A Functional Perspective on Logic Programming

Alexander Vandenbroucke

KU LEUVEN

General Setting

LP

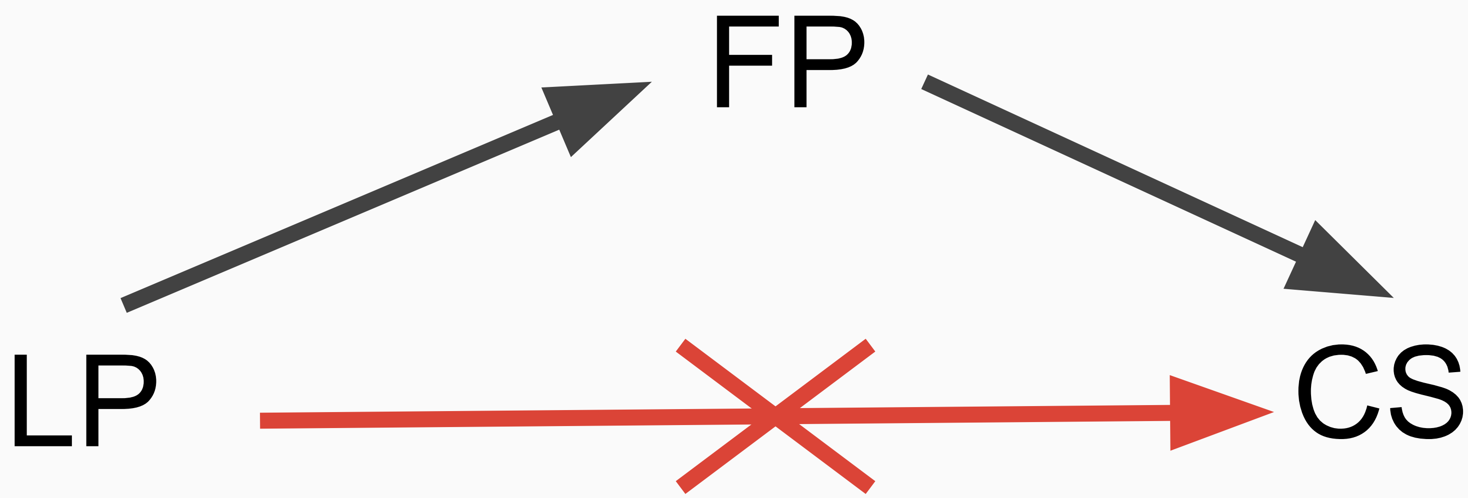
General Setting

$$\text{LP} \longrightarrow \text{CS}$$

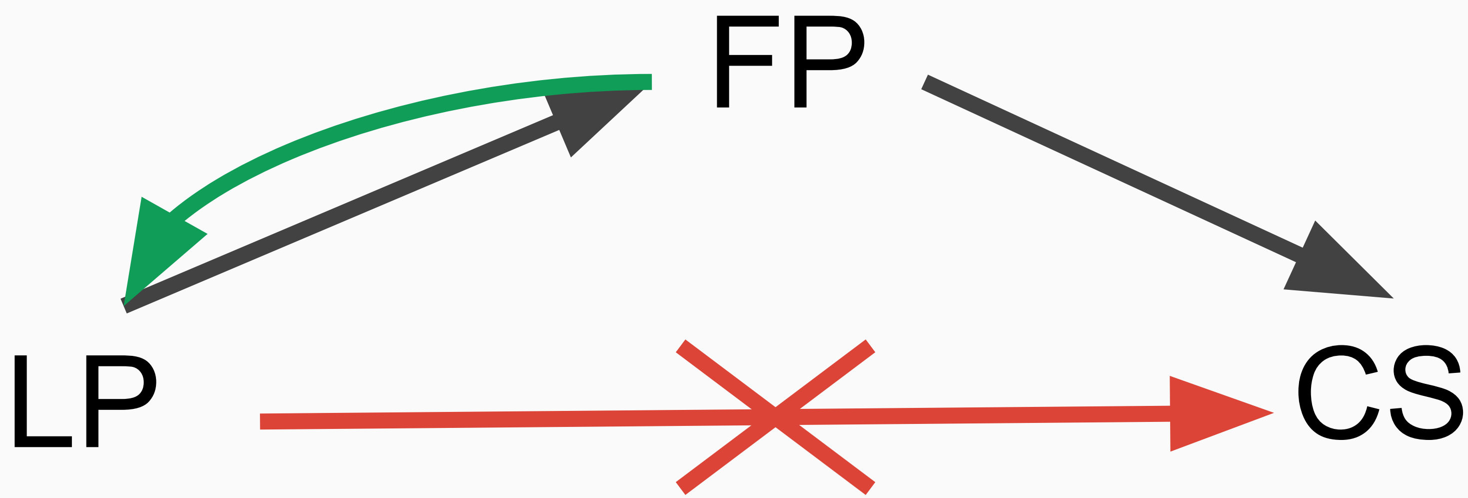
General Setting

$$\mathrm{LP} \not\longrightarrow \mathrm{CS}$$

General Setting



General Setting



Tabling as a Library with
Delimited Control
[Desouter et al. 2015]

SWI Prolog

Probabilistic Programming

Background

Probabilistic Models (AI)

Ad-Hoc ■—■ Ad-Hoc ■—■ Ad-Hoc

Background

Probabilistic Models (AI)

Ad-Hoc ■—■ Ad-Hoc ■—■ Ad-Hoc

Prevents effective reuse, communication, ….

Background

Probabilistic Models (AI)

Ad-Hoc Ad-Hoc Ad-Hoc

Prob. Prog
Language
ProbLog

ProbLog

probabilistic fact

```
0.5 :: heads1.
0.6 :: heads2.
```

https://dtai.cs.kuleuven.be/problog/

ProbLog

probabilistic fact

```
0.5 :: heads1.
0.6 :: heads2.
twoHeads :- heads1,heads2.
```

logic rule

https://dtai.cs.kuleuven.be/problog/

ProbLog

probabilistic fact

```
0.5 :: heads1.
0.6 :: heads2.
twoHeads :- heads1,heads2.
```

logic rule

queries

```
query(heads1).
query(twoHeads).
```

https://dtai.cs.kuleuven.be/problog/

ProbLog

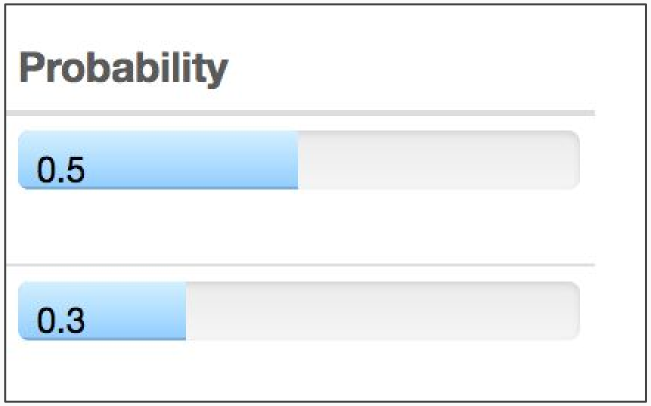
probabilistic fact

```
0.5 :: heads1.
0.6 :: heads2.
twoHeads :- heads1,heads2.
```

