More Fixpoints! (Functional Pearl)

JOACHIM BREITNER, unaffiliated, Germany

Haskell's laziness allows the programmer to solve some problems naturally and declaratively via recursive equations. Unfortunately, if the input is "too recursive", these very elegant idioms can fall into the dreaded black hole, and the programmer has to resort to more pedestrian approaches.

It does not have to be that way: We built variants of common pure data structures (Booleans, sets) where recursive definitions are productive. Internally, the infamous `unsafePerformIO` is at work, but the user only sees a beautiful and pure API, and their pretty recursive idioms – magically – work again.

CCS Concepts: • **Software and its engineering** → **Recursion**; **Functional languages**.

Additional Key Words and Phrases: Haskell, recursion, fixpoint

ACM Reference Format:
Joachim Breitner. 2023. More Fixpoints! (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP, Article 211 (August 2023), 25 pages. https://doi.org/10.1145/3607853

1 INTRODUCTION

Haskell is a pure and lazy programming language, and this laziness allows us to express some algorithms very elegantly, by recursively referring to currently calculated values. A typical and famous example is the following definition of the Fibonacci numbers as an infinite stream:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

This is often called "knot-tying", because a value (here `fibs`) has a definition involving itself.

1.1 Tying the Knot with Graphs

A maybe more practical example is the following calculation of the reflexive transitive closure of a graph, i.e. for each vertex the set of vertices reachable from it. Let us represent a graph as a map from vertices (of type `Int`) to lists of adjacent vertices, which we assume to be in the domain of the map (This keeps the examples concise, as then the lookup operator `M.!` won't fail):

```
import qualified Data.Map as M
import qualified Data.Set as S
type Graph = M.Map Int [Int]
```

The reflexive transitive closure can be very elegantly expressed by knot-tying a map from vertices to their set of reachable vertices:

```
rTrans1 :: Graph -> Graph
rTrans1 g = M.map S.toList reaches
  where
    reaches :: M.Map Int (S.Set Int)
    reaches = M.mapWithKey (\v vs -> S.insert v (S.unions [ reaches M.! v' | v' <- vs ])) g
```

Author's address: Joachim Breitner, mail@joachim-breitner.de, unaffiliated, Freiburg, Germany.

2475-1421/2023/8-ART211
https://doi.org/10.1145/3607853

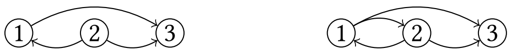

Fig. 1. Two graphs

This code is remarkably close to the prosaic specification "the reachable vertices from a vertex v are v itself, plus all the vertices reachable from any of its successors vs"; hence we can consider this program to be *declarative*.

Note how the definition of reaches refers to itself – we are "tying a knot".

1.2 It Works, Until It Doesn't

This is the kind of code we like to impress our strict-language-using friend with, and it works quite nicely on a small graph (Figure 1, left graph):

```
ghci> rTrans1 (M.fromList [(1,[3]),(2,[1,3]),(3,[])])
fromList [(1,[1,3]),(2,[1,2,3]),(3,[3])]
```

At least until our strict-language-using friend challenges us to add just one small edge to the graph (Figure 1, right graph):

```
ghci> rTrans1 (M.fromList [(1,[2,3]),(2,[1,3]),(3,[])])
fromList [(1,fromList ^CInterrupted.
```

Now the graph has a *cycle* ($1 \longrightarrow 2 \longrightarrow 1$) which makes our code get lost in an infinite loop, until we abort the program.

This is quite disappointing! In order to handle cyclic graphs as input as well, we have to implement this in a much more tedious way, maybe with an explicit loop, keeping track of the set of seen vertices (see Appendix A if you really want to see it, but the goal is that you shouldn't have to). It works, and most of us have likely written that idiom before, but we cannot impress our friend with that.

But it seems it should work: The declarative specification, from which we have derived rTrans1, holds for cyclic graphs as well, so it does not seem too unreasonable to expect the above code to handle cyclic graphs as well. Where does it go wrong? The way we use the lazy map data structure is fine; it helps us to express the set of reachable vertices by way of other such sets. The problem is that the set data structure, with its operations insert and union, is not lazy enough: union wants to know the value of its arguments before it can produce something useful, and thus cannot be used in a recursive, knot-tying way.

