The Table Monad in Haskell

Alexander Vandenbroucke
KU Leuven
alexander.vandenbroucke@kuleuven.be

Tom Schrijvers
KU Leuven
tom.schrijvers@kuleuven.be

Frank Piessens
KU Leuven
frank.piessens@kuleuven.be

Abstract

Non-deterministic functions are functions that can return multiple answers. When such a function calls itself recursively it can produce an infinite number of answers where only finitely many distinct results exist. Tabling is a technique that allows the combination of recursion and non-determinism in Prolog.

In this paper we construct the Table Monad, introducing tabling for non-deterministic functions in Haskell. This monad is constructed using the "Effect-Handlers"-approach. A variation on tabling, tabling with aggregates is also discussed. Benchmarks show that for some cases this kind of tabling is more efficient.

Categories and Subject Descriptors D.3 [*Programming Languages*]

Keywords Haskell, Tabling, Effect Handlers, Logic Programming, Non-determinism

1. Introduction

How long is the list of solutions computed by the following Haskell program?

$$
\begin{aligned}
&p :: [(Int, Int)] \\
&p = (1,2) : [(y,x) \mid (x,y) \leftarrow p] \\
&\ggg print\ p \\
&[(1,2),(2,1),(1,2),(2,1),...]
\end{aligned}
$$

How many solutions does the following Tabled Prolog program compute?

```
:- table p/2.
p(1,2).
p(Y,X) :- p(X,Y).

?- p(A,B).
A=1, B=2;
A=2, B=1;
no.
```

The Haskell program runs forever, producing an infinite list of answers, although only two distinct answers exist ((1,2) and (2,1)). In contrast, the Prolog program produces both distinct answers only once. The `table` directive is responsible for this behaviour. It instructs Prolog to *table* the predicate `p/2`.

Tabling [9] is a technique that originated in Logic Programming. It allows programs like the one above to terminate or be computed more efficiently.

Can we informally explain why the Haskell program does not terminate? Intuitively the behaviour of the program is caused by the co-occurrence of two concepts: recursion and non-determinism. Clearly, a program that does not contain recursion cannot run forever. Conversely, a program that can run forever must contain recursion. In this instance, it is the function p that recursively calls itself.

By non-determinism [7] we mean that a function that returns zero, one or more results (as opposed to just one): the function p does not return a single result, but returns a list of all its results (which are infinitely many). The first answer that p produces is $(1,2)$. Then p can recursively derive a new answer $(2,1)$. From $(2,1)$ it derives $(1,2)$ anew, and so on. If we allow unbridled recursion in conjunction with non-determinism, infinite recursive loops can be our reward.

Prolog has inherent non-determinism: resolution of a Prolog goal (such as `?-p(A,B)` in the example) can return multiple answers. Had the Prolog example program not been augmented with the `table` directive, it would have computed the infinite sequence `A=1,B=2;A=2,B=1;A=1;B=2;...` It would suffer from exactly the same problem as the Haskell program.

If tabling solves these issues for Prolog, then it should solve them for non-determinism in Haskell as well.
The main contributions of this paper are:

- An interface for writing recursive non-deterministic programs. It consists of the $Nondet$-typeclass and open recursion. This interface allows enough freedom to implement tabling. In particular, it allows the implementation to prevent infinite recursion (Section 3).
- An implementation of the *Table Monad* that instantiates the $Nondet$-typeclass and deals with open recursion. This implementation is based on the "Effect-Handlers"-approach [10], which factors the discovery of a monad in two tasks:
  1. The abstract syntax, describing all operations that must be supported (Section 4).
  2. The effect handler that maps an abstract syntax tree to an effect, where Tabling is such an effect (Section 5).
- An effect handler that implements a slightly different form of tabling based on aggregates. This allows a much more efficient computation in certain cases (Section 5.3)
- Benchmarks that demonstrate the benefits of aggregates and an effect handler that uses dependency analysis (Section 6).

[Copyright notice will appear here once 'preprint' option is removed.]

2. Overview

To prevent non-deterministic recursion from going awry, we look to Prolog for inspiration. Emulating Prolog was what sparked interest in non-determinism in the first place [2]. By analogy, emulating Tabled Prolog with Tabled Non-determinism should be equally viable.

Non-determinism, as provided by lists, admits a Monad instance:

$$\begin{aligned}&\textbf{instance } \mathit{Monad}\ [\,]\ \textbf{where}\\&\quad \mathit{return}\ a = [a]\\&\quad [\,] \gg\!= f = [\,]\\&\quad (x : xs) \gg\!= f = (f\ x) +\!\!+ (xs \gg\!= f)\end{aligned}$$

This is the standard $Monad$ instance for lists [7]. We want to model tabling with a monad as well. This gives us composable effects. When we have multiple tabled computations, they can be easily composed with monadic bind ($\gg\!=$) and we can layer multiple effects using monad transformers.

Discovering a new monadic structure that also implements tabling as well as providing non-determinism and recursion is a difficult task that requires a lot of creativity and ingenuity. Instead we will use the "Effect Handlers"-approach to create the monad, and "open recursion" to rein in recursion.

