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:

\[ 1 \div 2 \]

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

\[ \text{swap} \ (m, n) = (n, m) \]
\[ \text{pair} = (1, 2) \div \text{swap} \text{pair} \]

The corresponding Haskell code is:

\[
\text{swap} :: [(a, b)] \rightarrow [(b, a)] \\
\text{swap} e = [(m, n) | (n, m) \leftarrow e] \\
\text{pair} :: [(\text{Int}, \text{Int})] \\
\text{pair} = [(1, 2)] \div \text{swap} \text{pair}
\]

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

\[
\Rightarrow \text{pair} \\
[(1, 2), (2, 1), (1, 2), (2, 1) \ldots]
\]

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 \(\text{pair}\) is then the set \([1, 2], (2, 1)\).

We can execute this model:

\[
\text{swap} :: (\text{Ord} \ a, \text{Ord} \ b) \Rightarrow \text{Set} \ (a, b) \rightarrow \text{Set} \ (b, a) \\
\text{swap} = \text{map} \ (\lambda (m, n) \rightarrow (n, m)) \\
\text{pair} :: \text{Set} \ (\text{Int}, \text{Int}) \\
\text{pair} = \text{singleton} (1, 2) \cup \text{swap} \text{pair} \\
\Rightarrow \text{pair} \\
\text{fromList} *** \text{Exception: <loop>}
\]

Haskell lazily prints the first part of the \(\text{Set}\) constructor, and then has to compute \(\text{union}\) 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-occurrence 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 \(\mathcal{R}^\bullet \). 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

1Here \(\text{map}\) comes from \text{Data.Set}
2. OVERVIEW

2.1 Non-determinism

A computation is non-deterministic if it has several possible outcomes. In this paper, we interpret such non-deterministic computations as the set of their results.

Consider the following non-deterministic expression that produces either 1 or 2:

\[ 1 \oplus 2 \]

For this example, the semantics is given by the set \( \{ 1, 2 \} \). The semantic function \( \cdot \) formally characterizes this interpretation:

\[
\begin{align*}
\llbracket n \rrbracket &= \{ n \} \\
\llbracket e_1 \oplus e_2 \rrbracket &= \llbracket e_1 \rrbracket \cup \llbracket e_2 \rrbracket
\end{align*}
\]

The semantics of a literal \( n \) is the singleton of \( n \), and the semantics of a choice \( e_1 \oplus e_2 \) is the union of the semantics of the left and right branches. A simple calculation shows that \( \llbracket 1 \oplus 2 \rrbracket \) is indeed \( \{ 1, 2 \} \).

\[ \llbracket 1 \oplus 2 \rrbracket = \llbracket 1 \rrbracket \cup \llbracket 2 \rrbracket = \{ 1 \} \cup \{ 2 \} = \{ 1, 2 \} \]

Let us extend our semantics to allow for addition of non-deterministic values:

\[ \llbracket e_1 + e_2 \rrbracket = \{ n + m \mid n \in \llbracket e_1 \rrbracket, m \in \llbracket e_2 \rrbracket \} \]

Now, consider the expression:

\[ (1 \oplus 2) + (1 \oplus 2) \]

Here we have an expression that contains an addition of two choices. This expression has 4 possible outcomes of which two coincide: the result is either 2 (1+1), 3 (1+2 or 2+1) or 4 (2+2).\(^\dagger\) Again calculation gives the expected result:

\[
\begin{align*}
\llbracket (1 \oplus 2) + (1 \oplus 2) \rrbracket &= \{ n + m \mid n \in \llbracket 1 \oplus 2 \rrbracket, m \in \llbracket 1 \oplus 2 \rrbracket \} \\
&= \{ n + m \mid n \in \{ 1, 2 \}, m \in \{ 1, 2 \} \} \\
&= \{ 2, 3, 4 \}
\end{align*}
\]

Recursion

The next example presents a recursive non-deterministic expression \( pair \) which chooses between \( (1, 2) \) and the new primitive \( swap \) to swap \( pair \)’s components around.

\[ \llbracket \text{pair} \rrbracket = \{ (1, 2) \oplus \text{swap pair} \} \]

The semantics of \( swap e \) is the set obtained by flipping all pairs in the set of results of \( e \):

\[ \llbracket swap e \rrbracket = \{ (m, n) \mid (n, m) \in \llbracket e \rrbracket \} \]

The semantics of \( \text{pair} \) are given by the following recursive equation:

\[
\begin{align*}
\llbracket \text{pair} \rrbracket &= \{ (1, 2) \oplus \text{swap pair} \} \\
\iff \llbracket \text{pair} \rrbracket &= \llbracket (1, 2) \rrbracket \cup \llbracket \text{swap pair} \rrbracket \\
\iff \llbracket \text{pair} \rrbracket &= \{ (1, 2) \} \cup \{ (m, n) \mid (n, m) \in \llbracket \text{pair} \rrbracket \}
\end{align*}
\]

\( \dagger \)The conventional list-based semantics would be \( \{ 2, 3, 3, 4 \} \). This equation admits infinitely many solutions, e.g.

\[
\begin{align*}
\llbracket \text{pair} \rrbracket &= \{ (1, 2), (2, 1) \}, \\
\llbracket \text{pair} \rrbracket &= \{ (0, 0), (1, 2), (2, 1) \}, \\
\llbracket \text{pair} \rrbracket &= \{ (1, 1), (1, 2), (2, 1) \}, \\
\ldots
\end{align*}
\]

However, we can identify a least solution: the set \( \{ (1, 2), (2, 1) \} \) is contained in every other solution.

As we saw previously, a naive translation of this idea in Haskell does not work:

\[
\begin{align*}
\text{det} &:: \text{Ord} a \Rightarrow a \rightarrow \text{Set} a \\
\text{det} x &= \text{singleton} x \\
\text{(.)} &:: \text{Ord} a \Rightarrow \text{Set} a \rightarrow \text{Set} a \rightarrow \text{Set} a \\
a \oplus b &= \text{union} a b \\
\text{swap} &:: (\text{Ord} a, \text{Ord} b) \Rightarrow \text{Set} (a, b) \rightarrow \text{Set} (b, a) \\
\text{swap} &= \text{Data.Set.map} (\lambda(x, y) \rightarrow (y, x)) \\
\text{pair} &:: \text{Set} (\text{Int}, \text{Int}) \\
\text{pair} &= \text{det} (1, 2) \oplus \text{swap pair} \\
\gg\gg \text{pair} \\
\text{fromList} \quad \text{Exception: } <\text{loop>}
\end{align*}
\]

The reason it does not work is that under Haskell’s (cpo-based) semantics the equation (1) has a different least solution: \( \bot \) (i.e. non-termination).\(^1\) As this additional \( \bot \) in the domain is clearly not the desired solution, we cannot rely on Haskell’s native semantics for recursion.

The main contribution of this paper is to reformulate the problem as a different least fixed point problem, for which we can iteratively compute the solution. Moreover, our approach incurs minimal overhead for the programmer, compared to writing the function using the conventional recursive approach.

2.2 Effect Handlers

Monads are a way to model side-effects such as non-determinism in a pure functional programming language [25]. In this paper, we use effect handlers to construct a monad for non-determinism and recursion. Effect handlers [10, 27] factor the problem of modeling effectful computations into two parts: first a syntax is introduced to represent all relevant operations, second effect handlers are defined that interpret the syntax within a semantic domain.