1.3 We Need Better Sets!

In this paper we present a data structure for sets, called RSet, where such recursive expressions do work! Its API is almost the same as that of Data.Set, Haskell's standard library for finite sets. In particular, it also consists of plain *pure* functions – no monads necessary. The fragment of the API relevant for our example is:

```
insert  :: Ord a => a -> RSet a -> RSet a
unions  :: Ord a => [RSet a]    -> RSet a
get     ::          RSet a      -> Set a
```

We find the two operations used by rTrans1, with type signatures mirroring those of Data.Set exactly, and a function get that converts such a RSet to a normal Set.

We can use this data structure without changing the structure of our code; we just swap out the operations (imported **qualified as** RS) and convert back to conventional sets in the end:

```
rTrans2 :: Graph -> Graph
rTrans2 g = M.map (S.toList . RS.get) reaches
   where
      reaches :: M.Map Int (RS.RSet Int)
      reaches = M.mapWithKey (\v vs -> RS.insert v (RS.unions [ reaches M.! v' | v' <- vs ])) g
```

And indeed, this *can* handle cyclic graphs, and get the correct result:

```
ghci> rTrans2 (M.fromList [(1,[2,3]),(2,[1,3]),(3,[])])
fromList [(1,[1,2,3]),(2,[1,2,3]),(3,[3])]
```

1.4 Contributions

From the user's point of view, that is almost all there is to say: There is a library of types (sets, Booleans, maps) you can use like the conventional types, and suddenly your favorite knot-tying tricks work even better. In Section 2 we'll explore this library from the user's point of view some more, followed by a larger program analysis case-study in Section 3. Then we take a look at how the library works under the hood (Section 4), and point out some limitations (Section 5). Finally we take a brief glance at related approaches (Section 6).

The main contribution of this paper is to demonstrate that we can make more data types recursively definable, and thus solve more problems elegantly and declaratively. This can be implemented as a regular Haskell library[1], using GHC's unsafePerformIO primitive under the hood, but providing a *safe* and *pure* interface.

2 EXPLORATION

In the introduction we have used a data type RSet with an API that resembles that of the Set data structure in Haskell's Data.Set library. Let us explore this data structure some more from the user's point of view, to get a better understanding of how it is different from the ordinary Set, and to whet the appetite for the look at its implementation in Section 4. Figure 2 gives a comprehensive overview of the API.

2.1 Just an Isomorphic Copy?

At the first glance, RSet looks like an isomorphic copy of Set, with get :: RSet a -> Set a and mk :: Set a -> RSet a converting between the types, and all the operations on RSet behave as their counterpart on Set. Let's quickly check that [Claessen and Hughes 2000]:

```
ghci> quickCheck $ \s -> RS.get (RS.mk s) === s
+++ OK, passed 100 tests.
ghci> quickCheck $ \s1 s2 -> RS.get (RS.union s1 s2) === S.union (RS.get s1) (RS.get s2)
+++ OK, passed 100 tests.
```

The second equation generalizes to all operations in the API, giving them a straight-forward specification in terms of the corresponding operation on the underlying ordinary data type. But there must be a difference, else we would not be writing this paper.

[1]Latest version at https://hackage.haskell.org/package/rec-def, archived artifact at [Breitner 2023]

```
module Data.Recursive.Set where              -- imported as RS here
  data RSet a
  get          ::               RSet a ->              Set a
  mk           ::               Set a ->               RSet a
  empty        ::                                      RSet a
  singleton    ::               a ->                   RSet a
  insert       :: Ord a => a -> RSet a ->              RSet a
  delete       :: Ord a => a -> RSet a ->              RSet a
  union        :: Ord a => RSet a -> RSet a ->         RSet a
  unions       :: Ord a => [RSet a] ->                 RSet a
  intersection :: Ord a => RSet a -> RSet a ->         RSet a
  member       :: Ord a => a -> RSet a ->              RBool
  null         ::               RSet a ->              RDualBool
  when         ::               RBool -> RSet a ->     RSet a
  id           ::               RSet a ->              RSet a

module Data.Recursive.Bool where             -- imported as RB here
  data RBool
  get         :: RBool ->            Bool
  mk          :: Bool ->             RBool
  true, false ::                     RBool
  (&&), (||)  :: RBool -> RBool ->   RBool
  and, or     :: [RBool] ->          RBool
  not         :: RBool ->            RDualBool
  id          :: RBool ->            RBool

module Data.Recursive.DualBool where         -- imported as RDB here
  data RDualBool
  get         :: RDualBool ->                RBool
  mk          :: Bool ->                     RDualBool
  true, false ::                             RDualBool
  (&&), (||)  :: RDualBool -> RDualBool ->   RDualBool
  and, or     :: [RDualBool] ->              RDualBool
  not         :: RDualBool ->                RBool
  id          :: RDualBool ->                RDualBool
```