Effect Handlers The idea is to decompose the creation of the monad in two phases.

First, an abstract syntax is created. This syntax is in essence an embedded domain specific language (DSL) that allows us to describe all the desired operations. In particular, it should support non-determinism and recursion explicitly. The abstract syntax tree (AST) for such a language always admits an easily defined monad instance.

Second, we define an effect handler (or several handlers) mapping the AST to the desired sequence of effects. In doing so, we have effectively separated the act of coming up with the monad into interface (abstract syntax supporting the operations and monad instance) and implementation (the effect handler).

This process can be done abstractly first: we can invent a grammar that precisely specifies the abstract syntax of the DSL, and we can define a semantic function that gives this syntax a precise meaning. Only then must we deal with the gritty details of implementing this abstract syntax and semantics in Haskell (reasoning about abstract statements where such clutter is absent is often easier).

In the abstract, we are chiefly concerned with correctness: has the desired effect been achieved? This correctness property then carries over into Haskell, where we concern ourselves operational issues such as efficiency. The process is cyclic though: performance concerns that are raised in Haskell are met with formally verified measures, before they are implemented (unless they are trivial).

Programming languages have a concrete syntax and an abstract syntax. Usually the concrete syntax is used to write programs, and abstract syntax is used to interpret or compile the program. Both the concrete and abstract syntax can be written in Haskell. The DSL then becomes an Embedded DSL: written and interpreted entirely within Haskell, as opposed to writing the DSL program in a separate source language.

Different effects can be assigned to the same syntax tree. Consider a semantic function that associates a program with the set of all its distinct solutions. This semantics gives the same solution as Tabled Prolog (modulo unification, negation as failure).

Instead of always putting all solutions in the set, we can be more selective. For instance, we can retain only the optimal solution with respect to some ordering, allowing more efficient implementations of aggregates such as minimum or maximum.

This idea is formalized by generalizing the semantics for sets to any complete lattice. Intuitively, a complete lattice is a partially ordered set where for any subset we can find a smallest element of the set that is greater or equal than all the members of the subset. In particular, sets form a complete lattice under union, and any totally ordered set (for example the integers) can be extended to form a lattice under maximum or minimum.

Open Recursion To gain control over recursion, the program must be rewritten to open up recursion to external influences. The recursion in the example

$$p = (1,2) : [(y,x) \mid (x,y) \leftarrow p]$$

is closed: code outside p cannot influence the recursion. Contrast this with p_{open}:

$$\begin{aligned}&p_{open} :: [(\mathit{Int},\mathit{Int})] \to [(\mathit{Int},\mathit{Int})]\\&p_{open}\ r = (1,2) : [(y,x) \mid (x,y) \leftarrow r]\end{aligned}$$

The function p_{open} takes the recursive call as an explicit argument. Writing $p = \mathit{fix}\ p_{open}$ where fix is the standard Haskell fixed point operator, recovers the original p. By substituting a different fixed point operator we can decide when the recursion stops.

3. *Open* + *Nondet* Programming Interface

Programmers do not usually write in abstract syntax directly. In the same vein, a more convenient interface to open recursion and non-determinism is provided. The intent of open recursion can be more clearly stated using a type synonym:

$$\textbf{type}\ \mathit{Open}\ s = s \to s$$

The example function p then becomes:

$$\begin{aligned}&p_{open} :: \mathit{Open}\ [(\mathit{Int},\mathit{Int})]\\&p_{open}\ r = (1,2) : [(y,x) \mid (x,y) \leftarrow r]\end{aligned}$$

Non-determinism can be captured with the following typeclass:

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

The first function fail indicates a failed computation, i.e. a computation without any results. The $\vee$-operator represents non-deterministic choice between two possible non-deterministic computations (which can make choices in turn). Notice that $Nondet$ is a subclass of $Monad$. As an immediate consequence, programmers may use $\mathit{return}\ a$ to indicate a successful computation witnessed by some result a and $\gg\!=$ to sequence computations. With this notation, the example is:

$$\begin{aligned}&p_{nd} :: \mathit{Nondet}\ m \Rightarrow \mathit{Open}\ (m\ (\mathit{Int},\mathit{Int}))\\&p_{nd}\ r = \mathit{return}\ (1,2) \vee \textbf{do}\ (x,y) \leftarrow r\\&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (y,x)\end{aligned}$$

To verify, we can instantiate $Nondet$ for lists:

$$\begin{aligned}&\textbf{instance } \mathit{Nondet}\ [\,]\ \textbf{where}\\&\quad \mathit{fail} = [\,]\\&\quad (\vee) = (+\!\!+)\end{aligned}$$

then

$$\begin{aligned}&\ggg \mathit{fix}\ p_{nd} :: [(\mathit{Int},\mathit{Int})]\\&[(1,2),(2,1),(1,2),(2,1),\ldots]\end{aligned}$$

exactly like before.