The syntax of non-deterministic computations can be modeled with the following data type \( ND \), which supports three operations: \( \text{Success}_{ND} a \) is a deterministic computation with result \( a \), \( \text{Fail}_{ND} \) is a failed computation, and \( \text{Or} l r \) represents a non-deterministic choice between two non-deterministic computations \( l \) and \( r \).

\[
\begin{align*}
\text{data} \ ND a &= \text{Success}_{ND} a \\
&\quad \mid \text{Fail}_{ND} \\
&\quad \mid \text{Or}_{ND} (\text{ND} a) (\text{ND} a)
\end{align*}
\]

Because the above data type is a free monad [20] we can easily define the following \textit{Monad} instance:

\[
\begin{align*}
\text{instance Monad ND where} \\
\text{return} a &= \text{Success}_{ND} a \\
\text{Success}_{ND} a \gg f &= f a \\
\text{Fail}_{ND} \gg f &= \text{Fail}_{ND} \\
\text{Or}_{ND} l r \gg f &= \text{Or}_{ND} (l \gg f) (r \gg f)
\end{align*}
\]

\(^1\)In Haskell we work in the domain \( \mathcal{P}(\mathbb{N} \times \mathbb{N}) \cup \{ \bot, \subseteq v \} \), where every type is inhabited by \( \bot \), representing non-termination, and \( \subseteq v \) for any value \( v \).
This monad instance substitutes \( f \) for every \( \text{Success}_{ND} \) \( a \) in the data structure, leaves \( \text{Fail} \) untouched, and recurses on both branches of \( \text{Or} \) \( l \) \( r \). With this monad instance, the example \((1 \rightarrow 2) + (1 \rightarrow 2)\) is expressed as:

\[
\text{example}_{ND} :: \text{ND} \ \text{Int}
\]

\[
\text{example}_{ND} = \text{do} \ x \leftarrow 1 \cdot \text{Or}_{ND} \text{'} \ \text{return} \ 2
\]

\[
y \leftarrow 1 \cdot \text{Or}_{ND} \text{'} \ \text{return} \ (x + y)
\]

Values of type \( \text{ND} \ a \) are abstract syntax trees. The function \text{example}_{ND} constructs such an abstract syntax tree. The interpreter \( nd \) decodes this tree according to the semantics defined by \( [\cdot] \). It turns \( \text{Success} \) into a singleton set, \( \text{Fail} \) into an empty set, and \( \text{Or} \) into the union of the interpretations of its branches.

\[
\begin{align*}
\text{nd} :: \text{Ord} \ a & \Rightarrow \text{ND} \ a \rightarrow \text{Set} \ a \\
\text{nd} (\text{Success}_{ND} \ a) & = \text{singleton} \ a \\
\text{nd} \ \text{Fail}_{ND} & = \text{empty} \\
\text{nd} \ (\text{Or}_{ND} \ l \ r) & = \text{union} \ (\text{nd} \ l) \ (\text{nd} \ r)
\end{align*}
\]

The equivalent of \( \text{pair} \) in the abstract syntax is \( \text{pair}_{ND} \):

\[
\begin{align*}
\text{pair}_{ND} :: \text{ND} \ (\text{Int}, \text{Int}) \\
\text{pair}_{ND} = \text{return} \ (1,2) \cdot \text{Or}_{ND} \text{'} \ f \mapsto \text{swap} \ \text{pair}_{ND} \ \\
\text{swap} \ (x,y) & = (y,x)
\end{align*}
\]

3. EXPlicating Recursion

In order to obtain a finite syntax tree, we have to once more change the representation of non-deterministic computations. First, we add a constructor to the abstract syntax to explicitly represent a recursive call. Second, we replace all recursive calls with this constructor, and finally, we define a new effect handler that interprets the now finite syntax tree, producing the desired solution. The following data type models recursive calls in addition to non-determinism:

\[
\begin{align*}
\text{data} \ \text{NDRec} \ i \ o \ a = & \ \text{Success} \ a \\
& \mid \ \text{Fail} \\
& \mid \ \text{Or} \ (\text{NDRec} \ i \ o \ a) \ (\text{NDRec} \ i \ o \ a) \\
& \mid \ \text{Rec} \ i \ (\text{a} \rightarrow \text{NDRec} \ i \ o \ a)
\end{align*}
\]

The first three constructors (1–3) capture non-determinism, exactly like the previously defined data type \( \text{ND} \). The last constructor (4) captures a recursive call: \( \text{Rec} \ a \ k \) represents a recursive call with argument \( a : i \), and continuation \( k : o \rightarrow \text{NDRec} \ i \ o \ a \). For convenience, we define four additional smart constructors. The smart constructor \( \text{rec} \) performs a recursive call and immediately wraps the result in a successful computation:

\[
\begin{align*}
\text{rec} :: & i \rightarrow \text{NDRec} \ i \ o \ a \\
\text{rec} \ i & = \text{Rec} \ i \text{ Success}
\end{align*}
\]

The smart constructor \( \text{choice} \) picks a computation from a list in a non-deterministic fashion.

\[
\begin{align*}
\text{choice} :: & [\text{NDRec} \ i \ o \ a] \rightarrow \text{NDRec} \ i \ o \ a \\
\text{choice} & = \text{foldr} \ \text{Or} \ \text{Fail}
\end{align*}
\]

The smart constructor \( \text{choose} \) picks an element from a list in a non-deterministic fashion.

\[
\begin{align*}
\text{choose} :: & [a] \rightarrow \text{NDRec} \ i \ o \ a \\
\text{choose} & = \text{choice} \circ \text{map} \ \text{Success}
\end{align*}
\]

The smart constructor \( \text{guard} \) returns () if its argument is true and fails otherwise.

\[
\begin{align*}
\text{guard} :: & \text{Bool} \rightarrow \text{NDRec} \ i \ o () \\
\text{guard} \ b & = \text{if} \ b \ \text{then} \ () \ \text{else} \ \text{Fail}
\end{align*}
\]

The data type \( \text{NDRec} \) is again a free monad, the corresponding monad instance is:

\[
\begin{align*}
\text{instance} \ Monad \ (\text{NDRec} \ i \ o \ a) \ where \\
\text{return} \ a & = \text{Success} \ a \\
\text{Success} \ a & \gg= \text{f} = \text{f} \ a \\
\text{Fail} & \gg= \text{f} = \text{Fail} \\
\text{Or} \ l \ r & \gg= \text{f} = \text{Or} \ (l \gg= \text{f}) \ (r \gg= \text{f}) \\
\text{Rec} \ i \ k & \gg= \text{f} = \text{Rec} \ i \ (\lambda x \rightarrow k \ x \gg= \text{f})
\end{align*}
\]

4. EFFECT HANDLER FOR EXPLICIT RECURSION

4.1 Denotational semantics

In this section we formalize the meaning of the abstract syntax. The meaning of a non-deterministic function \( f \) of type \( I \rightarrow \text{NDRec} \ I \ O \ O \) is obtained by a function \( [\cdot] : I \rightarrow \mathcal{P}(O) \). This function maps values of type \( I \) onto a subset of the values described by the type \( O \).

\[
[\cdot] : (I \rightarrow \text{NDRec} \ I \ O \ O) \rightarrow (I \rightarrow \mathcal{P}(O))
\]

Let us first define the semantics of an easier case: suppose we already have an environment \( s : I \rightarrow \mathcal{P}(O) \) that contains a partial set of solutions for every call \( f(a) \), i.e.

\[
s(a) \subseteq [f](a) \text{ for every } a \in I
\]

\[
[\cdot] :: (I \rightarrow \text{NDRec} \ I \ O \ O) \rightarrow (I \rightarrow \mathcal{P}(O))
\]

4The new continuation is the Kleisli composition \( k \gg f \) where \( \gg \) :: \( \text{Monad} \ m \Rightarrow (a \rightarrow m \ b) \rightarrow (b \rightarrow m \ c) \rightarrow (a \rightarrow m \ c) \).