Fig. 2. The API of recursively definable sets and Booleans

2.2 Recursion!

The difference is that with RSet, *recursively defined expressions work!* For example, using the ordinary Set from Data.Set (imported **qualified as** S), evaluating recursive expressions tends to hang:

```
ghci> let s = S.insert 42 s in s
fromList ^CInterrupted.
```

With RSet, it simply works:

```
ghci> let s = RS.insert 42 s in RS.get s
fromList [42]
```

It works for larger expressions as well

```
ghci> let s = RS.insert 42 (RS.union (RS.insert 23 s) (RS.delete 42 s)) in RS.get s
fromList [23,42]
```

Not even mutual recursion poses a problem:

```
ghci> let s1 = RS.insert 42 s2
ghci|     s2 = RS.insert 23 s3
ghci|     s3 = RS.delete 42 s1
ghci|  in (RS.get s1, RS.get s2, RS.get s3)
(fromList [23,42],fromList [23],fromList [23])
```

In these examples, we build the graph of recursively defined RSet values explicitly, using **let**. In practice one would more likely construct that graph using lazy data structures and knot-tying, maybe dynamically based on some input, as done in the introduction.

2.3 Fixpoints

It may come as a pleasant surprise that these expressions are productive, i.e. that we even obtain a (non-bottom) result. But is it the right result? If we look at the last example above we can see that the ordinary sets we get for each of the three variables makes the three defining equations true:

```
ghci> let s1 = S.fromList [23,42]; s2 = S.fromList [23]; s3 = S.fromList [23]
ghci> s1 == S.insert 42 s2
True
ghci> s2 == S.insert 23 s3
True
ghci> s3 == S.delete 42 s1
True
```

Good! That is how we want equations in a functional programming language to behave.

At this point you might interject that these are not the only possible solutions to this system of equations. Returning to the smaller example of **let** s = RS.insert 42 s, we find that our result S.fromList [42] indeed solves the equation s == S.insert 42 s, but so does S.fromList [42,43]. Still, we would not consider that a “good” solution, and would be surprised if we’d get that. That is because we expect the result to be the **least fixpoint**; the solution that is, among all possible solutions, the smallest with regard to a particular partial order.

In the context of sets the natural order is subset inclusion. Therefore, a possibly recursive expression of RSet values will evaluate to the smallest sets solving the definitional equations.

It will always do so, provided that

- only finitely many RSet values are involved and
- no definition of a rs :: RSet depends on the expression RS.get rs.

Using RS.get drops us in the world of ordinary sets, and the magic disappears:

```
ghci> let s = RS.mk (RS.get (RS.insert 42 s)) in RS.get s
fromList ^CInterrupted.
```

In this sense, RS.mk . RS.get is not the identity function.

2.4 More Than Sets

The library not only provides recursively definable sets, but also other data types, in particular a variant of the ordinary Booleans that are recursively definable; see Figure 2 for an excerpt of the API. Again, we have analogues of the usual operations (literals, conjunction and disjunction), and as before, a possibly recursively defined expression of type RBool will evaluate to "the" Boolean value which solves these defining equations.