4. Abstract syntax

The data type Table defines the abstract syntax tree of a tabled computation:

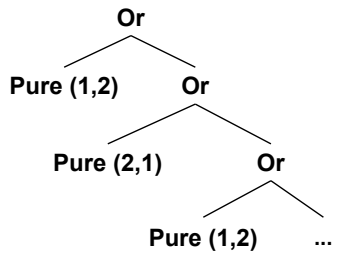


Figure 1. The infinite tree $fix\ p_{nd} :: Table\ p\ b\ c\ (Int, Int)$.

$$
\begin{aligned}
&\textbf{data}\ Table\ p\ b\ c\ a\\
&\quad = Pure\ a\\
&\quad \mid Fail\\
&\quad \mid Or\ (Table\ p\ b\ c\ a)\ (Table\ p\ b\ c\ a)\\
&\quad \mid Call\ p\ b\ (c \to Table\ p\ b\ c\ a)
\end{aligned}
$$

The $Table$ type has 4 arguments: p is the type of the tabled function, b is the argument to a tabled function, c is the type of its result, and a is the type of the result of the entire computation.

Each of the constructors corresponds to a specific operation:

- `Pure a` is a successful computation witnessed by a,
- `Fail` is a failed computation (a computation without any results),
- `Or x y` is a non-deterministic choice between computations x and y,
- `Call f b cont` is a recursive call to f with argument b, and $cont$ is a continuation. A continuation is a function that resumes the computation when it is given the result of the call.

Thinking of $Table$ as syntax tree, the following $Monad$ instance simply substitutes parts of the tree for $\gg\!=$, and wraps a value in a $Pure$ leaf for $return$.

$$
\begin{aligned}
&\textbf{instance}\ Monad\ (Table\ f\ b\ c)\ \textbf{where}\\
&\quad return = Pure\\
&\quad (Pure\ a) \qquad \gg\!= f = f\ a\\
&\quad Fail \qquad\quad \gg\!= f = Fail\\
&\quad a\ `Or`\ b \qquad \gg\!= f = (a \gg\!= f)\ `Or`\ (b \gg\!= f)\\
&\quad Call\ p\ b\ cont \gg\!= f = Call\ p\ b\ (\lambda c \to cont\ c \gg\!= f)
\end{aligned}
$$

The $Nondet$ instance simply uses the corresponding $Table$ constructors:

$$
\begin{aligned}
&\textbf{instance}\ Nondet\ (Table\ f\ b\ c)\ \textbf{where}\\
&\quad fail = Fail\\
&\quad (\vee) = Or
\end{aligned}
$$

The expression $fix\ p_{nd} :: Table\ p\ b\ c\ (Int, Int)$ creates an infinite AST (see Figure 1).

Such an infinite AST is not suitable as input to an effect handler. The infinity is caused by the fix-operator, the this operator substitutes a new infinite tree for the right child of any Or node.

A smarter fixed point operator $close$ can produce a finite AST for open recursive functions:

$$
\begin{aligned}
&\textbf{newtype}\ P\ b\ c = P\ \{unP :: (b \to Table\ (P\ b\ c)\ b\ c\ c)\}\\
&close\ ::\ Open\ (b \to Table\ (P\ b\ c)\ b\ c\ c)\\
&\qquad \to b \to Table\ (P\ b\ c)\ b\ c\ c\\
&close\ o = \textbf{let}\ t\ b = Call\ (P\ (o\ t))\ b\ Pure\ \textbf{in}\ o\ t
\end{aligned}
$$

Instead of simply substituting the function itself, a $Call$ node with a predicate P is substituted. This ties the recursive knot, making the tree finite. However, $p_{nd} :: Nondet\ m \Rightarrow Open\ (m\ (Int, Int))$ does not quite match the required type signature $Open\ (b \to Table\ (P\ b\ c)\ b\ c\ c)$. A type $Open\ (() \to s)$ is isomorphic to $Open\ s$, so p_{nd} is equivalent p_{close}:

$$
\begin{aligned}
&p_{close} :: Nondet\ m \Rightarrow Open\ (() \to m\ (Int, Int))\\
&p_{close}\ c\ () = return\ (1,2) \vee \textbf{do}\ (y, x) \leftarrow c\ ()\\
&\qquad\qquad\qquad\qquad\qquad\qquad return\ (x, y)
\end{aligned}
$$

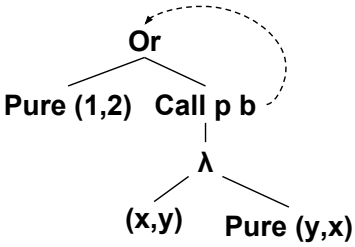


Figure 2. The finite tree $close\ p_{close}\ ()$.

The expression $close\ p_{close}\ ()$ creates a finite tree, displayed in Figure 2. The dashed arrow is used to indicate that Call contains reference to the Or-node in its predicate $P\ b\ c$.

5. Effect Handler

Given an abstract syntax tree, a compiler generates code corresponding to this syntax. The code that the compiler generates can be thought of as the *meaning or semantics* of the abstract syntax. An effect handler assigns a formal semantics to an AST in a similar fashion, but the meaning is represented more abstractly to simplify reasoning about the semantics.