5In deference to mathematical convention, we use uppercase characters for meta-variables \( I \) and \( O \) when using mathematical syntax.
Then for every syntax tree \( t : ND I O O \) we can define a semantic function \( \mathcal{R}[^t](s) \). This semantic function gives us the set of results associated with \( t \), given \( s \). Because the \( ND I O O \) data type is inductive, we define the function \( \mathcal{R}[^·] \) using structural recursion, as follows:

\[
\begin{align*}
\mathcal{R}[^\cdot] & : NDRec I O O \rightarrow (I \rightarrow \mathcal{P}(O)) \rightarrow \mathcal{P}(O) \\
\mathcal{R}[^\mathit{Success} \, x] & \equiv \{ x \} \\
\mathcal{R}[^\mathit{Fail}] & \equiv \emptyset \\
\mathcal{R}[^\mathit{Or} \, l \, r] & \equiv \mathcal{R}[^l](s) \cup \mathcal{R}[^r](s) \\
\mathcal{R}[^\mathit{Rec} \, i \, k] & \equiv \bigcup_{x \in s(i)} \mathcal{R}[^k(x)](s)
\end{align*}
\]

The two base cases are fairly simple: in the deterministic case (2) the result is just a singleton set of the result, and in the failure case (3) it is the empty set.

There are two inductive cases as well: a binary choice (4) and a recursive call (5). A binary choice is handled by taking the union of the results of the left and right branches. A recursive call has an argument \( i \) and a continuation \( k \). The result is obtained by finding the set of outcomes in the environment \( s \), and then applying the continuation \( k \) to every element \( x \in s(i) \), and taking the union of the results.

Now we can define \( [^·] \) as follows: since \( \mathcal{R}[^f(a)](\{ f \}) \) gives the set of outcomes of \( f(a) \) given environment \( \{ f \} \), and \( \{ f \}(a) \) gives the set of outcomes of \( f(a) \), the following must hold about \( \mathcal{R}[^f(a)](\{ f \}) \):

\[
\mathcal{R}[^f(a)](\{ f \}) = \{ f \} \quad (\forall a \in I)
\]

[equivalence of \( \lambda \)-abstraction and \( \forall \)-quantification]

\[
\Leftrightarrow \lambda a.\mathcal{R}[^f(a)](s) = \lambda a.\{ f \}(a)
\]

\[
\Leftrightarrow \lambda a.\mathcal{R}[^f(a)](\{ f \}) = \{ f \} \quad (\eta\text{-reduction})
\]

\[
\Leftrightarrow \{ f \} \text{ is a fixed point of } \lambda a.\lambda s.\mathcal{R}[^f(a)](s)
\]

Now all that remains is to choose a canonical fixed point for \( \{ f \} \), such that it corresponds to the desired meaning. Note that environments of the type \( I \rightarrow \mathcal{P}(O) \) (such as \( \{ f \} \)) are ordered by the ordering relation \( \sqsubseteq \):

\[
f \sqsubseteq g \iff \forall a \in I : f(a) \sqsubseteq g(a)
\]

The desired fixed point is the least fixed point\(^6\) denoted by \( \mathit{lfp}(\cdot) \), when fixed points are ordered by \( \sqsubseteq \):

\[
\{ f \} = \mathit{lfp}(\lambda a.\lambda s.\mathcal{R}[^f(a)](s))
\]

(6)

To make this more concrete, consider the denotational semantics of \( \mathit{pairNDRec} \).

\[
\begin{align*}
\mathit{pairNDRec}(\cdot) \equiv \lambda i \mathcal{R}[^\mathit{Rec} i \mathit{(()(0))} & \quad \text{[by (6)]} \\
| \mathit{\lambda}().\{(1, 2), (2, 1)\} & \text{is the least fixed point} \\
\end{align*}
\]

\[
\{(1, 2), (2, 1)\} \\
\]

\[\text{function application}\]

\[\{(1, 2), (2, 1)\}\]

--

\(\text{The least fixed point is well defined, as the function is always continuous (if the syntax tree is finite) and therefore monotonic, but the domain } I \rightarrow \mathcal{P}(O) \text{ may not possess the finite ascending chain property, preventing us from computing the solution in a finite amount of time.}\)

---

\(\text{Functions and types residing in } \text{Data.Map} \text{ are imported with the qualifier } \text{M}, \text{ except the lookup operator "!".}\)
If the syntax tree contains a deterministic result (line 4), we simply add this result to the environment. If the syntax contains a failure (line 5), the environment remains unchanged. When presented with a binary choice (line 6), we first update the environment according to the left branch and then according to the right branch. Finally, in the case of a recursive call \textit{Rec} \textit{j} \textit{k}, we first check if the map contains an entry for the argument \textit{j} (line 7). If it has no such entry, we add an entry containing the empty set (line 8), otherwise we update the environment based on the existing elements (line 9).

Inserting an empty entry for \textit{j} ensures that the next iteration of \textit{step} also updates the map for \textit{expr} \textit{j}.

We still need a function \textit{lfp} to iteratively compute a least fixpoint. As an initial value, this function keeps calling itself until the result no longer changes.

\textbf{Graph Example} We compute reachability in a cyclic directed graph using our non-determinism framework. We use the adjacency list representation of a graph where every \textit{Node} has a label (of type \textit{String}) and a list of adjacent nodes:

\begin{verbatim}
data Node = Node { label :: String, adj :: [Node] } instance Eq Node where n1 ≡ n2 = label n1 ≡ label n2 instance Ord Node where n1 ≤ n2 = label n1 ≤ label n2
\end{verbatim}

Our example graph (see Figure 3) consists of five nodes. It contains two cycles: from 3 to 4 and back, and from 5 to itself.

\begin{verbatim}
[ n1, n2, n3, n4, n5 ] = [ Node "1" [ n2, n5 ], Node "2" [ n3 ], Node "3" [ n4 ], Node "4" [ n3, n1 ], Node "5" [ n1 ] ]
\end{verbatim}

Reachability is straightforwardly computed as:

\begin{verbatim}
reach :: Node \rightarrow NDRec Node Node Node
reach n = return n `Or` (choose (adj n)) \gg= rec
\end{verbatim}

i.e. \textit{n} is reachable from \textit{n} and all nodes reachable from a neighbor of \textit{n} are reachable from \textit{n}.

\begin{verbatim}
\gg= runNDRec reach n1
fromList [ Node 1, Node 2, Node 3, Node 4, Node 5 ]
\end{verbatim}

5. DEPENDENCY TRACKING

The effect handler that was provided in Section 4 computes the least fixed point in a very naive way, resulting in sub par performance. To illustrate the problem, consider the following program to compute the Fibonacci numbers:

\begin{verbatim}
fib :: Int \rightarrow NDRec Int Int Int
fib n | n \equiv 0 = return 0
| n \equiv 1 = return 1
| otherwise = do f1 \leftarrow rec (n - 1)
f2 \leftarrow rec (n - 2)
return (f1 + f2)
\end{verbatim}

The evolution of the environment for \textit{runNDRec} \textit{fib} \textit{2} after each application of \textit{step} is shown in Table 1 (ignore the list after the comma for now). Every value in the environment is recomputed in every step. The work performed by the effect handler is monotonic: the work performed by a single iteration is also performed in all later iterations. In the case of \textit{fib} this leads to an \textit{O}(n^2) runtime.6

This duplication of work has two sources:

1. The naive least fixed point computation \textit{lfp} iteratively applies the \textit{step} function which folds over all keys in the environment, even when the set of outcomes of that key has not changed.

2. The function \textit{go} computes too much: the only part of the syntax tree that is influenced by a recursive call \textit{Rec} \textit{j} \textit{k} is the \textit{continuation} \textit{k} of that recursive call. Therefore only the continuation needs to be recomputed.

Although efficiency is not the main focus of this paper, we would be remiss not to address such a glaring problem. Especially since the solution is simple: both problems can be solved by keeping track of which continuation depends on which recursive call (identified by its argument), and only evaluating the continuation once for every new value of the recursive call.