What happens if both True and False would solve an equation, like in the following case?

```
ghci> let x = x RB.|| x in RB.get x
False
```

We can see that RBool considers False as the least element, and for some use-cases that is the right choice. But for other use-cases, one would prefer True over False. Therefore, the library provides a separate module and data type RDualBool, again with the full set of operations on Booleans, but this time returning True if possible:

```
ghci> let x = x RDB.|| x in RDB.get x
True
```

2.5 Monotonicity

These data types – RSet, RBool, RDualBool – are not silos, and you will find among the functions in Figure 2 some that connect these types: negation on Booleans, member checks on sets, the emptiness check on sets, and the function when, which guards a set depending on a boolean. This means that even recursive expressions involving multiple of these types will produce a result.

So why does RS.member return a RBool, but RS.null returns a RDualBool? And why is there no function RB.not :: RBool -> RBool? It is because all functions involved here must be *monotonic*: smaller arguments must lead to smaller results. And because the empty set is smaller than non-empty sets, RS.member must return a RBool (where False is smaller than True), but RS.null must go to RDualBool (where True is smaller than False).

If we did not pay attention to this while defining the API, and added non-monotonic functions (like RB.not :: RBool -> RBool), the user would be able to write equations that do not have a solution, such as **let** x = not x, and we would like to statically rule that out.

The underlying bit of theory is the theorem that a monotone function $f : A \longrightarrow A$ on a partially ordered set A with least element $\bot \in A$ and finite height has a unique least fixed point. This is well-known consequence of the Knaster-Tarski theorem (e.g. see [Pottier 2009]), and if the sentence means something to you, you probably already saw it coming. And if it doesn't, it does not matter for reading this paper.

2.6 Termination

Another function from Data.Set that we do not have is map :: Ord b => (a -> b) -> Set a -> Set b. This may be a bit surprising, as this function is perfectly monotonic with regard to the subset relation. But it can cause other problems: Imagine we had it, and wrote

```
let s = RS.insert 0 (RS.map (+1) s) in RS.get s
```

Does this equation have a solution? Clearly the set `s` needs to contain `0`. But then it also needs to contain `1`. And `2`. And so on. So the solution would have to be the set of *all* natural numbers, but that is not something that `Data.Set`, being a data structure of *finite* sets, can represent.

So we cannot allow this function for `RSet` if we want to guarantee a result for every finite recursive expression.

For the theoretically inclined, this is linked to the theorem that if A has finite height, the smallest fixed-point of a monotonic function f can be found by starting with $\bot$ and iterating f. You might be irked that the type `Set a`, ordered by subset inclusion, does not actually have finite height (if `a` is not finite). With the current API (without functions like `map`) for every *finite* `RSet` expression there are only finitely many possible members, and thus the relevant "subtype" has finite height, and all is well again.

It would not be unreasonable, however, to add `map` to the `RSet` API, as it may be quite useful for some applications, and maybe in these applications the equations have a finite solution just fine. If we'd do that, we could no longer *guarantee* termination for all possible expressions (as shown by the example above). But if an expression yields a result, it will be the least fixed point of the defining equation. One can argue that this would be fine for a Haskell library: Haskell programmers are used to dealing with partiality due to non-termination.

2.7 The Black Hole

We said that "all finite, possibly recursive expressions yield a result". Unfortunately, that is not completely true: If a value of type `RSet` is defined to be simply itself, with none of the `RSet` operations involved, it will not work:

```
ghci> let s = s in RS.get s
fromList ^CInterrupted.
```

And it's not for lack of a solution: Clearly the empty set is the least solution to the equation `s == s`.

Because our library is but a library, despite the apparent magic inside (which we will uncover in Section 4), with a definition like `let s = s`, it has no chance to insert its magic. The problem goes away as soon as *any* function from the API is involved in the definition, even if it is semantically the identity:

```
ghci> let x = RS.unions [x] in RS.get x
fromList []
```

This is a little stumbling block when using this library. And while programmers are unlikely to write `let x = x` directly, the same thing can happen when tying the knot via a lazy data structure. In that case, the programmer is advised to insert a semantic identity function in a suitable position; the API provides `RS.id :: RSet a -> RSet a` for that purpose. A programming language that integrates these features first-class could take care of this automatically.