For convenience, denote $Table\ (P\ b\ c)\ b\ c\ a$ by $Table\ A$, and use $p(b)$ to denote $unP\ p\ b$. The effect handler is a function $\mathbb{T}[\![\,.\,]\!] : Table\ A \to \mathcal{P}(A)$, mapping syntax to the set of all its solutions (a subset of A). Owing to the recursive nature of the AST, $\mathbb{T}[\![\,.\,]\!]$ is defined inductively.

base cases For $Pure\ a$ and $Fail$ the set of solutions is the singleton set $\{a\}$ and the empty set $\emptyset$ respectively:

$$
\begin{aligned}
\mathbb{T}[\![\ Pure\ a\ ]\!] &= \{a\}\\
\mathbb{T}[\![\ Fail\ ]\!] &= \emptyset
\end{aligned}
$$

inductive cases The solution $Or\ a\ b$ of is the union of the solutions of its two choices:

$$
\mathbb{T}[\![\ Or\ a\ b\ ]\!] = \mathbb{T}[\![\ a\ ]\!] \cup \mathbb{T}[\![\ b\ ]\!]
$$

How do we compute the set of solutions for $Call\ p\ b\ cont$? Assume that we have a function s that knows all the solutions for all predicates and arguments that are used in $p(b)$, then we could also compute the results for $p(b)$ with the function $\mathbb{C}[\![\,.\,]\!]$ below. The function $\mathbb{C}[\![\,.\,]\!]$ is like $\mathbb{T}[\![\,.\,]\!]$ except that it computes the result of $Call\ q\ k\ cont$ by applying $cont$ on every solution in $s(q,k)$ and taking the union of the results.

$$
\begin{aligned}
&\mathbb{C}[\![\,.\,]\!] : Table\ C \to (P\ B\ C \times B \to \mathcal{P}(C)) \to \mathcal{P}(C)\\
&\mathbb{C}[\![\ Pure\ c\ ]\!](s) = \{c\}\\
&\mathbb{C}[\![\ Fail\ ]\!](s) = \emptyset\\
&\mathbb{C}[\![\ Or\ a\ b\ ]\!](s) = \mathbb{C}[\![\ a\ ]\!](s) \cup \mathbb{C}[\![\ b\ ]\!](s)\\
&\mathbb{C}[\![\ Call\ q\ k\ cont\ ]\!](s) = \bigcup_{c \in s(q,k)} \mathbb{C}[\![\ cont(c)\ ]\!](s)
\end{aligned}
$$

If s contains not just the results for the predicates used in $p(b)$, but *all* predicates, then the result that $\mathbb{C}[\![\,.\,]\!]$ computes must be known to s, i.e. we have $\mathbb{C}[\![\,p(b)\,]\!](s) = s(p,b)$ for all predicates p and arguments b. Then s must be a fixed point of the function $\lambda s.\lambda(q,k).\mathbb{C}[\![\,q(k)\,]\!](s)$. In fact, we can prove[1] that this function has a least fixed point.

$$\begin{aligned}&\text{Let } table : P\ B\ C \times B \to \mathcal{P}(C)\\ &\quad table(p,b) = lfp(\lambda s.\lambda(q,k).\mathbb{C}[\![\,q(k)\,]\!](s))(p,b)\end{aligned}$$

where $lfp(.)$ denotes the least fixed point operator. With the $table$ function we can now define the semantics of a $Call\ p\ b\ cont$:

$$\mathbb{T}[\![\ Call\ p\ b\ cont\ ]\!] = \bigcup_{c \in table(p,b)} \mathbb{T}[\![\ cont(c)\ ]\!]$$

To demonstrate, consider the following least fixed point,

$$s(q,k) = \begin{cases}\{(1,2),(2,1)\} & \text{if q = p and } k = ()\\ \emptyset & \text{otherwise}\end{cases}$$

where p is the predicate called by $close\ p_{close}\ ()$.
Now we have for $\mathbb{T}[\![\ close\ p_{close}\ ()\ ]\!]$:

$$\begin{aligned}\mathbb{T}[\![\ close\ p_{close}\ ()\ ]\!] &= \mathbb{T}[\![\ Call\ p\ ()\ (\lambda(y,x) \to Pure\ (y,x))\ ]\!]\\ &= \bigcup_{(y,x) \in table(p,())} \mathbb{T}[\![\ Pure(x,y)\ ]\!]\\ &= \bigcup_{(y,x) \in s(p,())} \mathbb{T}[\![\ Pure(x,y)\ ]\!]\\ &= \mathbb{T}[\![\ Pure(2,1)\ ]\!] \cup \mathbb{T}[\![\ Pure(1,2)\ ]\!]\\ &= \{(2,1),(1,2)\}\end{aligned}$$

this is exactly the solution found by Tabled Prolog.