logic rule

queries

```
query(heads1).
query(twoHeads).
```



https://dtai.cs.kuleuven.be/problog/

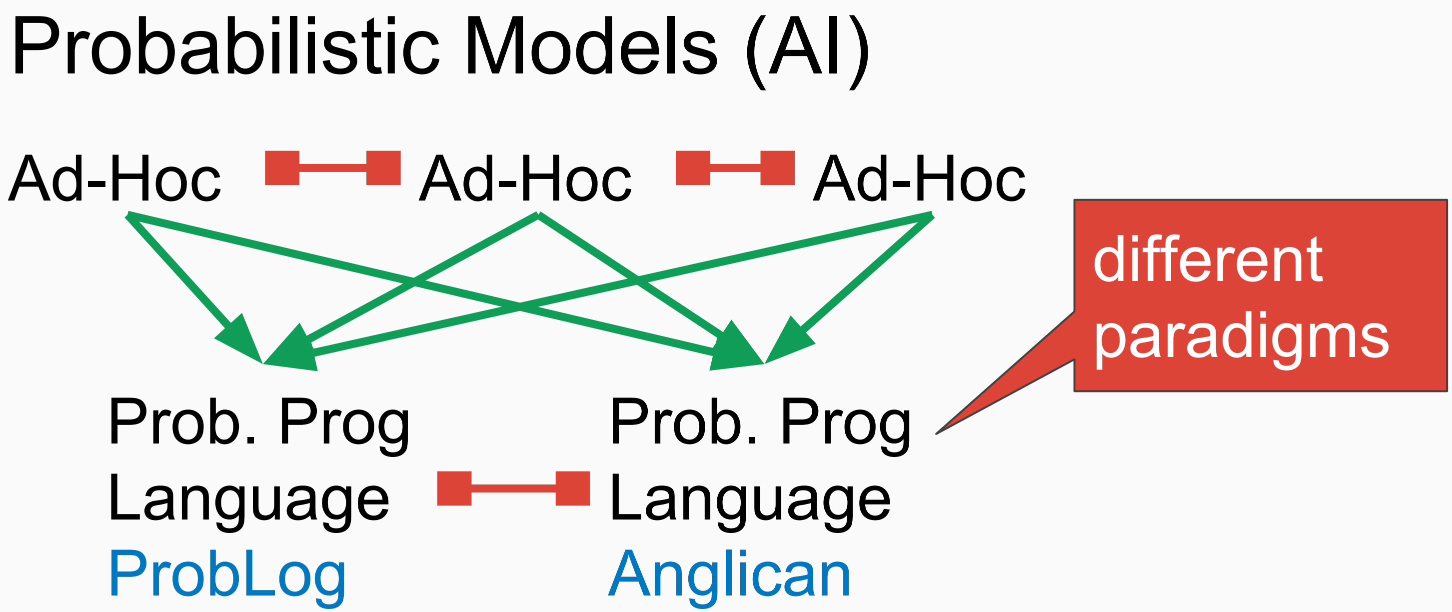
Background

Probabilistic Models (AI)

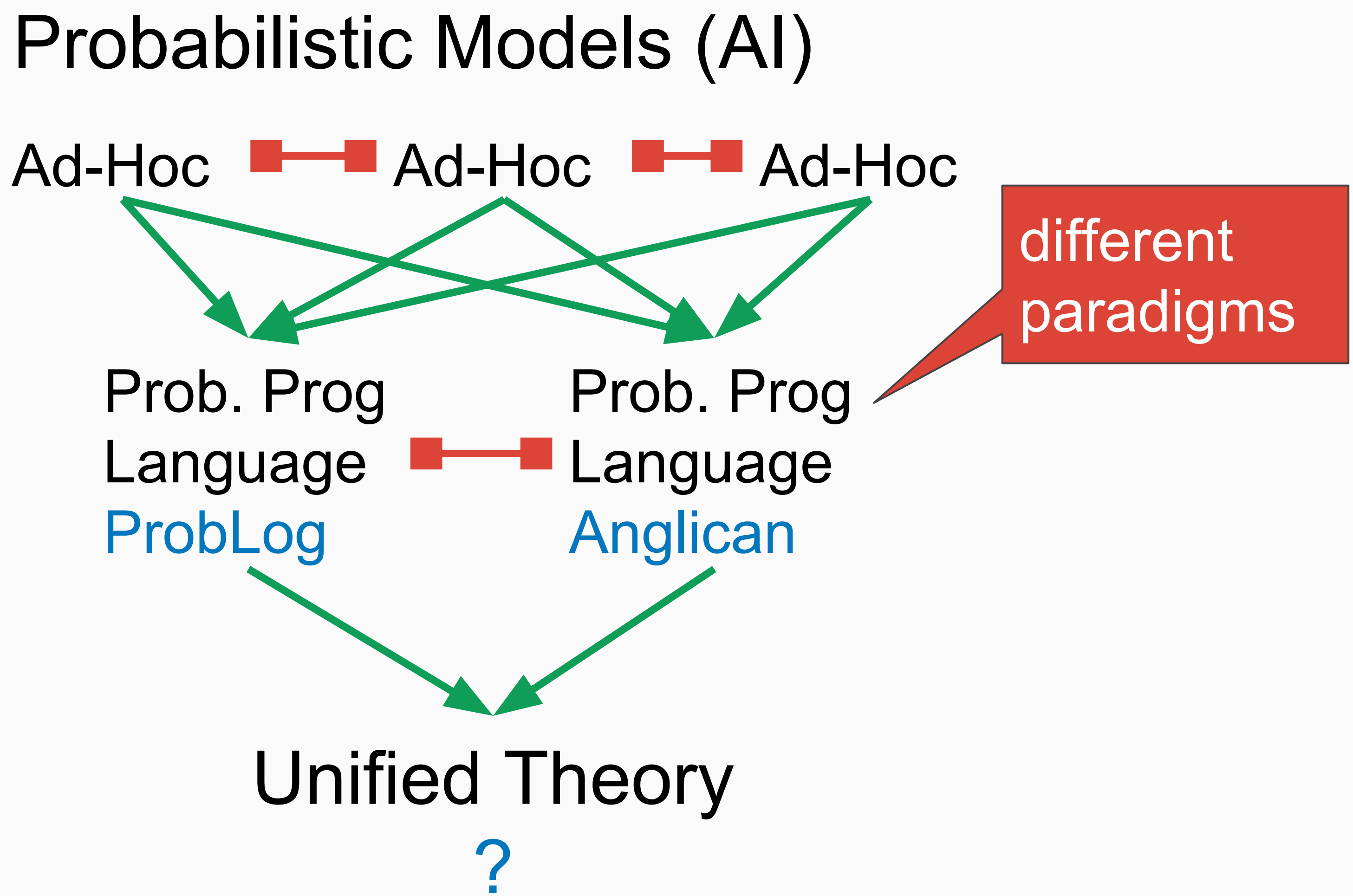
Ad-Hoc Ad-Hoc Ad-Hoc

Prob. Prog
Language
ProbLog

Background


Probabilistic Models (AI)
Ad-Hoc
Ad-Hoc
Ad-Hoc
Prob. Prog Language
ProbLog
Prob. Prog Language
Anglican
different paradigms

Background


Probabilistic Models (AI)
Ad-Hoc
Ad-Hoc
Ad-Hoc
different paradigms
Prob. Prog Language
ProbLog
Prob. Prog Language
Anglican
Unified Theory
?

Current Status

Anglican has dynamic structure

ProbLog has static structure

```
0.5 :: heads1.
0.6 :: heads2.
twoHeads :- heads1,heads2.
```

Does **not** influence the structure of

Current Status

ProbLog has static structure

Applicative functors:
~ programs with static structure

Current Status

ProbLog has static structure

Applicative functors:

faster inference

Current Status

Anglican has dynamic structure

Monads:

~ programs with dynamic structure

Current Status

Anglican: monadic

↑

ProbLog2: other structures

↑

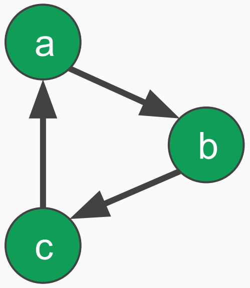
ProbLog: Applicative functors

Tabling and Answer Subsumption

Background - Regular Prolog

```
edge(a,b). edge(b,c). edge(c,a).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

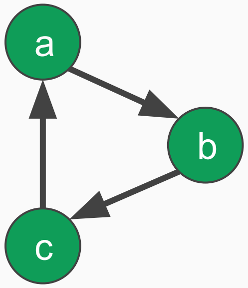


Background - Regular Prolog

```
edge(a,b). edge(b,c). edge(c,a).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

?- path(a,b).
true.

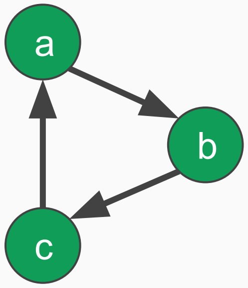


Regular Prolog

```
edge(a,b). edge(b,c). edge(c,a).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

?- path(a,b).
true;
true;
...



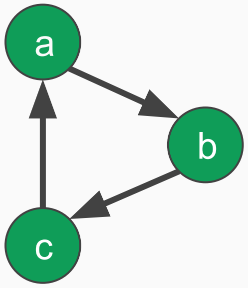
Regular Prolog

```
edge(a,b). edge(b,c). edge(c,a).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

?- path(a,d).