3 CASE STUDY: A PROGRAM ANALYSIS

Before we leave the user's point of view and look under the hood of the library, let us walk through a slightly larger and more realistic use-case. We hope that this example shows that using recursively definable values allows for noticeably more declarative and elegant programs.

3.1 First Without Recursion

Let us consider a small program analysis of a functional language with lazy let-bindings, (mutual) recursion and exceptions. It should determine whether evaluating an expression may throw an (uncaught) expression.

```
type V    = String
data Exp = Var V | Lam V Exp | App Exp Exp | Throw | Catch Exp
         | Let V Exp Exp | LetRec [(V, Exp)] Exp
```

Fig. 3. An AST for a functional language with mutual recursion and exceptions

To set the stage, Figure 3 contains a typical Haskell datatype for its abstract syntax. The LetRec constructor takes a list of declarations (each with a name and definition) and a body; all bound variables are in scope in all the right-hand sides and the body. Variable names bound in Lam, Let and LetRec shadow outer occurrences. (The resemblance to GHC's intermediate language Core [Peyton Jones and Marlow 2002] is certainly not a coincidence.)

Let us ignore LetRec at first, and write our analysis as a simple traversal of the AST:

```
canThrow :: Exp -> Bool
canThrow e = go M.empty e
  where
    go :: M.Map V Bool -> Exp -> Bool
    go env (Var v)       = env M.! v
    go env Throw         = True
    go env (Catch e)     = False
    go env (Lam v e)     = go (M.insert v False env) e
    go env (App e1 e2)   = go env e1 || go env e2
    go env (Let v e1 e2) = go env' e2
      where
        env_bind = M.singleton v (go env e1)
        env'     = M.union env_bind env
```

Our language is lazy, so to determine whether evaluating a variable can throw, we have to carry around an environment of type M.Map V Bool where we remember whether the corresponding right-hand side could throw. Throw and Catch certainly can resp. cannot throw. The analysis isn't higher order, so for lambdas we assume they can throw if their body can throw, and for applications if either subexpression can throw. Finally for Let, we extend the environment with the analysis result of the bound variable's right-hand side and descend.

3.2 Avoiding the Black Hole

So far, so standard. But what about LetRec? Here, the analysis result of each right-hand side depends on the analysis results of all right-hand sides. We can try to simply do what we did in the Let case:

```
    go env (LetRec binds e) = go env' e
      where
        env_bind = M.fromList [ (v, go env' e) | (v,e) <- binds ]
        env'     = M.union env_bind env
```

Note that, crucially, we use the extended environment env' not only for the body, but also for the right-hand sides.

Alas, this does not work: As soon as we try to analyze an expression that uses recursion, we will fall into a black hole. The crux is that Bool is not recursively definable, because its operations (here only disjunction, (||)) are not lazy.

But if we use RBool instead of Bool, it just works:

```
canThrow :: Exp -> Bool
canThrow e = RB.get (go M.empty e)
  where
    go :: M.Map V RBool -> Exp -> RBool
    go env (Var v)       = env M.! v
    go env Throw         = RB.true
    go env (Catch e)     = RB.false
    go env (Lam v e)     = go (M.insert v RB.false env) e
    go env (App e1 e2)   = go env e1 RB.|| go env e2
    go env (Let v e1 e2) = go env' e2
      where
        env_bind = M.singleton v (go env e1)
        env'     = M.union env_bind env
    go env (LetRec binds e) = go env' e
      where
        env_bind = M.fromList [ (v, RB.id (go env' e)) | (v,e) <- binds ]
        env'     = M.union env_bind env
```

All we had to do was to use RBool instead of Bool in the type of the local, use the corresponding operations (RB.true, RB.false and RB.||) and project out to normal Booleans at the end (using RB.get). A slight blemish is that we also had to insert a call to RB.id to not fall over the input LetRec [("x", Var "x")], as explained in Section 2.7.

It is worth noting that the environment (M.Map V RBool) is still a conventional lazy map, *not* some RMap. We use it to tie the knot for the recursive equations on RBool, just as we did in the example in the introduction, but no further magic is needed.