5.1 Effect Handler in Haskell

Because $\mathbb{T}[\![\,.\,]\!]$ is essentially a function, and Haskell is a functional programming language, we can easily translate it into Haskell (using Set from `Data.Set` and $foldMap$ from `Data.Foldable`).

```
𝕋 :: (Ord a, Ord c) ⇒ Table (P b c) b c a → Set a
𝕋 (Pure a)          = singleton a
𝕋  Fail             = empty
𝕋 (Or a b)          = 𝕋 a `union` 𝕋 b
𝕋 (Call p b cont)   = foldMap (𝕋 ∘ cont) (table p b)

table :: Ord c ⇒ P b c → b → Set c
table = lfp (λs → λq k → ℂ (unP q k) s)
            (λq k → empty)

ℂ :: Ord c
  ⇒ Table (P b c) b c c
  → (P b c → b → Set c)
  → Set c
ℂ (Pure a)          s = singleton a
ℂ  Fail             s = empty
ℂ (Or a b)          s = ℂ a s `union` ℂ b s
ℂ (Call p b cont)   s = foldMap (flip ℂ s ∘ cont) (s p b)

lfp :: (a → a) → a → a
```

5.2 Dependency analysis

In the previous section, the definition of the least fixed point operator *lfp* was intentionally left undefined. Computing the least fixed point is where different implementations of the same semantics have the most leeway.

The most performant solution (as shown by benchmarks, see later) performs dependency analysis between the predicates. Briefly, the idea is that when the dependencies of a predicate don't change, the predicate doesn't change either. When we iteratively compute the least fixed point, we only run $\mathbb{C}$ on those predicates that have changed dependencies. When there are no predicates with changed dependencies, we have reached a fixed point. This idea has been implemented in Haskell, and the results can be observed in the Section 6. Due to space constraints the implementation itself is elided.

5.3 Aggregates

The semantics $\mathbb{T}[\![\,.\,]\!]$ described in Section 5 associates an abstract syntax tree with the full set of all its solutions. Frequently, we are not interested in *all* solutions but only a select few, e.g. the optimum solution(s).

For some problems, where the solution space is really large (e.g. the Knapsack Problem, the Subset Sum problem), it is worthwile to compute the optimum with a different approach that avoids generating all intermediate results.

Selecting particular elements from a set is exactly the operation that an aggregate (such as `MIN` or `MAX`) accomplishes. The notion of an aggregate as "the worst thing that is is better than all elements in a subset of A" formalized with *Complete Lattices*.

For a more in-depth explantation, see Appendix A

6. Benchmarks

To substantiate the claims in Section 5.2 and Section 5.3, we compare implementations with and without dependency analysis or aggregates for a number of problems. The detailed test results are listed in Appendix B

The first problem is the knapsack problem. This problem is a natural fit for aggregates (see Appendix A). There are three implementations, displayed in Figure 3: *dumb*: dependency analysis, no aggregates; *iter*: aggregates, no dependency analysis; *advanced*: aggregates and dependency analysis. Aggregates and dependency analysis are clearly hugely beneficial.

The second benchmark computes all strongly connected components in a small directed graph (only 20 vertices, with 40, 80 and 160 edges). There are three implementations, displayed in Figure 4: *iter*: no dependency analysis, no aggregates; *advanced*: dependency analyis, no aggregates; *lattice*: dependency analysis, uses the aggregate of all solutions (this aggregate is semantically identical to $\mathbb{T}$. Dependency analysis outperforms *iter*. It seems that the compiler can generate slightly faster code for the specific *advanced*-implementation, compared to the more general *lattice* code.

In the final problem we compute the shortest path between any two nodes in a weighted directed graph. In this case there are four implementation, displayed in Figure 5: *dijkstra*: a straight-forward implementation of Dijkstra's shortest path algorithm, without any tabling; *iter* and *advanced*: both use aggregates, only the latter uses dependency analysis. Dijkstra's algorithm significantly outperforms the other implementations, but the (beneficial) impact of dependency analysis is still clear.

7. Related Work

Pareja-Flores, and Peña and Velázquez-Iturbide [3] discuss a Tabling approach in Haskell that requires manual program transformations, as opposed to our DSL approach.

Wadler[7] introduced "lists of successes" as a way of modelling non-determinism in functional programming, which enspired the simple example used throughout this paper. A later publication [8]

[1] Proof is given in Alexander's master thesis.

by the same author suggests using monads as a way of structuring functional programs. Indeed, the monad instance for lists is already mentioned in this paper. Several more advanced monad-transformers for non-determinism are discussed by Kiselyov, Shan, Friedman and Sabry [2].

A particularly accessible explanation of Effect Handlers is given by Schrijvers and Hinze[10].

Functional logic programming languages, such as Curry[1] exhibit properties of both functional and logical programming languages. Curry, for instance is heavily based on Haskell, supporting almost all Haskell constructs. In addition, it also provides syntactical constructs to return non-deterministic results, or have the system search for non-deterministic solutions for so called "free" variables. Non-deterministic functions in Curry run into the same problem with recursion that is discussed in this paper.