<infinite loop>



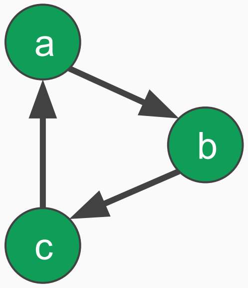
Tabled Prolog

```
:- table path/2.
edge(a,b). edge(b,c). edge(c,a).

path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z),path(Z,Y).
```

?- path(a,b).
true.

?- path(a,d).
false.



Tabled Prolog

YAP!

Applied Logic Systems

SWI Prolog

PROLOG B

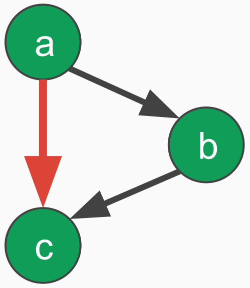
xsb

Regular Prolog

```
p(X,Y,1) :- e(X,Y).
p(X,Y,D) :- e(X,Z), p(Z,Y,D1),
            D is D1 + 1.
shortest(X,Y,MinD) :-
  findall(D,p(X,Y,D),List),
  min_list(List,MinD).
```

?- shortest(a,c,D).

D = 1.

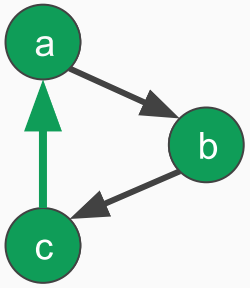


Regular Prolog

```
p(X,Y,1) :- e(X,Y).
p(X,Y,D) :- e(X,Z), p(Z,Y,D1),
            D is D1 + 1.
shortest(X,Y,MinD) :-
  findall(D,p(X,Y,D),List),
  min_list(List,MinD).
```

?- shortest(a,c,D).

<infinite loop>



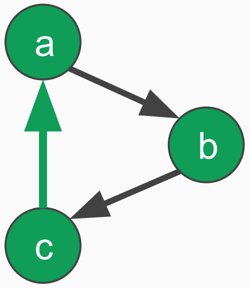
Answer Subsumption

```
:- table p(+,+,min).

p(X,Y,1) :- e(X,Y).
p(X,Y,D) :- e(X,Z), p(Z,Y,D1),
            D is D1 + 1.
```

?- p(a,c,D).

D = 2.



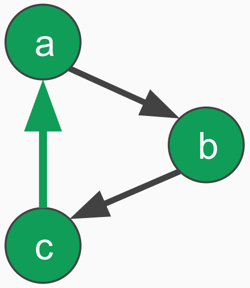
Answer Subsumption

```
:- table p(+,+,min).

p(X,Y,1) :- e(X,Y).
p(X,Y,D) :- e(X,Z), p(Z,Y,D1),
            D is D1 + 1.
```

?- p(a,c,D).
D = 2.

tabling modes



Current Status

Fixing Non-determinism (IFL 2015)

Fixing Non-determinism

Alexander Vandenbroucke   Tom Schrijvers   Frank Piessens

{alexander.vandenbroucke, tom.schrijvers, frank.piessens}@kuleuven.be
Department of Computer Science
KU Leuven

ABSTRACT

Non-deterministic computations are conventionally modelled by lists of their outcomes. This approach provides a concise declarative description of certain problems, as well as a way of generically solving such problems.

However, the traditional approach falls short when the non-deterministic problem is allowed to be recursive: the recursive problem may have infinitely many outcomes, giving rise to an infinite list.

Yet there are usually only finitely many distinct *relevant* results. This paper shows that this set of interesting results corresponds to a least fixed point. We provide an implementation based on algebraic effect handlers to compute such least fixed points in a finite amount of time, thereby allowing non-determinism and recursion to meaningfully co-occur in a single program.

CCS Concepts

•Software and its engineering → Functional languages; Recursion;

Keywords

Haskell, Tabling, Effect Handlers, Logic Programming, Non-determinism, Least Fixed Point

1. INTRODUCTION

Non-determinism [24] models a variety of problems in a declarative fashion, especially those problems where the solution depends on the exploration of different choices. The conventional approach represents non-determinism as lists of possible outcomes. For instance, consider the semantics of the following non-deterministic expression:

$$1 \mathbin{?} 2$$

This expression represents a non-deterministic choice (with the operator ?) between 1 and 2. Traditionally we model this with the list $[1, 2]$. Now consider the next example:

$$\begin{aligned} &\mathit{swap}\ (m, n) = (n, m) \\ &\mathit{pair} = (1, 2) \mathbin{?} \mathit{swap}\ \mathit{pair} \end{aligned}$$

 ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

The corresponding Haskell code is:

$$\begin{aligned} &\mathit{swap} :: [(a, b)] \rightarrow [(b, a)] \\ &\mathit{swap}\ e = [(m, n) \mid (n, m) \leftarrow e] \\ &\mathit{pair} :: [(\mathit{Int}, \mathit{Int})] \\ &\mathit{pair} = [(1, 2)] \mathbin{+\!\!+} \mathit{swap}\ \mathit{pair} \end{aligned}$$

This is an executable model (we use ⋙ to denote the prompt of the GHCi Haskell REPL):

$$\begin{aligned} &\ggg \mathit{pair} \\ &[(1, 2), (2, 1), (1, 2), (2, 1) \ldots \end{aligned}$$

We get an infinite list, although only two distinct outcomes ($(1, 2)$ and $(2, 1)$) exist. The conventional list-based approach is clearly inadequate in this example. In this paper we model non-determinism with sets of values instead, such that duplicates are implicitly removed. The expected model of *pair* is then the set $\{(1, 2), (2, 1)\}$. We can execute this model: [1]

$$\begin{aligned} &\mathit{swap} :: (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow \mathit{Set}\ (a, b) \rightarrow \mathit{Set}\ (b, a) \\ &\mathit{swap} = \mathit{map}\ (\lambda(m, n) \rightarrow (n, m)) \\ &\mathit{pair} :: \mathit{Set}\ (\mathit{Int}, \mathit{Int}) \\ &\mathit{pair} = \mathit{singleton}\ (1, 2)\ \text{`}\mathit{union}\text{`}\ \mathit{swap}\ \mathit{pair} \\ &\ggg \mathit{pair} \end{aligned}$$

```
fromList *** Exception:  <loop>
```

Haskell lazily prints the first part of the *Set* constructor, and then has to compute *union* infinitely many times. As an executable model of non-determinism it clearly remains inadequate: it fails to compute the solution $\{(1, 2), (2, 1)\}$.

This paper solves the problem caused by the co-occurence of non-determinism and recursion, by recasting it as the least fixed point problem of a *different* function. The least fixed point is computed explicitly by iteration, instead of implicitly by Haskell's recursive functions.

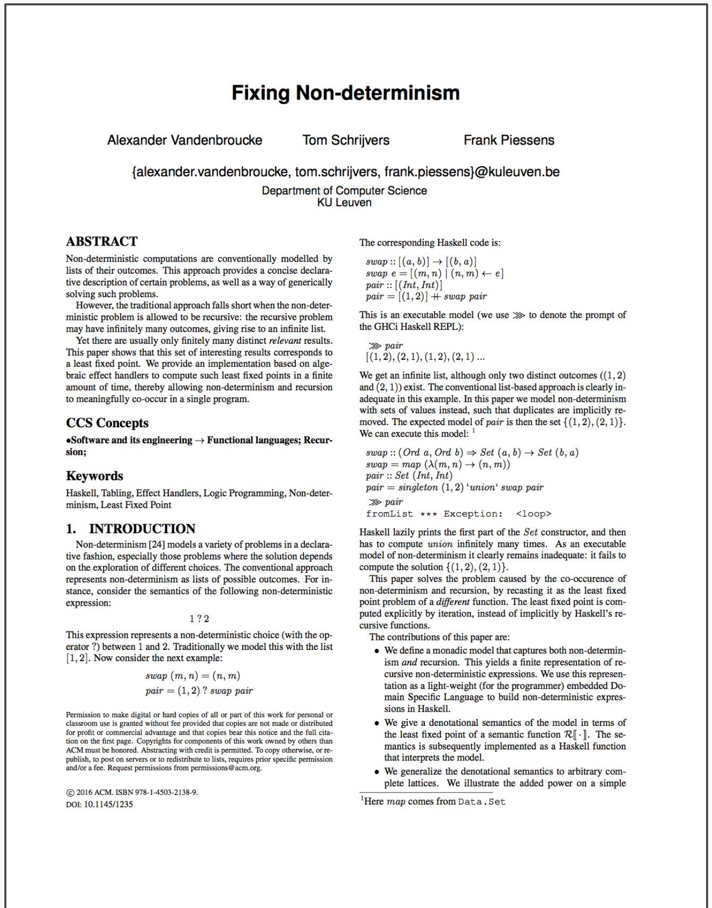
The contributions of this paper are:

- We define a monadic model that captures both non-determinism *and* recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalize the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple

[1] Here *map* comes from `Data.Set`

Current Status

Fixing Non-determinism (IFL 2015)



Fixing Non-determinism

Alexander Vandenbroucke   Tom Schrijvers   Frank Piessens

{alexander.vandenbroucke, tom.schrijvers, frank.piessens}@kuleuven.be
Department of Computer Science
KU Leuven

ABSTRACT

Non-deterministic computations are conventionally modelled by lists of their outcomes. This approach provides a concise declarative description of certain problems, as well as a way of generically solving such problems.

However, the traditional approach falls short when the non-deterministic problem is allowed to be recursive: the recursive problem may have infinitely many outcomes, giving rise to an infinite list.

Yet there are usually only finitely many distinct *relevant* results. This paper shows that this set of interesting results corresponds to a least fixed point. We provide an implementation based on algebraic effect handlers to compute such least fixed points in a finite amount of time, thereby allowing non-determinism and recursion to meaningfully co-occur in a single program.

CCS Concepts

•**Software and its engineering → Functional languages; Recursion;**

Keywords

Haskell, Tabling, Effect Handlers, Logic Programming, Non-determinism, Least Fixed Point

1. INTRODUCTION

Non-determinism [24] models a variety of problems in a declarative fashion, especially those problems where the solution depends on the exploration of different choices. The conventional approach represents non-determinism as lists of possible outcomes. For instance, consider the semantics of the following non-deterministic expression:

$$1 \mathbin{?} 2$$

This expression represents a non-deterministic choice (with the operator ?) between 1 and 2. Traditionally we model this with the list $[1, 2]$. Now consider the next example:

$$\begin{aligned} &swap\ (m, n) = (n, m) \\ &pair = (1, 2) \mathbin{?} swap\ pair \end{aligned}$$

DOI: 10.1145/1235

The corresponding Haskell code is:

$$\begin{aligned} &swap :: [(a, b)] \to [(b, a)] \\ &swap\ e = [(m, n) \mid (n, m) \leftarrow e] \\ &pair :: [(Int, Int)] \\ &pair = [(1, 2)] \mathbin{+\!\!+} swap\ pair \end{aligned}$$

This is an executable model (we use ⋙ to denote the prompt of the GHCi Haskell REPL):

$\ggg pair$
$[(1, 2), (2, 1), (1, 2), (2, 1) \ldots$

We get an infinite list, although only two distinct outcomes ($(1, 2)$ and $(2, 1)$) exist. The conventional list-based approach is clearly inadequate in this example. In this paper we model non-determinism with sets of values instead, such that duplicates are implicitly removed. The expected model of *pair* is then the set $\{(1, 2), (2, 1)\}$. We can execute this model: [1]

$$\begin{aligned} &swap :: (Ord\ a, Ord\ b) \Rightarrow Set\ (a, b) \to Set\ (b, a) \\ &swap = map\ (\lambda(m, n) \to (n, m)) \\ &pair :: Set\ (Int, Int) \\ &pair = singleton\ (1, 2)\ \text{`}union\text{`}\ swap\ pair \end{aligned}$$

$\ggg pair$
```
fromList *** Exception:  <loop>
```

Haskell lazily prints the first part of the *Set* constructor, and then has to compute *union* infinitely many times. As an executable model of non-determinism it clearly remains inadequate: it fails to compute the solution $\{(1, 2), (2, 1)\}$.

This paper solves the problem caused by the co-occurence of non-determinism and recursion, by recasting it as the least fixed point problem of a *different* function. The least fixed point is computed explicitly by iteration, instead of implicitly by Haskell's recursive functions.

The contributions of this paper are:

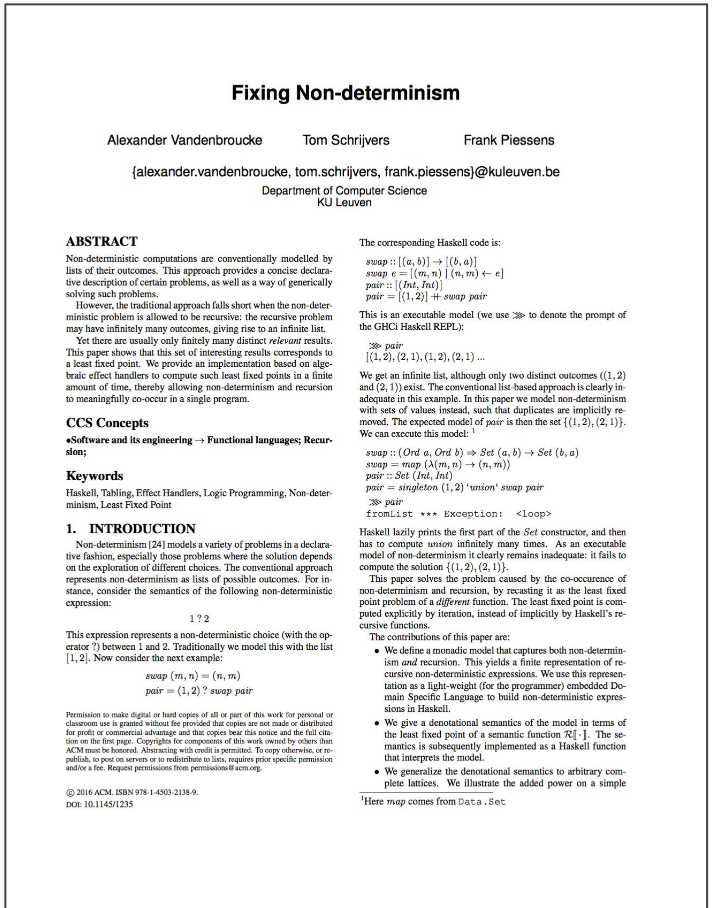
- We define a monadic model that captures both non-determinism *and* recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalize the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple

[1] Here *map* comes from `Data.Set`

➔ when non-determinism and recursion co-occur

Current Status

Fixing Non-determinism (IFL 2015)



Fixing Non-determinism

Alexander Vandenbroucke   Tom Schrijvers   Frank Piessens

{alexander.vandenbroucke, tom.schrijvers, frank.piessens}@kuleuven.be

Department of Computer Science
KU Leuven

ABSTRACT

Non-deterministic computations are conventionally modelled by lists of their outcomes. This approach provides a concise declarative description of certain problems, as well as a way of generically solving such problems.

However, the traditional approach falls short when the non-deterministic problem is allowed to be recursive: the recursive problem may have infinitely many outcomes, giving rise to an infinite list.

Yet there are usually only finitely many distinct *relevant* results. This paper shows that this set of interesting results corresponds to a least fixed point. We provide an implementation based on algebraic effect handlers to compute such least fixed points in a finite amount of time, thereby allowing non-determinism and recursion to meaningfully co-occur in a single program.