Figure 4 shows the code for the interpreter \textit{runNDk} that tracks dependencies. This effect handler relies on seven auxiliary functions defined for the type \textit{Env} \textit{i} \textit{o}. This type abstracts over environments that contain continuations in addition to a set of values.

We show their signatures and implementations in Figure 4.

The computation starts with the call to \textit{go} on line 1, which returns the environment after the least fixed point computed. From this environment the result is extracted using \textit{!!!}. The \textit{go} function proceeds by case analysis: if the syntax tree is \textit{Success} \textit{x} and the value \textit{x} is not yet in the environment, the environment is updated by sequentially applying every continuation that is currently in the environment (line 2-3). If the syntax tree is a failed computation the environment is unchanged (line 4). If the syntax tree is a binary choice (line 5) the environment is updated first for the left

\begin{verbatim}
2: [2],[] 1: {1},[go 2 \circ k]\ 0: [0],[go 1 \circ k]
2: [2],[1] t: {1},[go 2 \circ k]\ 0: [0],[go 1 \circ k]
2: [2],[] t: {1},[go 2 \circ k]\ 0: [0],[go 1 \circ k]
2: [2],[] t: {1},[go 2 \circ k]\ 0: [0],[go 1 \circ k]
2: [2],[] t: {1},[go 2 \circ k]\ 0: [0],[go 1 \circ k]
\end{verbatim}

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
1 & 2 & 3 \\
\hline
4 & 5 & 6 \\
\hline
\end{tabular}
\caption{The graph used in the reachable and shortest path example.}
\end{table}

\begin{table}
\centering
\begin{tabular}{|c|c|c|}
\hline
0 & 0 & 0 \\
\hline
1 & 0 & 0 \\
\hline
0 & 1 & 0 \\
\hline
0 & 0 & 1 \\
\hline
0 & 1 & 1 \\
\hline
\end{tabular}
\caption{The trace of the environment for \textit{runNDRec} \textit{fib} 2.}
\end{table}

6modulo a logarithmic factor for the use of finite maps.
branch, then for the right branch. If the syntax tree is a recursive call \( \text{Rec} \ j \ k \) (line 6-10), and there is no entry for \( j \) in the map (line 9), i.e. this is the first time we see a call to \( j \), a continuation for \( j \) is added (line 7) and the recursive function is evaluated for \( j \). Otherwise (line 10), a new continuation is added for \( j \), and this continuation is applied to every known result of \( j \). The least fixed point is reached when the \( \text{tfNew} \) check on line 2 no longer succeeds.

The trace for \( \text{runNDk} \ fib \ 2 \) is identical to the trace for \( \text{runND} \ fib \ 2 \), but now only a single entry is updated in each iteration. Moreover, the result from the call to 1 only triggers the continuation \( k_0 = \lambda f_1 \rightarrow \text{Rec} \ 0 \ (f_1 + f_2) \) and the result of 0 only triggers computation of \( h = \lambda f_2 \rightarrow \text{return} (1 + f_2) \), instead of the entire syntax tree \( fib \ 2 \).

In summary, we obtain a linear runtime for \( \text{runNDk} \ fib \ 2 \).

6. LATTICES

Reconsider the graph defined in Section 4. Instead of just finding all reachable nodes, we may want to additionally know the length of the shortest path. The program \( sp \) is a first attempt at computing the shortest path between \( src \) and \( dst \).

\[
\begin{align*}
\text{sp} & : \text{Node} \to \text{Node} \to \text{NDRec} \ Node \ Int \\
\text{sp} & \ text{dst} \ text{src} \\
| \ text{dst} = \ text{src} & \Rightarrow \ return \ 0 \\
| \ otherwise & \Rightarrow \ chooses (adj \ text{src}) \ \exists = \ text{rec} \ \exists = \ text{return} \ (+1)
\end{align*}
\]

The denotational semantics gives us the following value for the expression \( sp \ n_1 \text{rec} n_2 \):

\[\text{[sp n_1 rec]} n_2 \{2, 4, 6, 8, \ldots\}\]

Since this value is infinite, the effect handler does not terminate. In order to implement a terminating effect handler, we need to first define an alternative denotational semantics that at least produces a finite value, which we can then compute in a finite amount of time. **Lattice-based Semantics** Instead of powersets \( \mathcal{P}(O) \), we generalize the semantic domain to an arbitrary complete lattice \( L \). On this domain we define a new denotational semantics for non-deterministic computations. The signature of this denotational semantics is:

\[\lldot : (I \to \text{NDRec} I \ L \ L) \to (I \to L)\]

where \( \langle L, \sqcup \rangle \) is a complete lattice.

A complete lattice is a partially ordered set \( \langle S, \sqcup \rangle \) such that for every subset \( X \subseteq S \) there exists a least upper bound, i.e. an element \( \sqcup X \) that is the least element that is larger than every element in \( X \), more formally:

**Definition 1 (Complete Lattice).** A complete lattice \( \langle L, \sqcup \rangle \) is a partially ordered set \( \langle L, \sqcup \rangle \) where \( \forall X \subseteq \ L \) there exists a least upper bound \( \sqcup X \in L \), such that:

\[\forall z \in L : \sqcup X \sqsubseteq z \iff \forall x \in X : x \sqsubseteq z\]

It follows from the definition that \( \sqcup X \) is unique. A complete lattice \( L \) always has a least element \( \bot \) that is smaller than all other elements. It corresponds to the least upper bound of the empty set: \( \bot = \sqcup \emptyset \). With a slight abuse of notation we also write \( a \sqcup b \) to denote the least upper bound \( \sqcup \{a, b\} \), pronounced “join”.

As before, we first consider the semantics for the case where we already possess an environment \( s : I \to L \) containing the (partial) solution for every call \( f(a) \), i.e.

\[s(a) \sqsubseteq \text{[f]}_L(a) \text{ for every } a \in I\]

Then for every syntax tree \( t : \text{NDRec} I \ L \ L \) we can define the denotational semantics \( \mathcal{R} \lldot \), in the new setting:

\[
\begin{align*}
\mathcal{R}[\cdot]_L & : \text{NDRec} I \ L \ L \to (I \to L) \\
\mathcal{R} & \ Success x \ [L](s) = x \\
\mathcal{R} & \ Fail \ [L](s) = \perp \\
\mathcal{R} & \ Or \ l r \ [L](s) = \mathcal{R}[l]_L(s) \sqcup \mathcal{R}[r]_L(s) \\
\mathcal{R} & \ Rec \ j k \ [L](s) = \mathcal{R}[k(s(j))]_L(s)
\end{align*}
\]

This semantics is very similar to the previously defined semantics for sets. In particular, the semantics \( \mathcal{R}[\cdot]_L \) almost corresponds to the semantics \( \mathcal{R}[\cdot]_S \) specialized to the powerset of the output type \( (\mathcal{R}[\cdot]_P(O)) \). We will make this relationship more precise in Section 6.1.