3.3 Using All the Values

The example above calculates only one final result, namely whether the whole expression canThrow. A real compiler pass might want to use the analysis result at each node to update the AST (e.g. remove redundant calls to Catch, or remember analysis results in the AST as GHC would do). To keep the example simple for the paper we do not do that, as the necessary plumbing would obscure the point (see Appendix B for the complete function).

But our library does allow it: we can use RB.get within the function to get the analysis result for *any* subexpression and return a changed AST accordingly, e.g.

```
go :: M.Map V RBool -> Exp -> (RBool, Exp)
...
go env (Catch e)     = (RB.false, new_e)
  where
    (can_throw, e') = go env e
    new_e | RB.get can_throw = Catch e'
          | otherwise        = e'
```

We'd have to be careful that the RBool returned by go does not depend on any decision made based on a boolean that was obtained with RB.get. For such a fused analysis/transformation pass this is typically possible.

3.4 Alternative Approaches

What would we do if we did not have RBool at our disposal? Here are some common options:

- We can perform an explicit fixpoint analysis in the `LetRec` case: Initialize the `env_bind` with all variables mapped to `False`, descend, check if now any analysis result has changed, and if so, re-analyze all of them, until we find a solution.
  Maybe we can be more clever and re-analyze only some of them.
  Maybe some of that logic can be extracted into a suitable fixpoint operator.
  In any case, we would obscure the declarative intent of the code with lower-level bookkeeping.
- If we do that naively, we can run into the problem that in the presence of nested recursive lets, the nested fixpoint iteration comes with exponential complexity.
  In the case where the analysis result is persisted as annotations in the syntax tree anyways, we can address this issue by starting the fixpoint iteration not from bottom, but from the result of the previous outer iteration. GHC does this, as explained in Sergey et al. [2017, Section 6.6], but again at the cost of more plumbing obscuring the code's intent.
- Another approach is to gather the full data flow problem from the *whole* AST, solve it globally (thus also avoiding the problem mentioned in the previous bullet), and then distributing the analysis again to where we need it. This is reminiscent of how a constrained-based type inference algorithm works.
  It is a satisfying to know that the solving algorithm is free from having to follow the syntactic nesting structure of the program, and that the repeated passes of the fixpoint iteration do not require us to traverse and re-analyze the AST over and over. The solver only sees the pure, distilled data-flow equations.
  There are, however, petty practical issues with this approach (representing the equations as data, uniquely naming the cells, the separate passes for collecting the equations and using the results). At this point, one is likely going to hide this bookkeeping in a suitable monad, which can recover some of the lost elegance, but if the code could otherwise be written as pure functions, that is still quite a price to pay – see Section 6.5 for where one such attempt led to.

When our library is applicable, it allows us to retain the concise elegance of the pure code that does not bother with the "how" of solving equations, while under the hood the solver has a comprehensive global view of the problem.

4 UNDER THE HOOD

We hope that by now you are eager to learn how the `RSet` library is implemented. It is a regular Haskell library, without dedicated compiler support nor using compiler plugins. Maybe this sounds impossible, and we agree: The API and specification presented in the previous section *cannot* be implemented in normal, safe, pure Haskell code.[2]

But it can be implemented using *"unsafe"* features; in particular GHC's infamous function `unsafePerformIO :: IO a -> a`, which allows arbitrary side-effects in pure code. Before you turn away in disgust please allow us to quote Peyton Jones et al. [1999] from their publication introducing this primitive:

> However "unsafe" is not the same as "wrong". It simply means that the programmer, not the compiler, must undertake the proof obligation that the program's semantics is unaffected by the moment at which all these side effects take place. [...] So, we regard the primitives of this paper as *the raw material from which experienced systems programmers can construct beautiful abstractions.*

This is our goal; whether the abstraction presented in Section 2 is beautiful is in the eye of the beholder.

[2]At least we believe it is not possible, for reasons we lay out in Section 5.1.

4.1 A Naive Implementation

The core idea behind the implementation can be explained in two simple steps: First, we use an imperative API to declare values, register their relationships and read their values, and then we wrap that in a pure and sufficiently lazy API. We begin by outlining a naive implementation that initially ignores issues of reentrancy-safety, modularity, performance and space-leaks.