CCS Concepts

•Software and its engineering → Functional languages; Recursion;

Keywords

Haskell, Tabling, Effect Handlers, Logic Programming, Non-determinism, Least Fixed Point

1. INTRODUCTION

Non-determinism [24] models a variety of problems in a declarative fashion, especially those problems where the solution depends on the exploration of different choices. The conventional approach represents non-determinism as lists of possible outcomes. For instance, consider the semantics of the following non-deterministic expression:

$$1 \mathbin{?} 2$$

This expression represents a non-deterministic choice (with the operator ?) between 1 and 2. Traditionally we model this with the list $[1, 2]$. Now consider the next example:

$$\begin{aligned} &\mathit{swap}\ (m, n) = (n, m) \\ &\mathit{pair} = (1, 2) \mathbin{?} \mathit{swap}\ \mathit{pair} \end{aligned}$$

 ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

The corresponding Haskell code is:

$$\begin{aligned} &\mathit{swap} :: [(a, b)] \to [(b, a)] \\ &\mathit{swap}\ e = [(m, n) \mid (n, m) \leftarrow e] \\ &\mathit{pair} :: [(\mathit{Int}, \mathit{Int})] \\ &\mathit{pair} = [(1, 2)] +\!\!+\ \mathit{swap}\ \mathit{pair} \end{aligned}$$

This is an executable model (we use $\ggg$ to denote the prompt of the GHCi Haskell REPL):

$$\begin{aligned} &\ggg \mathit{pair} \\ &[(1, 2), (2, 1), (1, 2), (2, 1) \ldots \end{aligned}$$

We get an infinite list, although only two distinct outcomes ($(1, 2)$ and $(2, 1)$) exist. The conventional list-based approach is clearly inadequate in this example. In this paper we model non-determinism with sets of values instead, such that duplicates are implicitly removed. The expected model of *pair* is then the set $\{(1, 2), (2, 1)\}$. We can execute this model: [1]

$$\begin{aligned} &\mathit{swap} :: (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow \mathit{Set}\ (a, b) \to \mathit{Set}\ (b, a) \\ &\mathit{swap} = \mathit{map}\ (\lambda(m, n) \to (n, m)) \\ &\mathit{pair} :: \mathit{Set}\ (\mathit{Int}, \mathit{Int}) \\ &\mathit{pair} = \mathit{singleton}\ (1, 2)\ \text{`}\mathit{union}\text{`}\ \mathit{swap}\ \mathit{pair} \\ &\ggg \mathit{pair} \end{aligned}$$

```
fromList *** Exception:  <loop>
```

Haskell lazily prints the first part of the *Set* constructor, and then has to compute *union* infinitely many times. As an executable model of non-determinism it clearly remains inadequate: it fails to compute the solution $\{(1, 2), (2, 1)\}$.

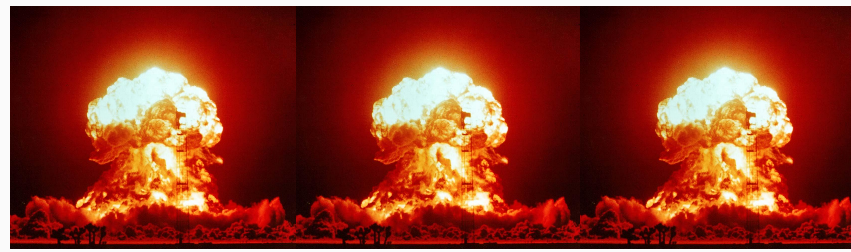
This paper solves the problem caused by the co-occurence of non-determinism and recursion, by recasting it as the least fixed point problem of a *different* function. The least fixed point is computed explicitly by iteration, instead of implicitly by Haskell's recursive functions.

The contributions of this paper are:

- We define a monadic model that captures both non-determinism *and* recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalize the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple

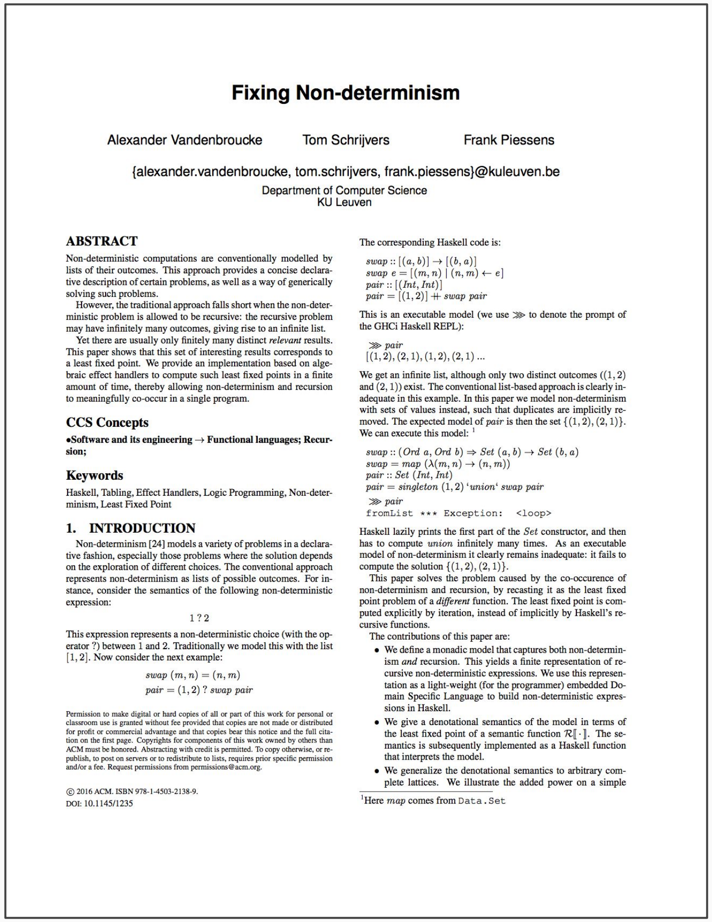
[1] Here *map* comes from `Data.Set`

➔ when non-determinism and recursion co-occur



Current Status

Fixing Non-determinism (IFL 2015)



Fixing Non-determinism

Alexander Vandenbroucke Tom Schrijvers Frank Piessens

{alexander.vandenbroucke, tom.schrijvers, frank.piessens}@kuleuven.be
Department of Computer Science
KU Leuven

ABSTRACT

Non-deterministic computations are conventionally modelled by lists of their outcomes. This approach provides a concise declarative description of certain problems, as well as a way of generically solving such problems.

However, the traditional approach falls short when the non-deterministic problem is allowed to be recursive: the recursive problem may have infinitely many outcomes, giving rise to an infinite list.

Yet there are usually only finitely many distinct *relevant* results. This paper shows that this set of interesting results corresponds to a least fixed point. We provide an implementation based on algebraic effect handlers to compute such least fixed points in a finite amount of time, thereby allowing non-determinism and recursion to meaningfully co-occur in a single program.

CCS Concepts

•Software and its engineering → Functional languages; Recursion;

Keywords

Haskell, Tabling, Effect Handlers, Logic Programming, Non-determinism, Least Fixed Point

1. INTRODUCTION

Non-determinism [24] models a variety of problems in a declarative fashion, especially those problems where the solution depends on the exploration of different choices. The conventional approach represents non-determinism as lists of possible outcomes. For instance, consider the semantics of the following non-deterministic expression:

$$1 \mathbin{?} 2$$

This expression represents a non-deterministic choice (with the operator ?) between 1 and 2. Traditionally we model this with the list $[1, 2]$. Now consider the next example:

$$\mathit{swap}\ (m, n) = (n, m)$$
$$\mathit{pair} = (1, 2) \mathbin{?} \mathit{swap\ pair}$$

DOI: 10.1145/1235

The corresponding Haskell code is:

```
swap :: [(a, b)] → [(b, a)]
swap e = [(m, n) | (n, m) ← e]
pair :: [(Int, Int)]
pair = [(1, 2)] ++ swap pair
```

This is an executable model (we use ⋙ to denote the prompt of the GHCi Haskell REPL):

```
⋙ pair
[(1, 2), (2, 1), (1, 2), (2, 1) ...
```

We get an infinite list, although only two distinct outcomes ($(1, 2)$ and $(2, 1)$) exist. The conventional list-based approach is clearly inadequate in this example. In this paper we model non-determinism with sets of values instead, such that duplicates are implicitly removed. The expected model of *pair* is then the set $\{(1, 2), (2, 1)\}$. We can execute this model: [1]

```
swap :: (Ord a, Ord b) ⇒ Set (a, b) → Set (b, a)
swap = map (λ(m, n) → (n, m))
pair :: Set (Int, Int)
pair = singleton (1, 2) `union` swap pair
⋙ pair
fromList *** Exception:  <loop>
```

Haskell lazily prints the first part of the *Set* constructor, and then has to compute *union* infinitely many times. As an executable model of non-determinism it clearly remains inadequate: it fails to compute the solution $\{(1, 2), (2, 1)\}$.