Warren [9] gives a good introduction to Tabled Prolog using the XSB-Prolog system. Several Prolog systems such as XSB-Prolog[6] and YapTab[5] also support aggregates.

8. Conclusion

We demonstrated that there exists a problem when we combine non-determinism with open recursion. We present a solution in the form of Tabling, inspired by Tabled Prolog. Tabling is implemented as the Table monad. This monad is created using the "Effect-Handlers"-approach, where we split this task in abstract syntax and effect handlers.

After defining the abstract syntax, we can achieve different effects with different effect handlers. In particular, we discuss two optimizations of the basic effect handler: dependency analysis and aggregates. The benefit of these optimizations have been demonstrated with a set of benchmarks.

9. Future Work

The implementation of effect handlers still has much room for improvement. The YapTab Prolog system, for instance, supports parallel exectution of Tabling [4]. More fundamental changes are also possible. Probabilistic Tabled Programs are such a possibility. In this case the result of a tabled computation is no longer a set of solutions but a probability distribution on this set.

Acknowledgments

This work was partially supported by the Flemish Fund for Scientific Research (FWO).

References

[1] S. Antoy and M. Hanus. Functional logic programming. *Commun. ACM*, 53(4):74–85, Apr. 2010. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/1721654.1721675`.

[2] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). In *ACM SIGPLAN Notices*, volume 40, pages 192–203. ACM, 2005.

[3] C. Pareja-Flores, R. Peña, and J. A. Velázquez-Iturbide. A tabulation transformation tactic using Haskell arrays. In *Fuji International Workshop on Functional and Logic Programming. World Scientific, Singapur*, pages 211–223, 1995.

[4] R. Rocha, F. Silva, and V. Santos Costa. Yaptab: A tabling engine designed to support parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[5] J. Santos and R. Rocha. Efficient support for mode-directed tabling in the yaptab tabling system. *arXiv preprint arXiv:1301.7680*, 2013.

[6] T. Swift and D. S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *Logics in Artificial Intelligence*, pages 300–312. Springer, 2010.

[7] P. Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

[8] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[9] D. S. Warren. *Programming in Tabled Prolog*, volume 1. `http://www3.cs.stonybrook.edu/~warren/xsbbook/`, 1999.

[10] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In *Haskell 2014*, 2014. URL `/Research/papers/haskell2014.pdf`.

A. Aggregates

We use complete lattices to capture the notion of an aggregate.

DEFINITION 1 (Complete Lattice). *A partially ordered set $(S, \sqsubseteq)$ is a complete lattice $(S, \sqsubseteq, \cup)$ if for every $X \subseteq S$ the least upper bound (l.u.b.) $\sqcup X$ is defined, such that:*

$$\forall z \in S : \sqcup X \sqsubseteq z \iff \forall x \in X : x \sqsubseteq z$$

The l.u.b. of $\emptyset$ is denoted by the symbol $\bot$ ("bottom"), i.e

$$\sqcup \emptyset = \bot$$

In Haskell lattices are captured by the following typeclass:

```
class Lattice l where
  (⊑) ::  l  → l → Bool
  ⊔   :: [l] → l
  ⊥   ::  l
  ⊥ = ⊔ []
```

The program only concerns itself with values of type A, but the end result should be an aggregate (a lattice) of type $L(A)$. Hence we need a way to convert between these two types.

DEFINITION 2. *Consider two functions $\uparrow\colon A \to L(A)$ and $\downarrow\colon L(A) \to \mathcal{P}(A)$:*

```
class Coerce i a where
  ↑ :: i →  a
  ↓ :: a → [i]
```

These functions must satisfy the following laws:

$$\downarrow (\uparrow s) = \{s\}$$
$$\sqcup \{\uparrow s \mid s \in \downarrow l\} = l$$

Now we can define a new semantics $\mathbb{T}_\mathbb{L}$[2]:

```
𝕋𝕃 :: forall l a b c ∘ (Lattice l, Coerce a l, Coerce c l)
   ⇒ Table (P b c) b c a → l
𝕋𝕃 (Pure a)        =↑ a
𝕋𝕃  Fail           = ⊥
𝕋𝕃 (Or a b)        = ⊔ [𝕋𝕃 a, 𝕋𝕃 b]
𝕋𝕃 (Call p b cont) =
  ⊔ [𝕋𝕃 (cont c) | c ←↓ (tableL p b :: l)]

tableL :: (Lattice l, Coerce c l) ⇒ P b c → b → l
tableL = lfp (λs → λq k → ⊔ [s q k, ℂ𝕃 (unP q k) s])
             (λq k → ⊥)

ℂ𝕃 :: (Lattice l, Coerce c l)
   ⇒ Table (P b c) b c c → (P b c → b → l) → l
ℂ𝕃 (Pure c)        s =↑ c
ℂ𝕃  Fail           s = ⊥
```

[2] The `ScopedTypeVariables` extension is required.