For \( f \lldot \) we would again like to compute the least fixed point of \( \lambda s.\lambda a.\mathcal{R}[f(a)]_L(s) \). However, unlike for sets, this function is not guaranteed to be monotonic for arbitrary lattices in general, so a least fixed point might not always exist. Instead, we define the operator \( \triangleright \):

**Definition 2.** The operator \( \triangleright \) is defined as:

\[
\begin{align*}
f & \triangleright 0 = \perp \\
f & \triangleright (i + 1) = f \triangleright i \sqcup f(f \triangleright i) \\
f & \triangleright \infty = \bigcup_{i=0}^{\infty}(f \triangleright i)
\end{align*}
\]

Observe that the sequence \( f \triangleright 0, f \triangleright 1, f \triangleright 2, \ldots, f \triangleright \infty \) is non-decreasing.\(^9\)

If the domain of \( f \) has the finite ascending chain property this sequence is finite, and then \( f \triangleright \infty \) must be a fixed point. Thus, we define \( \lldot \) as:

\[\lldot f = (\lambda s.\lambda a.\mathcal{R}[f(a)]_L(s)) \triangleright \infty\]

Moreover, if \( \lambda s.\lambda a.\mathcal{R}[f(a)]_L(s) \) is continuous, then \( \triangleright \) computes the least fixed point, i.e.

\[\text{lfp}(\lambda s.\lambda a.\mathcal{R}[f(a)]_L(s)) = (\lambda s.\lambda a.\mathcal{R}[f(a)]_L(s)) \triangleright \infty = \lldot f\]

Otherwise, \( \triangleright \) may compute a fixed point that is strictly greater than the least fixed point. In practice, however, almost all functions are continuous.

**Shortest Path Example** For our shortest path problem, the relevant partial order is \( \langle \mathbb{N} \cup \{\infty\}, \subseteq \rangle \) where

\[a \subseteq b \iff a = \infty \text{ or } b \leq a\]

which is the reverse order of the canonical order on \( \mathbb{N} \), i.e. smaller (shorter) is better. Since the canonical order is well-founded (i.e. has no infinite descending chains), \( \subseteq \) has the finite ascending chain property. The relevant lattice is \( \langle \mathbb{N} \cup \{\infty\}, \subseteq \rangle \) where

\[\sqcup X = \begin{cases} \infty & \text{if } X = \emptyset \\ \min_{\subseteq} X & \text{otherwise} \end{cases}\]

In this lattice the least fixed point exists, since continuation \( \text{return}_0 \) (+1) in \( \text{sp} n_1 \) is continuous.

\[\lldot \text{sp} n_1 \{n_2 \in \mathbb{N}(\infty) \} n_2 = 3\]

\^9 For environments, assume the order \( \preceq \) defined as:

\[s_1 \preceq s_2 \iff \forall a \in I : s_1(a) \sqsubseteq s_2(a)\]
runNDk :: (Ord i, Ord o) ⇒ (i → NDRec i o o) → i → Set o
runNDk expr $i = go $i (expr $i) emptyEnv !!! $i where
go $i (Success x) m = guard (not member o m) !! m
  where newEnv = foldr ($x) (store i x m) (cons i m)
go $i Fail = m

  go $i (Or l r) m = go $i l m ∪ go $i r m
  go $i (Rec j k) m =
    let newEnv = addCont $j (expr $j) newEnv
     in case results $j m of
     Nothing → go $j (expr $j) newEnv
     Just rs → foldr (go $i o k) newEnv (toList rs)

ifNew :: (Ord i, Ord o)
  ⇒ i → o → Env i o o → Env i o o
ifNew i o env newEnv =
  if maybe True (∼ o member o) (results i m)
  then newEnv
  else env

Figure 4: Effect Handler and Environment type for dependency tracking.

The length of the shortest path from node 2 to node 1 is indeed 3.
When a path does not exist, we get the following length:

\[
\emptyset \cup \text{null} (\text{null}) (\text{null}) = \infty
\]

Notice the similarity to solving a problem with Dynamic Programming. The idea of dynamic programming is to compute the optimal solution to a problem by combining optimal solutions to smaller instances of the same problem. In this setting, recursive calls correspond to instances of the same problem. The lattice allows us to keep only the optimal solution to a given instance, from which the result of a larger instance is computed.

Subset Sum Example Another example is the following: Given a list of integer numbers and an integer s, find the shortest non-empty sublist that sums to s.
To solve this problem, first observe that lists form a lattice when ordered by descending length, if we adjoin a bottom element InfList representing any list of infinite length. This is embodied by the Shortest-datatype (see Section 4.2 for the definition of Lattice).

data Shortest = InfList | List [Int]

cons :: Int → Shortest → Shortest
cons n InfList = InfList
cons n (List ns) = List (n : ns)

instance Lattice Shortest where
  bottom = InfList
  join InfList a = a
  join a InfList = a
  join (List a) (List b)
    | length a ≤ length b = List a
    | otherwise = List b

The function sss returns the Shortest list which sums to its first argument and is a sublist of its second argument.

  sss :: (Int, [Int]) → NDRec (Int, [Int]) Shortest Shortest
  sss (n, [])
    | n ≡ 0 = return (List [])
    | otherwise = Fail
  sss (n, x : xs) = choice [rec (n, x),
                          fmap (cons x) (rec (n + x, xs))]

When the input list is empty we can only construct an empty list with sum 0. When the input list is non-empty, the shortest list summing to n is obtained by either recursively searching for a list summing to n, or consuming x onto a list that sums to n – x.

The function \( \mathcal{R} \) on \( sss (n, x) \) is continuous for all \( (n, x) \). Moreover, the environment possesses a finite ascending chain property: every recursive call decreases the size of the input list by one. Then, given a finite input list, there can only be a finitely many recursive calls.

For instance, consider the application of \texttt{sss} to \((10, [5, 0, 5])\):

\[
\text{data Production} = \text{Symbol} \rightarrow \text{Symbol}
\]

There is a single \texttt{Symbol} to the left of \texttt{→}, called the head, and zero or more symbols, the body, on the right hand side. Symbols can be either terminals or non-terminals:

type Symbol = Char

isTerminal :: Symbol → Bool
isTerminal s = (∼ isUpper s)

Terminals may not appear in the head of a rule. For our purpose we distinguish terminals and non-terminals based on whether they are upper or lower case characters.

Now, consider the following grammar:

\[
\text{grammar} :: \text{Grammar}
\]

This grammar describes a simple expression language consisting of one identifier \((a)\), one literal \((1)\) and a binary operator \((+)\). We implement a program that analyzes grammars for nullability: a symbol is \texttt{Nullable} if it derives the empty string. Nullability is charac-
terized by a function from Symbol to Any, where Any is a lattice of booleans, where join is disjunction and False is bottom.

def newtype Any = Any { getAny :: Bool }

instance Lattice Any where
  bottom = Any False
  join (Any a) (Any b) = Any (a ∨ b)

Given this lattice, the implementation of nullable is straightforward:

**nullable** :: Grammar → Symbol → NDRec Symbol Any Any

nullability grammar

isTerminal s = return (Any False)
otherwise = do
  head ← body ← choose g
  guard (head ≡ s)
  nullables ← mapM rec body
  return (Any (and (map getAny nullables)))

A terminal symbol is never nullable. In order to decide if a non-terminal symbol s is nullable, we need to check if there exists a production with s in the head and with a body consisting of nullable symbols. Note that s may also occur in the body.

We evaluate nullability for grammar:

nullability grammar

and every

Lattice l

class Lattice l where
  bottom :: l
  join :: l → l → l

This typeclass prioritizes the view of the ⊔-operator as a binary operator because this is the most useful in practice.

Now we replace every FailF in a value of type NDRec with bottom and every OrF with join:

runND :: (Functor f, Lattice o)
  ⇒ Free (NDRec f o) o → Free f o
runND = fold return alg where
  alg (Inl FailF) = return bottom
  alg (Inl (OrF l r)) = join <$> l <<< r
  alg (Inr f) = Free f

The effect handler runND effectively eliminates the NDRec effect from the syntax tree. Only the recursive effect RecF i o remains. This effect is then interpreted by runRec:

runRec :: (Ord i, Eq a, Lattice o)
  ⇒ (i → Free (RecF i o) o) o → i → o
runRec expr = lfp step =o where
  =o = M.singleton =o bottom
  step m = foldr (λk → go (expr k) m (M.keys m))
  go expr = fold onReturn alg expr where
    onReturn x i = M.insertWith join i x
    alg (RecF j k) i m = case M.lookup j m of
      Nothing → k bottom i (M.insert j bottom m)
      Just l → k l i m

Note that runRec does not expect a plain effect, but an effectful function i → Free (RecF i o) o, and also returns a function i → o. Essentially, runRec is a memoizing fixpoint operator applied to the recursive equation defined by expr.