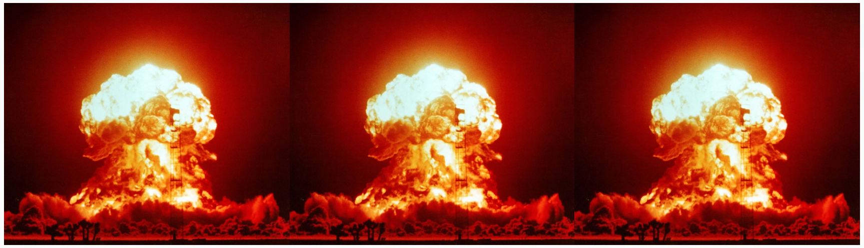
This paper solves the problem caused by the co-occurence of non-determinism and recursion, by recasting it as the least fixed point problem of a *different* function. The least fixed point is computed explicitly by iteration, instead of implicitly by Haskell's recursive functions.

The contributions of this paper are:

- We define a monadic model that captures both non-determinism *and* recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalize the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple

[1] Here *map* comes from `Data.Set`

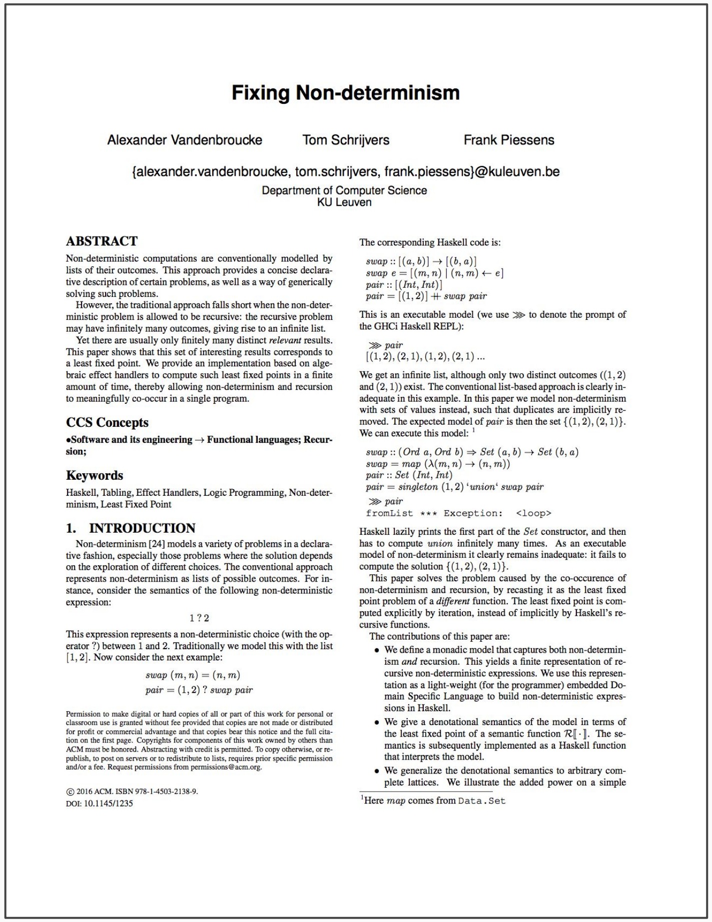
- ➔ when non-determinism and recursion co-occur



- ➔ essence of tabling

Current Status

Fixing Non-determinism (IFL 2015)



Fixing Non-determinism

Alexander Vandenbroucke   Tom Schrijvers   Frank Piessens

{alexander.vandenbroucke, tom.schrijvers, frank.piessens}@kuleuven.be
Department of Computer Science
KU Leuven

ABSTRACT

Non-deterministic computations are conventionally modelled by lists of their outcomes. This approach provides a concise declarative description of certain problems, as well as a way of generically solving such problems.

However, the traditional approach falls short when the non-deterministic problem is allowed to be recursive: the recursive problem may have infinitely many outcomes, giving rise to an infinite list.

Yet there are usually only finitely many distinct *relevant* results. This paper shows that this set of interesting results corresponds to a least fixed point. We provide an implementation based on algebraic effect handlers to compute such least fixed points in a finite amount of time, thereby allowing non-determinism and recursion to meaningfully co-occur in a single program.

CCS Concepts

•Software and its engineering → Functional languages; Recursion;

Keywords

Haskell, Tabling, Effect Handlers, Logic Programming, Non-determinism, Least Fixed Point

1. INTRODUCTION

Non-determinism [24] models a variety of problems in a declarative fashion, especially those problems where the solution depends on the exploration of different choices. The conventional approach represents non-determinism as lists of possible outcomes. For instance, consider the semantics of the following non-deterministic expression:

$$1 \mathbin{?} 2$$

This expression represents a non-deterministic choice (with the operator ?) between 1 and 2. Traditionally we model this with the list $[1, 2]$. Now consider the next example:

$$\begin{aligned} &\mathit{swap}\ (m, n) = (n, m)\\ &\mathit{pair} = (1, 2) \mathbin{?} \mathit{swap}\ \mathit{pair}\end{aligned}$$

 ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

The corresponding Haskell code is:

$$\begin{aligned} &\mathit{swap} :: [(a, b)] \to [(b, a)]\\ &\mathit{swap}\ e = [(m, n) \mid (n, m) \leftarrow e]\\ &\mathit{pair} :: [(\mathit{Int}, \mathit{Int})]\\ &\mathit{pair} = [(1, 2)] +\!\!+ \mathit{swap}\ \mathit{pair}\end{aligned}$$

This is an executable model (we use ⋙ to denote the prompt of the GHCi Haskell REPL):

$$\begin{aligned} &\ggg \mathit{pair}\\ &[(1, 2), (2, 1), (1, 2), (2, 1) \ldots\end{aligned}$$

We get an infinite list, although only two distinct outcomes ($(1, 2)$ and $(2, 1)$) exist. The conventional list-based approach is clearly inadequate in this example. In this paper we model non-determinism with sets of values instead, such that duplicates are implicitly removed. The expected model of *pair* is then the set $\{(1, 2), (2, 1)\}$. We can execute this model: [1]

$$\begin{aligned} &\mathit{swap} :: (\mathit{Ord}\ a, \mathit{Ord}\ b) \Rightarrow \mathit{Set}\ (a, b) \to \mathit{Set}\ (b, a)\\ &\mathit{swap} = \mathit{map}\ (\lambda(m, n) \to (n, m))\\ &\mathit{pair} :: \mathit{Set}\ (\mathit{Int}, \mathit{Int})\\ &\mathit{pair} = \mathit{singleton}\ (1, 2)\ \text{`}\mathit{union}\text{`}\ \mathit{swap}\ \mathit{pair}\\ &\ggg \mathit{pair}\end{aligned}$$

```
fromList *** Exception:  <loop>
```

Haskell lazily prints the first part of the *Set* constructor, and then has to compute *union* infinitely many times. As an executable model of non-determinism it clearly remains inadequate: it fails to compute the solution $\{(1, 2), (2, 1)\}$.

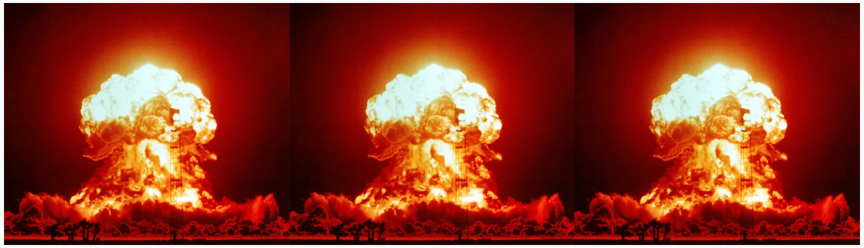
This paper solves the problem caused by the co-occurence of non-determinism and recursion, by recasting it as the least fixed point problem of a *different* function. The least fixed point is computed explicitly by iteration, instead of implicitly by Haskell's recursive functions.