4.2 An Imperative Core

A typical *imperative* API to describe and solve a set of recursive equations provides functions to

(1) register the variables, or *cells*, that occur in the equations
(2) define how such a cell is related to other cells
(3) finally read the value of the cells.

To keep the example code small, we focus on just sets and insertion as the only operation, and could imagine an API like the following:

```
data Cell a
newCell       ::                                  IO (Cell a)
defCellInsert :: Ord a => Cell a -> a -> Cell a -> IO ()
getCell       ::              Cell a ->           IO (Set a)
```

A typical use of this API, solving the two mutually related set equations

$$s_1 = \{42\} \cup s_2$$
$$s_2 = \{23\} \cup s_1$$

would be

```
ghci> c1 <- newCell
ghci> c2 <- newCell
ghci> defCellInsert c1 42 c2
ghci> defCellInsert c2 23 c1
ghci> getCell c2
fromList [23,42]
```

At this point, the actual implementation of this API is not that interesting. In a simple implementation a cell would consist of a current value (initialized to the empty set), and a list of cells depending on this value, and then the changes due to calls to `defCellInsert` are propagated through this network until no more changes need to be propagated (Appendix C re-phrases this summary in Haskell). Of course, more sophisticated algorithms may lurk underneath this interface.

4.3 The Pure Wrapping

The more interesting question is how to get from the imperative `Cell` code to the pure `RSet` API, consisting of just `insert :: a -> RSet a -> RSet a` and `get :: RSet a -> Set a`?

Clearly, `insert` must somehow both create a new cell, and define its equation. Furthermore, it has to be lazy in its second argument, else a recursive equation would immediately loop, so it somehow has to defer the call to `defCellInsert` a bit. This leads to the code seen in Figure 4, which we go through in detail:

A value of type `RSet` consists of three fields

- The `Cell a` backing the value we are defining.
- An `IO ()` action, deferred until the value is actually needed. A value is needed if `get` is used on this `RSet`, or on another `RSet` that depends on this one.
- A flag to remember if this deferred action has run already.

```
data RSet a = MkRSet (Cell a) (IO ()) (IORef Bool)

insert :: Ord a => a -> RSet a -> RSet a
insert x r2 = unsafePerformIO $ do
  c1 <- newCell
  done <- newIORef False
  let todo = do
        is_done <- readIORef done
        unless is_done $ do
          writeIORef done True
          let (MkRSet c2 todo2 _) = r2
          defCellInsert c1 x c2
          todo2
  return (MkRSet c1 todo done)

get :: RSet a -> Set a
get (MkRSet c todo _) = unsafePerformIO $ do
  todo
  getCell c
```

Fig. 4. Wrapping an imperative propagator library in a pure way

The function get does not do much: It triggers the todo action, and afterwards returns the current value of the cell. The interesting bits are in the insert function: It creates a new cell to represent the result, and a "done"-flag. It returns these together with a todo-action, which is *not yet run*. Note that the second argument, r2, is *not* looked at yet, so insert is lazy, as required.

The todo action itself uses the flag to ensure it is only run once. Only therein the value r2 is analyzed, and the relationship between the cells is registered. It also runs the todo action of the other cell. This way, a single call to get will recursively trigger the todo actions of all involved values – and the "done"-flags prevent that process from running in cycles.

4.4 Less Naively, Please

This code describes the essence of our idea, and easily generalizes to the other operations of the RSet API. It is, however, naive in a few ways that are worth discussing.

4.4.1 Reentrancy and Thread Safety. The done flag is used to ensure that the todo action is run exactly once. But if get is invoked concurrently, the code above is obviously racy.

Even worse: Because this is run from unsafePerformIO, even in a single-threaded environment we have to worry about reentrancy, as forcing any unevaluated user-provided expression could kick off execution of another call to get.

In our library we address this with careful use of the MVar concurrency primitive [Peyton Jones et al. 1996], and hide this cleanly behind a small abstraction for "possibly recursive IO-thunks" in the System.IO.RecThunk module.

The dejafu test library [Walker and Runciman 2015], which can exhaustively explore all possible interleavings of concurrent code, has proven invaluable when implementing that abstraction: more than once we thought we had finally achieved thread-safety, only to be told that we were not there yet, until we eventually were able make the tests pass.