Reconsider the shortest path program from Section 6:

spF :: Node → Node → NDRecF Node Dist Dist
spF dst src | src ≡ dst = return 0
  otherwise = do n ← chooseF (adj src)
               d ← recF n
               return (d + 1)
where the Dist datatype and its instances are defined as:

```haskell
data Dist = InfDist | Dist Int deriving (Eq, Show)
instance Lattice Dist where
  bottom = InfDist
  join InfDist a b = a
  join a b = InfDist
  join (Dist d1) (Dist d2) = Dist (min d1 d2)
instance Num Dist where
  InfDist + a = InfDist
  a + InfDist = InfDist
  Dist d1 + Dist d2 = Dist (d1 + d2)
  fromInteger d = Dist o fromInteger
```

Note that Dist has a Lattice-instance corresponding to the lattice \((\mathbb{N} \cup \{\infty\}, \sqcup)\) from Section 6. Combining runND and runRec gives an effect handler \(\text{runRec} \circ (\text{runND} \circ \text{f})\) implementing the semantic function \(\mathcal{J}_{\sqcup}\). For \(\text{spF}\) this effect handler computes the expected result:

\[
\Rightarrow \text{runRec} (\text{runND} \circ \text{spF} n_1) n_2 \quad \text{Dist} 2
\]

\[
\Rightarrow \text{runRec} (\text{runND} \circ \text{spF} n_1) n_0 \quad \text{InfDist}
\]

### Lifting Continuations

The semantic functions \(\mathcal{R}[-]_\sqcup\) and \(\mathcal{R}[-]_{\mathcal{I}}\) only differ in the case for \(\text{Rec} j k\): \(\mathcal{R}[-]_{\mathcal{I}}\) computes the union of the \(k\) applied to every element of \(s(j)\), while \(\mathcal{R}[-]_\sqcup\) directly applies the \(k\) to \(s(j)\). The similarity between \(\mathcal{R}[-]_{\mathcal{I}}\) and \(\mathcal{R}[-]_{\mathcal{O}}\) suggests that we can obtain the behaviour of the former with the latter if we just change the continuation in \(\text{Rec} j k\) appropriately. More formally, we want to construct a continuation \(k'\) such that:

\[
\mathcal{R} [\text{Rec} j k']_\mathcal{I}(s) = \mathcal{R} [\text{Rec} j k']_{\mathcal{O}}(s)
\]

This is achieved by the following code, that lifts a computation in the Free \((\text{NDF} + \text{RecF} i o)\)-monad to one in the Free \((\text{NDF} + \text{Rec} i (\text{Set} a))\)-monad.

```haskell
liftRec :: Ord o
         => Free (NDF + RecF i o) a
         -> Free (NDF + RecF i (Set o)) (Set o)
liftRec = fold (return o singleton) alg where
  alg (Inl f) = Free (Inl f)
  alg (Inr (RecF i k)) = Free (Inr (RecF i k')) where
  k' = fmap unions o traverse k o toList
```

Since the powerset \((\mathcal{P}(\mathcal{O}), \cup)\) forms a lattice, we can define a Lattice instance for Set:

```haskell
instance Ord a => Lattice (Set a) where
  bottom = empty
  join = union
```

Composing liftRec, runND and runRec yields a complete effect handler for Free \((\text{NDF} + \text{RecF} i o)\), corresponding to \(\mathcal{R}[-]_{\mathcal{I}}\):

\[
\Rightarrow \text{runNDRecF} :: (\text{Ord} i, \text{Ord} o)
\Rightarrow (i \rightarrow \text{Free} (\text{NDF} + \text{RecF} i o) a)
\Rightarrow (i \rightarrow \text{Set} o)
\text{runNDRecF} f = \text{runRec} (\text{runND} \circ \text{liftRec} o f)
```

### Examples

Redefining our running example:

```haskell
pairF :: () \rightarrow \text{NDRecF} () (\text{Int}, \text{Int})
pairF () = return (1, 2) \circ orF \text{ do } (x, g) \leftarrow \text{recF} ()
\text{ return } (y, x)
```

leads to the following familiar result:

\[
\Rightarrow \text{runRecF} \circ \text{pairF} ()
\text{ fromList } [(1, 2), (2, 1)]
\]

To recapitulate, we have shown how to implement the effect handler for \([\cdot]_{\mathcal{I}}\) for \(\text{NDRecF}\). On top of this we implement the effect handler for \([\cdot]_{\mathcal{O}}\) for sets, showing that the semantics for lattices generalize the semantics for sets.

### 7. MUTUAL RECURSION

The syntax we have defined so far has a serious limitation: it can only handle a single function at a time. This section lifts this limitation by extending the syntax to support several, potentially mutually recursive functions within the same non-deterministic computation. Moreover, these functions are allowed to possess different argument and return types.

#### Motivating Example

We demonstrate the added power with another grammar analysis. We implement a program that finds the First set of a symbol. A symbol \(X\) is in the First set of another symbol \(Y\) if any of the strings derived from \(Y\) start with \(X\). Solving First requires also solving Nullable.

Recall that nullability is characterized by a function from Symbol to Any. First is also characterized by a function, but one from Symbol to Set Symbol (recall that Set is also a lattice). We encode this fact with a Generalized Algebraic Data Type (GADT) [14]:

```haskell
data Analyze o where
  Nullable :: Symbol \rightarrow \text{Analyze Any}
  First :: Symbol \rightarrow \text{Set Symbol}
```

The arguments to the data constructors Nullable and First indicate the argument type (in both cases Symbol). The argument to the type constructor Analyze indicates the return type (Any or Set Symbol).

The syntax data type NDM is a variation of NDRec:

```haskell
data NDM i a where
  SuccessM :: a \rightarrow \text{NDM} i a
  FailM :: \text{NDM} i a
  OrM :: \text{NDM} i a \rightarrow \text{NDM} i a \rightarrow \text{NDM} i a
  RecM :: \text{Lattice} o
         \Rightarrow o \rightarrow (o \rightarrow \text{NDM} i a) \rightarrow \text{NDM} i a
```

The recursive call constructor RecM expects an existentially qualified parameter \(o\) (which must be a lattice). The previous syntax introduces \(o\) in the type, and as such fixes \(o\) to one particular type for the entire syntax tree. Because of the existential parameter, NDM does allow calls with different return types. Furthermore, the argument is of type \(i o\). When \(i\) is a GADT, such as Analyze, the type \(i o\) will only be inhabited for the desired \(o\) (e.g. Any and Set Symbol for Analyze).

We also define four additional smart constructors recM, chooseM, choiceM and guardM. Their meaning is analogous to the smart constructors for NDRec. For brevity, their signature and implementation is omitted here. With these smart constructors we define explicit recursive calls for Nullable and First.