The contributions of this paper are:

- We define a monadic model that captures both non-determinism *and* recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalize the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple

[1] Here *map* comes from `Data.Set`

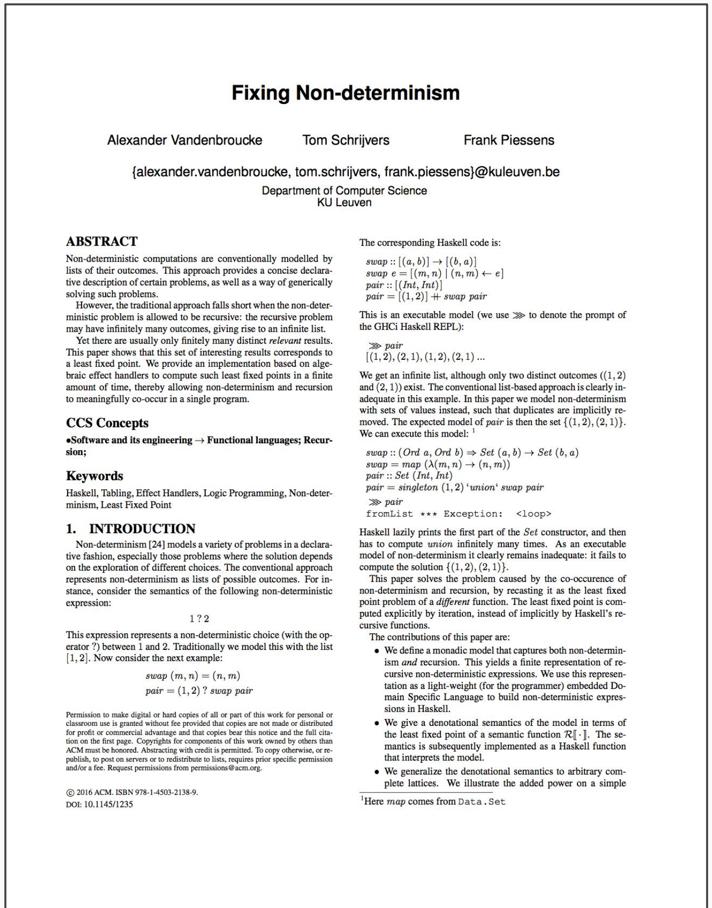
- ➔ when non-determinism and recursion co-occur



- ➔ essence of tabling
- ➔ implementation in Haskell

Current Status

Fixing Non-determinism (IFL 2015)



Fixing Non-determinism

Alexander Vandenbroucke Tom Schrijvers Frank Piessens

{alexander.vandenbroucke, tom.schrijvers, frank.piessens}@kuleuven.be
Department of Computer Science
KU Leuven

ABSTRACT

Non-deterministic computations are conventionally modelled by lists of their outcomes. This approach provides a concise declarative description of certain problems, as well as a way of generically solving such problems.

However, the traditional approach falls short when the non-deterministic problem is allowed to be recursive: the recursive problem may have infinitely many outcomes, giving rise to an infinite list.

Yet there are usually only finitely many distinct *relevant* results. This paper shows that this set of interesting results corresponds to a least fixed point. We provide an implementation based on algebraic effect handlers to compute such least fixed points in a finite amount of time, thereby allowing non-determinism and recursion to meaningfully co-occur in a single program.

CCS Concepts

•Software and its engineering → Functional languages; Recursion;

Keywords

Haskell, Tabling, Effect Handlers, Logic Programming, Non-determinism, Least Fixed Point

1. INTRODUCTION

Non-determinism [24] models a variety of problems in a declarative fashion, especially those problems where the solution depends on the exploration of different choices. The conventional approach represents non-determinism as lists of possible outcomes. For instance, consider the semantics of the following non-deterministic expression:

$$1 \mathbin{?} 2$$

This expression represents a non-deterministic choice (with the operator ?) between 1 and 2. Traditionally we model this with the list $[1, 2]$. Now consider the next example:

$$\begin{aligned} &swap\ (m, n) = (n, m) \\ &pair = (1, 2) \mathbin{?} swap\ pair \end{aligned}$$

DOI: 10.1145/1235

The corresponding Haskell code is:

$$\begin{aligned} &swap :: [(a, b)] \rightarrow [(b, a)] \\ &swap\ e = [(m, n) \mid (n, m) \leftarrow e] \\ &pair :: [(Int, Int)] \\ &pair = [(1, 2)] \mathbin{+\!\!+} swap\ pair \end{aligned}$$

This is an executable model (we use ⋙ to denote the prompt of the GHCi Haskell REPL):

$\ggg pair$
$[(1, 2), (2, 1), (1, 2), (2, 1) \ldots$

We get an infinite list, although only two distinct outcomes ($(1, 2)$ and $(2, 1)$) exist. The conventional list-based approach is clearly inadequate in this example. In this paper we model non-determinism with sets of values instead, such that duplicates are implicitly removed. The expected model of *pair* is then the set $\{(1, 2), (2, 1)\}$. We can execute this model: [1]

$$\begin{aligned} &swap :: (Ord\ a, Ord\ b) \Rightarrow Set\ (a, b) \rightarrow Set\ (b, a) \\ &swap = map\ (\lambda(m, n) \rightarrow (n, m)) \\ &pair :: Set\ (Int, Int) \\ &pair = singleton\ (1, 2)\ \text{`}union\text{`}\ swap\ pair \end{aligned}$$

$\ggg pair$

```
fromList *** Exception:  <loop>
```

Haskell lazily prints the first part of the *Set* constructor, and then has to compute *union* infinitely many times. As an executable model of non-determinism it clearly remains inadequate: it fails to compute the solution $\{(1, 2), (2, 1)\}$.

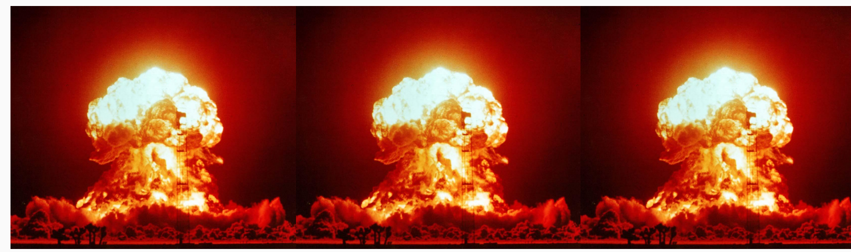
This paper solves the problem caused by the co-occurence of non-determinism and recursion, by recasting it as the least fixed point problem of a *different* function. The least fixed point is computed explicitly by iteration, instead of implicitly by Haskell's recursive functions.

The contributions of this paper are:

- We define a monadic model that captures both non-determinism *and* recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalize the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple

[1] Here *map* comes from `Data.Set`

- ➔ when non-determinism and recursion co-occur



- ➔ essence of tabling
- ➔ implementation in Haskell
- ➔ fixpoint (functional) semantics

Answer Subsumption - Example 2

```
p(0).
p(1).
p(2) :- p(X), X = 1.
p(3) :- p(X), X = 0.
```

?- p(X).
X = 0;
X = 1;
X = 2;
X = 3.

Answer Subsumption - Example 2

```
:- table p(max).
p(0).
p(1).
p(2) :- p(X), X = 1.
p(3) :- p(X), X = 0.
```

?- p(X).

Answer Subsumption - Example 2

```
:- table p(max).
p(0).
p(1).
p(2) :- p(X), X = 1.
p(3) :- p(X), X = 0.
```

?- p(X).

X = 0;

X = 1;

X = 2.

Answer Subsumption - Example 2

```
:- table p(max).
p(0).
p(1).
p(2) :- p(X), X = 1.
p(3) :- p(X), X = 0.
```

?- p(X).

X = 0;

X = 1;

X = 2.

1 overwrites 0

the last rule does not apply!

Current Status

Tabling with Sound Answer Subsumption

Under consideration for publication in Theory and Practice of Logic Programming 1

Tabling with Sound Answer Subsumption

Alexander Vandenbroucke
KU Leuven, Belgium
(*e-mail:* alexander.vandenbroucke@kuleuven.be)

Maciej Piróg
KU Leuven, Belgium
(*e-mail:* maciej.pirog@kuleuven.be)

Benoit Desouter
Ghent University, Belgium
(*e-mail:* benoit.desouter@ugent.be)

Tom Schrijvers
KU Leuven, Belgium
(*e-mail:* tom.schrijvers@kuleuven.be)

submitted 29 April 2016; revised 8 July 2016; accepted 22 July 2016

Abstract

Tabling is a powerful resolution mechanism for logic programs that captures their least fixed point semantics more faithfully than plain Prolog. In many tabling applications, we are not interested in the set of all answers to a goal, but only require an aggregation of those answers. Several works have studied efficient techniques, such as lattice-based answer subsumption and mode-directed tabling, to do so for various forms of aggregation.