$$\mathbb{C}_{\mathbb{L}}\ (Or\ a\ b) \qquad s = \sqcup\ [\mathbb{C}_{\mathbb{L}}\ a\ s, \mathbb{C}_{\mathbb{L}}\ b\ s]$$
$$\mathbb{C}_{\mathbb{L}}\ (Call\ p\ b\ cont)\ s = \sqcup\ [\mathbb{C}_{\mathbb{L}}\ (cont\ c)\ s \mid c \leftarrow\downarrow\ (s\ p\ b)]$$

Let's apply this to the knapsack problem:
An item has a *weight* and a *value*.

$$\textbf{type}\ Knapsack = [Item]$$
$$\textbf{data}\ Item = Item\ \{weight :: Int, value :: Int\}$$
$$\qquad \textbf{deriving}\ (Show, Eq, Ord)$$

In this particular problem we only consider three items: with weight 2 and value 1, with weight 2 and value 2 and with weight 5 and value 20.

$$item :: Nondet\ m \Rightarrow m\ Item$$
$$item = return\ (Item\ 2\ 1)$$
$$\qquad \vee return\ (Item\ 2\ 2)$$
$$\qquad \vee return\ (Item\ 5\ 20)$$

The *knapsack* function is the tabled program that solves the problem: The empty knapsack ([]) is always a solution. Otherwise, for every item, we try to fill a knapsack with the capacity that remains after putting the item in the knapsack.

$$knapsack :: Nondet\ m \Rightarrow Open\ (Int \to m\ Knapsack)$$
$$knapsack\ ks\ w \mid w \leqslant 0 \quad = return\ []$$
$$\qquad \mid otherwise = return\ [] \vee \textbf{do}$$
$$\qquad\quad i \leftarrow item$$
$$\qquad\quad \textbf{if}\ weight\ i > w\ \textbf{then}$$
$$\qquad\qquad fail$$
$$\qquad\quad \textbf{else do}$$
$$\qquad\qquad is \leftarrow ks\ (w - weight\ i)$$
$$\qquad\qquad return\ (i : is)$$

A *Knapsack* is a *Lattice*, and items can be coerced into a knapsack and vice versa.

$$\textbf{instance}\ Lattice\ Knapsack\ \textbf{where}$$
$$\quad k1 \sqsubseteq k2 = v1 < v2 \vee (v1 \equiv v2 \wedge w1 < w2)\ \textbf{where}$$
$$\qquad v1 = sum\ (map\ value\ \ k1)$$
$$\qquad w1 = sum\ (map\ weight\ k1)$$
$$\qquad v2 = sum\ (map\ value\ \ k2)$$
$$\qquad w2 = sum\ (map\ weight\ k2)$$
$$\quad \sqcup = foldr\ lmax\ \bot\ \textbf{where}$$
$$\qquad lmax\ a\ b = \textbf{if}\ a \sqsubseteq b\ \textbf{then}\ b\ \textbf{else}\ a$$
$$\quad \bot = []$$
$$\textbf{instance}\ Coerce\ [Item]\ Knapsack\ \textbf{where}$$
$$\quad \uparrow\!\models id$$
$$\quad \downarrow ks = [ks]$$

The old effect handler $\mathbb{T}$ generates all solutions (and all intermediate solutions), $\mathbb{T}_{\mathbb{L}}$ generates only the optimal ones.

$$\ggg \mathbb{T}_{\mathbb{L}}\ (close\ knapsack\ 5) :: Knapsack$$
$$[Item\ 2\ 2, Item\ 5\ 20]$$
$$\ggg \mathbb{T}\ (close\ knapsack\ 5)$$
$$fromList\ [[], [Item\ 2\ 1], [Item\ 2\ 1, Item\ 2\ 1],$$
$$[Item\ 2\ 1, Item\ 2\ 2], [Item\ 2\ 2], [Item\ 2\ 2, Item\ 2\ 1],$$
$$[Item\ 2\ 2, Item\ 2\ 2], [Item\ 5\ 20]]$$

Every complete lattice forms a commutative idempotent monoid[3]. Some aggregates do not form such a monoid. For instance, the Sum monoid: $mappend\ (Sum\ 1)\ (Sum\ 1) = (Sum\ 2)$ is clearly not idempotent. The List monoid is neither commutative nor idempotent. Hence, we cannot capture every useful aggregate with a complete lattice. Consequently, such aggregates cannot be solved with this method.

[3] With the following defintion: $mempty = \bot$, $mappend\ a\ b = \sqcup\ [a, b]$. Then $\sqcup\ [a, a] = a$ (idempotent) and $\sqcup\ [a, b] = \sqcup\ [b, a]$ (commutative)

B. Benchmark Data

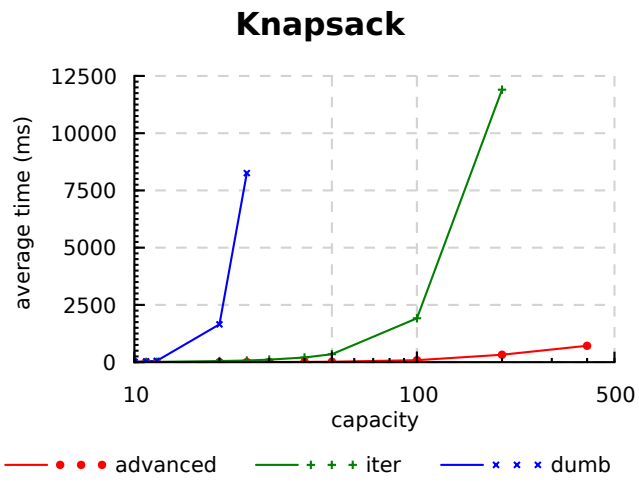


Figure 3. The results of the knapsack benchmark.