```haskell
nullable :: Symbol \rightarrow \text{NDM Analyze Bool}
nullable = fmap getAny o recM o Nullable
```

```haskell
first :: Symbol \rightarrow \text{NDM Analyze Symbol}
first s = recM (First s) \Rightarrow \text{chooseM} o toList
```

Solving Nullable and First is straightforward:

```haskell
analyze :: Grammar \rightarrow \text{Analyze o} \rightarrow \text{NDM Analyze o}
analyze g (Nullable s)
```

At last we have all the pieces to define the effect handler. The function `goM` evaluates an `NDM i o`, given an environment `EnvM i` and `runNDM` computes the least fixed point of `goM`:

```
| goM :: (Coerce i, Lattice o, Ord (Input i)) => i o -> NDM i o -> EnvM i |
| goM i (SuccessM x) m = M.insertWith (joinO i) (Input i) (Output i x) m |
| goM i (FailM m) = M.insertWith (joinO i) (Input i) m |
| goM i (OrM l r) = goM i l o r (goM i l m) |
| goM i (RecM j k) m = case M.lookup (Input j) m of |
| Nothing -> M.insert (Input j) (Output j bottom) m |
| Just s -> goM i (k s) m |
```

In order to determine the `First` set of a symbol `s`, we need to find the longest nullable prefix of the body for every production with `s` in the head. Then we find the `First` sets of every symbol in the prefix and the symbol directly following the prefix.

**Implementation** From `Analyze` we derive the actual input and output types that need to be stored in the environment:

```
type EnvM i = M.Map (Input i) (Output i)
data Input :: i => * -> * where |
  Input :: Lattice o => i o -> Input i |
data Output :: i => * -> * where |
  Output :: i o -> Output i |
```

The data types `Input` and `Output` wrap a particular instantiation of the `Analyze` type constructor in an existential quantification. The implementation of the `Eq` and `Ord` instances is elided for brevity.

We use the typeclass `Coerce` to type-safely coerce `Output Analyze` back to `Any` or `Set Symbol`, depending on which constructor of `Analyze` is pattern-matched.

```
class Coerce c where |
  coerce :: c o -> Output c -> Maybe o |
instance Coerce Analyze where |
  coerce (Nullable s) (Output (Nullable t) o) = Just o |
  coerce (First s) (Output (First t) o) = Just o |
  coerce a o = Nothing |
```

Now we lift `join` to `Outputs`:

```
joinO :: (Coerce i, Lattice o) => i o -> Output i -> Output i |
joinO i o1 o2 = |
  maybe a1 (Output i) (join <$> coerce i o1) |
```

At last we have all the pieces to define the effect handler. The function `goM` evaluates an `NDM i o`, given an environment `EnvM i` and `runNDM` computes the least fixed point of `goM`:

```
| goM :: (Coerce i, Lattice o, Ord (Input i)) => i o -> NDM i o -> EnvM i |
| goM i (SuccessM x) m = M.insertWith (joinO i) (Input i) (Output i x) m |
| goM i (FailM m) = M.insertWith (joinO i) (Input i) m |
| goM i (OrM l r) = goM i l o r (goM i l m) |
| goM i (RecM j k) m = case M.lookup (Input j) m of |
| Nothing -> M.insert (Input j) (Output j bottom) m |
| Just s -> goM i (k s) m |
```

This implementation is not very different from `runNDRec` (see Section 4), except that some wrapping and unwrapping of `Input` and `Output` is required. Note that dependency tracking is orthogonal to the problem of mutual recursion, i.e., dependency tracking is easily added.

**Evaluation** Using `runNDM` we now solve `First`, which requires solving `Nullable`, in the same computation.

```
analyzeF :: Grammar -> Symbol -> [Symbol] |
analyzeF g = foldr (analyzeF (symbol g) o) First |
```

The computed `First` sets of `E`, `Z` and `T` are the results one would expect from the definition: an expression may start with a parenthesis, a literal or an identifier, and similarly for sub-expressions and terms. We now extend `First` to strings of symbols:

```
firstS :: Grammar -> [Symbol] -> Set Symbol |
firstS g s = |
  let nulls = takeWhile analyzeN s |
  notNulls = dropWhile analyzeN s |
  prefix = nulls ++ take 1 notNulls |
  analyzeN = getAny $
    runNDM (analyze g) (symbol g) |
  in unions $ map (runNDM (analyze g) o) First |
```

So called `left-recursive conflicts` occur when the `First` sets of bodies of two rules with the same head have a non-empty intersection. The following function `hasConflict` uses `firstS` of rule bodies to detect conflicts:

```
hasConflict :: Grammar -> Bool |
hasConflict g = |
  let ruleHead (h -> w) = h |
  ruleBody (w -> b) = b |
  ruleConflict [r] = False |
  ruleConflict rs = (null (foldr1 intersection firsts)) |
  where firsts = map (firstS g o) ruleBody |
  in or $ map ruleConflict $ groupBy ((\equiv) `on` ruleHead) g |
```

Our example grammar is conflict free:

```
⇒ hasConflict grammar False |
```

However, in the following grammar the rules for `E` do conflict:

```
grammar2 :: Grammar |
grammar2 = ["E" -> "E + T", "E" -> "T", "T" -> "a", "T" -> "1"] |
⇒ hasConflict grammar2 True |
```

Note that the implementation of `firstS` and `hasConflict` is not very efficient: in between calls to `runNDM` the results are simply thrown away. We see two ways to resolve this: either have
In the IFL2015/benchmarks directory.

**Table 2: The benchmark results in milliseconds.**

<table>
<thead>
<tr>
<th></th>
<th>Size</th>
<th>Naive</th>
<th>Dependency Tracking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci</td>
<td>800</td>
<td>455.7</td>
<td>13.77</td>
</tr>
<tr>
<td></td>
<td>805</td>
<td>462.8</td>
<td>13.85</td>
</tr>
<tr>
<td></td>
<td>810</td>
<td>469.3</td>
<td>13.97</td>
</tr>
<tr>
<td>Knapsack</td>
<td>10</td>
<td>155.9</td>
<td>8.185</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>226.6</td>
<td>12.32</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>309.0</td>
<td>16.04</td>
</tr>
<tr>
<td>NQueens</td>
<td>7</td>
<td>4.864</td>
<td>1.259</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>152.9</td>
<td>26.79</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>914.8</td>
<td>132.0</td>
</tr>
<tr>
<td>SCC</td>
<td>30</td>
<td>57.31</td>
<td>8.843</td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>104.9</td>
<td>10.67</td>
</tr>
<tr>
<td></td>
<td>35</td>
<td>97.97</td>
<td>12.67</td>
</tr>
<tr>
<td>Shortest Path</td>
<td>8000</td>
<td>1306</td>
<td>676.6</td>
</tr>
<tr>
<td></td>
<td>8500</td>
<td>1362</td>
<td>718.7</td>
</tr>
<tr>
<td></td>
<td>9000</td>
<td>1451</td>
<td>775.1</td>
</tr>
</tbody>
</table>

runNDM return the environment EnEM directly to perform lookups, or embed firstS and hasConflict in NDM as well. Such an embedding requires extending Analyze with new constructors for FirstS and HasConflict. This is left as an exercise to the reader. In a more generic setting, one could imagine using a “data types à la carte”-approach to solve this extension more satisfactorily.

8. **BENCHMARKS**

To demonstrate the performance impact of dependency tracking, we evaluate several benchmarks with and without dependency-tracking effect handlers and compare their results.

We consider five problems: computing (large) fibonacci numbers, (a variation of) the knapsack problem, the nqueens problem, computing strongly connected components in a directed graph and finding the shortest path in a directed graph.

The benchmarks were build using the Criterion-benchmark harness and compiled using GHC 7.10.2 (with optimization setting -O2) on a 64-bit Linux machine with an Intel Core i7-2600 @ 3.6 Ghz and 16 GB of RAM. All code and results are available in the bitbucket repository.10

Table 2 shows the ordinary least squares regression of all samples of the execution time. The $R^2$ goodness of fit was above 0.99 in all cases. All values are in milliseconds.

For Fibonacci the first column is the index of the fibonacci number that is computed. For Knapsack it specifies the number of items to choose from (the capacity of the knapsack is fixed to 200). For NQueens, it indicates the number of queens that need to be placed on the board. For SCC it is the number of nodes in the graph (the number of edges is fixed to half the number of possible edges). Finally, for Shortest Path it is the number of edges in the graph.

The results show that effect handlers with dependency tracking massively outperform the naive effect handlers.

9. **RELATED WORK**

There are various related works.

**Least fixed point computations** In their seminal work [17] Scott and Strachey have introduced the idea of denotational mathematical semantics. In particular, they already define the semantics of programming language syntax based on least fixed points.10

runNDM return the environment EnEM to directly perform lookups, or embed firstS and hasConflict in NDM as well. Such an embedding requires extending Analyze with new constructors for FirstS and HasConflict. This is left as an exercise to the reader. In a more generic setting, one could imagine using a “data types à la carte”-approach to solve this extension more satisfactorily.

**Effect Handlers** Algebraic effects as a way to write effectful programs are the subject of much contemporary research [27, 10].

New languages have been created specifically for studying algebraic effect handlers: Eff [1] extends ML with syntax constructs for effect handlers and Frank [12] which has a Haskell-like syntax and tracks effect in its type system.

**Non-determinism** The classical introduction of non-determinism in functional programming based on the list monad is due to Watler [24]. Hinze [7] derives a backtracking monad transformer from a declarative specification. This transformer adds backtracking (non-determinism and failure) to any monad.

Kiselyov et al. [11] improve on Hinze’s work by implementing backtracking monad transformers which support fair non-deterministic choice. They present three different implementations: the classical approach based on Streams (i.e. lazy lists), one based on continuation passing style and another one based on control operators for delimited continuations.

**Tabling in Prolog** Since Prolog has native non-determinism, it too suffers from the issues we discussed in the introduction: non-determinism and recursion can interact to cause non-termination even when a finite solution does exist. The semantics of definite logic programs is given by the least fixed point of the immediate consequence operator $T_P$ [23] that strongly resembles the semantics given by $\cdot$ [ ]. Tabling is a technique that produces a different semantics in the same way we produce different semantics for non-determinism. Several Prolog implementations support tabling, such as XSB-Prolog [22, 26], B-Prolog [28], Yap-Prolog [15], ALS-Prolog [5] and Mercury [18]. XSB-Prolog in particular also implements lattice-based answer subsumption [21] which strongly resembles and has partially inspired the techniques of Section 6. Other systems provide similar functionality through mode-directed tabling [16, 29, 6].

**Explicit Recursion** We are not the first to tame unbridled recursion by representing recursive calls explicitly. Devriese and Piessens [3, 4] overcome the limitations of so-called “$\omega$-regular” grammars by representing recursive occurrences of non-terminals in production bodies explicitly.

McBride [13], working in the dependently typed total language Agda 11 gives a free monadic model for recursive calls. Several possible semantics are discussed, but none based on fixed points.

Oliveira and Cook [2] extend traditional algebraic datatypes to with explicit definition and manipulation of cycles, offering a practical and convenient way of programming graphs in Haskell. Their approach to explicating recursion differs from ours. Where we use the smart constructor rec to indicate a recursive call, they instead use open recursion. While this is a good fit for the explicit graph data structure they work with, for our purposes (non-deterministic computations) the former style is more convenient.


10

Jeannin et al. [8] give a category theoretic treatment of “Non-well-Founded computations”. They give many examples, several of which can be implemented in our non-deterministic framework. Others are beyond our iterative least fixed point solver, and require more sophisticated techniques, for instance, Gaussian elimination.

The use of least fixed points for solving grammar analysis problems has been extensively treated by Jeuring and Swierstra [9].

It is a well-known result that dynamic programming can be viewed as memoization of recursive equations. The related work section of Swadi et al. gives an overview [19].

**Explicit Recursion** We are not the first to tame unbridled recursion by representing recursive calls explicitly. Devriese and Piessens [3, 4] overcome the limitations of so-called “$\omega$-regular” grammars by representing recursive occurrences of non-terminals in production bodies explicitly.

McBride [13], working in the dependently typed total language Agda 11 gives a free monadic model for recursive calls. Several possible semantics are discussed, but none based on fixed points.

Oliveira and Cook [2] extend traditional algebraic datatypes to with explicit definition and manipulation of cycles, offering a practical and convenient way of programming graphs in Haskell. Their approach to explicating recursion differs from ours. Where we use the smart constructor rec to indicate a recursive call, they instead use open recursion. While this is a good fit for the explicit graph data structure they work with, for our purposes (non-deterministic computations) the former style is more convenient.

10

Jeannin et al. [8] give a category theoretic treatment of “Non-well-Founded computations”. They give many examples, several of which can be implemented in our non-deterministic framework. Others are beyond our iterative least fixed point solver, and require more sophisticated techniques, for instance, Gaussian elimination.

The use of least fixed points for solving grammar analysis problems has been extensively treated by Jeuring and Swierstra [9].

It is a well-known result that dynamic programming can be viewed as memoization of recursive equations. The related work section of Swadi et al. gives an overview [19].

**Effect Handlers** Algebraic effects as a way to write effectful programs are the subject of much contemporary research [27, 10].

New languages have been created specifically for studying algebraic effect handlers: Eff [1] extends ML with syntax constructs for effect handlers and Frank [12] which has a Haskell-like syntax and tracks effect in its type system.

**Non-determinism** The classical introduction of non-determinism in functional programming based on the list monad is due to Watler [24]. Hinze [7] derives a backtracking monad transformer from a declarative specification. This transformer adds backtracking (non-determinism and failure) to any monad.

Kiselyov et al. [11] improve on Hinze’s work by implementing backtracking monad transformers which support fair non-deterministic choice. They present three different implementations: the classical approach based on Streams (i.e. lazy lists), one based on continuation passing style and another one based on control operators for delimited continuations.

**Tabling in Prolog** Since Prolog has native non-determinism, it too suffers from the issues we discussed in the introduction: non-determinism and recursion can interact to cause non-termination even when a finite solution does exist. The semantics of definite logic programs is given by the least fixed point of the immediate consequence operator $T_P$ [23] that strongly resembles the semantics given by $\cdot$ [ ]. Tabling is a technique that produces a different semantics in the same way we produce different semantics for non-determinism. Several Prolog implementations support tabling, such as XSB-Prolog [22, 26], B-Prolog [28], Yap-Prolog [15], ALS-Prolog [5] and Mercury [18]. XSB-Prolog in particular also implements lattice-based answer subsumption [21] which strongly resembles and has partially inspired the techniques of Section 6. Other systems provide similar functionality through mode-directed tabling [16, 29, 6].

**Explicit Recursion** We are not the first to tame unbridled recursion by representing recursive calls explicitly. Devriese and Piessens [3, 4] overcome the limitations of so-called “$\omega$-regular” grammars by representing recursive occurrences of non-terminals in production bodies explicitly.

McBride [13], working in the dependently typed total language Agda 11 gives a free monadic model for recursive calls. Several possible semantics are discussed, but none based on fixed points.

Oliveira and Cook [2] extend traditional algebraic datatypes to with explicit definition and manipulation of cycles, offering a practical and convenient way of programming graphs in Haskell. Their approach to explicating recursion differs from ours. Where we use the smart constructor rec to indicate a recursive call, they instead use open recursion. While this is a good fit for the explicit graph data structure they work with, for our purposes (non-deterministic computations) the former style is more convenient.
tutions in a free monadic fashion, and have given these descriptions a concise denotational semantics in terms of sets, and more generally in terms of lattices. We have shown how to efficiently implement these semantics in Haskell using Effect Handlers.

11. ACKNOWLEDGEMENTS

We thank Maciej Piróg for pointing out related work, and Georgios Karachalios, Steven Keuchel, Amr Saleh and Paolo Torrini, Maarten Van den Heuvel and the IFL 2015 participants for their helpful feedback. We are grateful to the anonymous reviewers of the IFL 2015 PC committee for their useful comments. This research was partially supported by the Flemish Fund for Scientific Research (FWO). This work was partially supported by project GRACEFUL, which has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 640954.

12. REFERENCES