While much attention has been paid to expressivity and efficient implementation of the different approaches, soundness has not been considered. This paper shows that the different implementations indeed fail to produce least fixed points for some programs. As a remedy, we provide a formal framework that generalises the existing approaches and we establish a soundness criterion that explains for which programs the approach is sound.

KEYWORDS: tabling, answer subsumption, lattice, partial order, mode-directed tabling, denotational semantics, Prolog

1 Introduction

Tabling considerably improves the declarativity and expressiveness of the Prolog language. It removes the sensitivity of SLD resolution to rule and goal ordering, allowing a larger class of programs to terminate. As an added bonus, the memoisation of the tabling mechanism may significantly improve run time performance in exchange for increased memory usage. Tabling has been implemented in a few well-known Prolog systems, such as XSB (Swift and Warren 2010; Swift and Warren 2012),Yap (Santos Costa et al. 2012), Ciao (Chico de Guzmán et al. 2008) and B-Prolog (Zhou 2012), and has been successfully applied in various domains.

Much research effort has been devoted to improving the performance of tabling for

Current Status

Tabling with Sound Answer Subsumption


Under consideration for publication in Theory and Practice of Logic Programming 1

Tabling with Sound Answer Subsumption

Alexander Vandenbroucke
KU Leuven, Belgium
(*e-mail:* alexander.vandenbroucke@kuleuven.be)

Maciej Piróg
KU Leuven, Belgium
(*e-mail:* maciej.pirog@kuleuven.be)

Benoit Desouter
Ghent University, Belgium
(*e-mail:* benoit.desouter@ugent.be)

Tom Schrijvers
KU Leuven, Belgium
(*e-mail:* tom.schrijvers@kuleuven.be)

submitted 29 April 2016; revised 8 July 2016; accepted 22 July 2016

Abstract

Tabling is a powerful resolution mechanism for logic programs that captures their least fixed point semantics more faithfully than plain Prolog. In many tabling applications, we are not interested in the set of all answers to a goal, but only require an aggregation of those answers. Several works have studied efficient techniques, such as lattice-based answer subsumption and mode-directed tabling, to do so for various forms of aggregation.

While much attention has been paid to expressivity and efficient implementation of the different approaches, soundness has not been considered. This paper shows that the different implementations indeed fail to produce least fixed points for some programs. As a remedy, we provide a formal framework that generalises the existing approaches and we establish a soundness criterion that explains for which programs the approach is sound.

KEYWORDS: tabling, answer subsumption, lattice, partial order, mode-directed tabling, denotational semantics, Prolog

1 Introduction

Tabling considerably improves the declarativity and expressiveness of the Prolog language. It removes the sensitivity of SLD resolution to rule and goal ordering, allowing a larger class of programs to terminate. As an added bonus, the memoisation of the tabling mechanism may significantly improve run time performance in exchange for increased memory usage. Tabling has been implemented in a few well-known Prolog systems, such as XSB (Swift and Warren 2010; Swift and Warren 2012),Yap (Santos Costa et al. 2012), Ciao (Chico de Guzmán et al. 2008) and B-Prolog (Zhou 2012), and has been successfully applied in various domains.

Much research effort has been devoted to improving the performance of tabling for

➔ formal semantics in a ***functional style***

Current Status

Tabling with Sound Answer Subsumption

Tabling with Sound Answer Subsumption

Alexander Vandenbroucke
KU Leuven, Belgium
(*e-mail:* alexander.vandenbroucke@kuleuven.be)

Maciej Piróg
KU Leuven, Belgium
(*e-mail:* maciej.pirog@kuleuven.be)

Benoit Desouter
Ghent University, Belgium
(*e-mail:* benoit.desouter@ugent.be)

Tom Schrijvers
KU Leuven, Belgium
(*e-mail:* tom.schrijvers@kuleuven.be)

submitted 29 April 2016; revised 8 July 2016; accepted 22 July 2016

Abstract

Tabling is a powerful resolution mechanism for logic programs that captures their least fixed point semantics more faithfully than plain Prolog. In many tabling applications, we are not interested in the set of all answers to a goal, but only require an aggregation of those answers. Several works have studied efficient techniques, such as lattice-based answer subsumption and mode-directed tabling, to do so for various forms of aggregation.

While much attention has been paid to expressivity and efficient implementation of the different approaches, soundness has not been considered. This paper shows that the different implementations indeed fail to produce least fixed points for some programs. As a remedy, we provide a formal framework that generalises the existing approaches and we establish a soundness criterion that explains for which programs the approach is sound.

KEYWORDS: tabling, answer subsumption, lattice, partial order, mode-directed tabling, denotational semantics, Prolog

1 Introduction

Tabling considerably improves the declarativity and expressiveness of the Prolog language. It removes the sensitivity of SLD resolution to rule and goal ordering, allowing a larger class of programs to terminate. As an added bonus, the memoisation of the tabling mechanism may significantly improve run time performance in exchange for increased memory usage. Tabling has been implemented in a few well-known Prolog systems, such as XSB (Swift and Warren 2010; Swift and Warren 2012),Yap (Santos Costa et al. 2012), Ciao (Chico de Guzmán et al. 2008) and B-Prolog (Zhou 2012), and has been successfully applied in various domains.

Much research effort has been devoted to improving the performance of tabling for

- ➔ formal semantics in a ***functional style***
- ➔ correctness condition

Current Status

Tabling with Sound Answer Subsumption

Under consideration for publication in Theory and Practice of Logic Programming 1

Tabling with Sound Answer Subsumption

Alexander Vandenbroucke
KU Leuven, Belgium
(*e-mail:* `alexander.vandenbroucke@kuleuven.be`)

Maciej Piróg
KU Leuven, Belgium
(*e-mail:* `maciej.pirog@kuleuven.be`)

Benoit Desouter
Ghent University, Belgium
(*e-mail:* `benoit.desouter@ugent.be`)

Tom Schrijvers
KU Leuven, Belgium
(*e-mail:* `tom.schrijvers@kuleuven.be`)

submitted 29 April 2016; revised 8 July 2016; accepted 22 July 2016

Abstract

Tabling is a powerful resolution mechanism for logic programs that captures their least fixed point semantics more faithfully than plain Prolog. In many tabling applications, we are not interested in the set of all answers to a goal, but only require an aggregation of those answers. Several works have studied efficient techniques, such as lattice-based answer subsumption and mode-directed tabling, to do so for various forms of aggregation.

While much attention has been paid to expressivity and efficient implementation of the different approaches, soundness has not been considered. This paper shows that the different implementations indeed fail to produce least fixed points for some programs. As a remedy, we provide a formal framework that generalises the existing approaches and we establish a soundness criterion that explains for which programs the approach is sound.

KEYWORDS: tabling, answer subsumption, lattice, partial order, mode-directed tabling, denotational semantics, Prolog

1 Introduction

Tabling considerably improves the declarativity and expressiveness of the Prolog language. It removes the sensitivity of SLD resolution to rule and goal ordering, allowing a larger class of programs to terminate. As an added bonus, the memoisation of the tabling mechanism may significantly improve run time performance in exchange for increased memory usage. Tabling has been implemented in a few well-known Prolog systems, such as XSB (Swift and Warren 2010; Swift and Warren 2012),Yap (Santos Costa et al. 2012), Ciao (Chico de Guzmán et al. 2008) and B-Prolog (Zhou 2012), and has been successfully applied in various domains.

Much research effort has been devoted to improving the performance of tabling for

- ➔ formal semantics in a ***functional style***
- ➔ correctness condition
- ➔ Please read the paper or come see my talk on Thursday

Open Problems

★ Current Correctness is much too coarse leading to false positives

★ Verification is hard. Automation is preferable.

Summary

Summary

★ logic programming is the most declarative programming paradigm

★ but LP under appreciated

★ functional programming makes logic programming more presentable

Questions