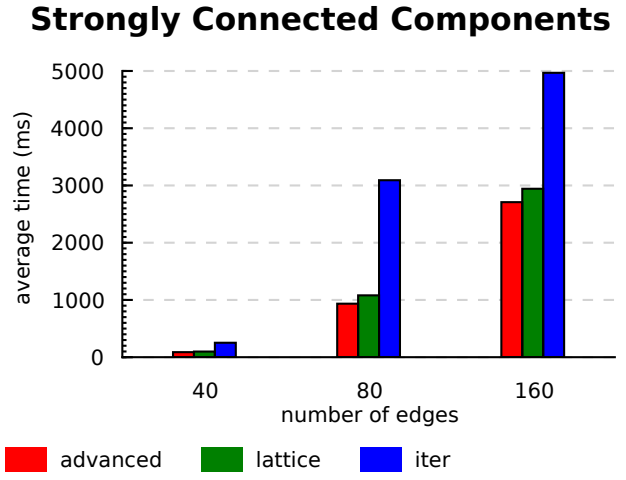


Figure 4. The results of the Strongly Connected Component benchmark.

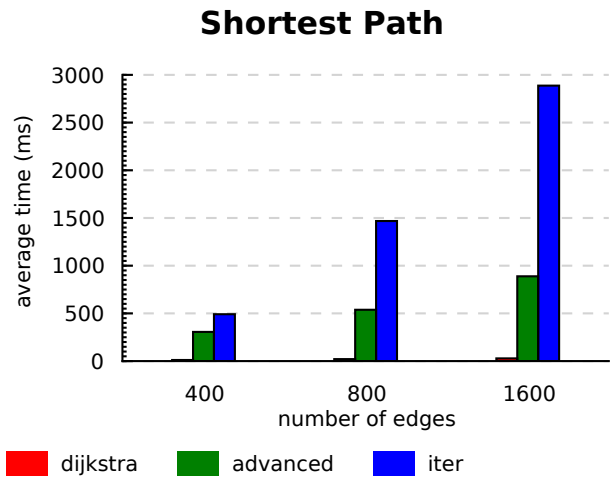


Figure 5. The results of the Shortest Path benchmark.

Capacity	Benchmark	Time	
		avg.	std. dev.
10	iter	5.86 ms	656 μs
	advanced	1.46 ms	13.4 μs
	dumb	15.1 ms	257 μs
11	iter	9.24 ms	1.39 ms
	advanced	1.67 ms	18.2 μs
	dumb	26.6 ms	205 μs
12	iter	10.8 ms	127 μs
	advanced	1.79 ms	27.5 μs
	dumb	45.1 ms	138 μs
20	iter	39.5 ms	187 μs
	advanced	4.51 ms	33.8 μs
	dumb	1.61 s	3.80 ms
25	iter	65.1 ms	610 μs
	advanced	5.79 ms	39.8 μs
	dumb	8.15 s	5.62 ms
30	iter	100 ms	832 μs
	advanced	8.05 ms	43.6 μs
40	iter	187 ms	3.35 ms
	advanced	12.6 ms	61.2 μs
50	iter	316 ms	939 μs
	advanced	19.8 ms	242 μs
100	iter	1.76 s	2.54 ms
	advanced	89.7 ms	1.43 ms
200	iter	10.8 s	25.1 ms
	advanced	238 ms	2.99 ms
400	advanced	699 ms	15.1 ms

Table 1. The results of the knapsack benchmark.

# Edges	Benchmark	Time	
		avg.	std. dev.
40	advanced	88.1 ms	1.38 ms
	lattice	98.5 ms	3.92 ms
	iter	252 ms	2.54 ms
80	advanced	935 ms	16.6 ms
	lattice	1.08 s	59.3 ms
	iter	3.09 s	1.52 ms
160	advanced	2.71 s	15.6 ms
	lattice	2.94 s	6.64 ms
	iter	4.97 s	52.7 ms

Table 2. The results of the Strongly Connected Component benchmark.

# Edges	Benchmark	Time	
		avg.	std. dev.
400	dijkstra	11.6 ms	265 μs
	iter	505 ms	3.80 ms
	advanced	335 ms	2.68 ms
800	dijkstra	21.1 ms	275 μs
	iter	1.53 s	13.6 ms
	advanced	565 ms	3.69 ms
1600	dijkstra	29.5 ms	908 μs
	iter	2.57 s	22.2 ms
	advanced	969 ms	5.07 ms

Table 3. The results fo the Shortest-Path benchmark.