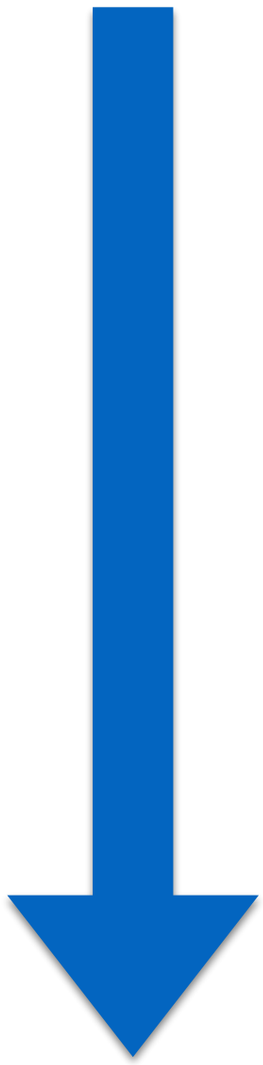
```
Or*Prolog ()
```

≅

```
eval(Goal,Flag)
```

Prolog Meta-Interpreter

Free Monad Transformer



$$\mathtt{Or}^{*}_{\mathtt{Prolog}}\ ()$$

$$\cong$$

```
eval(Goal,Flag)
```

```
return   or(LG,RG)
```

Prolog Meta-Interpreter

Meta-Interpreter Not Ideal

✗ too simple

ignoring lots of Prolog features

✗ too inefficient

replaces efficient WAM entirely

Free Monad Transformer



Prolog

Free Monad Transformer



Isomorphism
in the paper

Delimited Control

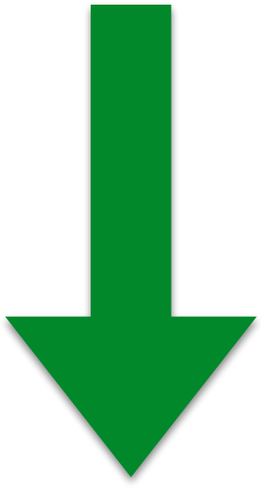
Prolog

Free Monad Transformer



Isomorphism
in the paper

Delimited Control



Delimited Continuations for Prolog
[T.S. et al., ICLP 2013]

Prolog Delimited Control

Solution in Prolog

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

?- sem(muddle(queens(8,Qs))).
```

✔ modular definition

✔ trivial composition

✔ firm semantic grounding

✔ rich expressivity

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

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

Wrapping Up

Summary

Summary

Clean semantics design
with algebraic effect handlers

Summary

Clean semantics design
with algebraic effect handlers
and delimited control in Prolog.

Summary

Clean semantics design
with algebraic effect handlers
and delimited control in Prolog.

Cross-fertilization pays off!

Thank You!