4.4.2 Modularity. The naive code above supports just one value type, Set, because the underlying imperative propagator library Cell only supports that type. The full library abstracts over the underlying propagator. This way we can have recursively defined values of different types (`RSet a`, `RBool`), operations connecting them (e.g. `member`) and so even allow solving heterogeneously typed sets of equations.

Supporting different propagator libraries also opens the way for important performance optimizations that are specific to various value types. The most generic propagator implementation assumes no structure on its values besides equality, and just keeps propagating changes until the graph has stabilized. But if we can have different propagator implementations for different types, smarter propagator libraries can be written.

For example for Booleans, a cell changes its value at most once, from `False` to `True`. Once it is `True`, it will never change again, and one can drop its connection to other cells (see the `Data.Propagator.P2` module).

Similarly, a propagator library for finite sets can propagate just deltas, instead of always recomputing the sets from its full inputs, to avoid repeating work. (This is not yet implemented in our library, but would be possible without affecting the public API.)

4.4.3 Space leaks. Another, maybe subtle, problem with our pure wrapping of an imperative propagator library is that it can easily lead to space leaks.

Consider the `insert` function. With `let rs2 = RS.insert x rs1` we create a new mutable cell for `rs2`, and tell the mutable cell in `rs1` to notify us of any changes. This means that now somewhere in `rs1` there is a reference to `rs2` – exactly the other way around from what one would expect from the code. This reference prevents resources allocated for `rs2` from being freed while `rs1` is alive.

This causes space leaks, where allocated resources are kept alive longer than expected or needed. As an extreme example, consider this interaction

```
ghci> RS.get (RS.insert 42 RS.empty)
fromList [42]
```

The call to `insert` allocates a new cell and registers it with the cell in `RS.empty`. But `RS.empty` is a *static* value, and will never be garbage collected!

To fix this, we exploit that once we are done calculating a value (either because we query it using `get`, or because it is needed in the calculation of such a value), it will never change any further. So our propagator API allows “freezing” a cell, which drops the references to the dependent cells, and the pure wrapper freezes cells after the value is computed.

5 CAVEATS

The library presents itself with a innocent looking, pure and simple interface (Figure 2), but the implementation is full of side-effects (as seen in Section 4). One might justifiably worry: Is this really pure? Did we create a beautiful abstraction, or are the types a lie?

We believe the library is indeed pure: The exposed API is designed so that a set of equations has a *unique* solution, and since it solves these equations, the nice equational theory one expects from a pure functional program still holds. Yet, there are some caveats worth pointing out.

5.1 Contextual Equivalences and Lambda Lifting

In general in Haskell, if we have a recursive value definition involving an expression `e`, it is ok to turn that into a recursive function definition abstracting over that expression: We can go from

```
let x = … x … e … in x
```

to

```
let x y = … (x y) … y … in x e
```

without changing the program's meaning. This transformation is called lambda lifting.

With our library, this change can now make the difference between termination and non-termination:

```
ghci> let rs = RS.insert 42 rs in RS.get rs
fromList [42]
ghci> let rs y = RS.insert y (rs y) in RS.get (rs 42)
fromList ^CInterrupted.
```

This is not a simple infelicity of the current implementation, but a fundamental limitation: In the original program, our library can get its (magic, `unsafePerformIO`-powered) hands on a *finite* graph of recursively defined values. In the transformed program there is now a recursive function that endlessly creates *new* values, and our library will never see a complete set of equations. It seems that no (reasonable) implementation of the interface in Figure 2 can solve this problem. (This also shows that a pure implementation of that interface indeed does not exist.)

This limitation exists already in Haskell, albeit in a less severe form: The nice and fast knot-tied `fibs` from the introduction becomes horribly inefficient once you turn it into a recursive *function* (`let fibs x y = x : y : zipWith (+) (fibs x y) (tail (fibs x y))`).

In that sense, lambda lifting is only an equivalence as long as we ignore (asymptotic) complexity, and breaking that equivalence may not be as bad as it first seems. In particular, since an optimizing compiler tends to be careful to not worsen the asymptotic complexity, and will not just willy-nilly break sharing, it will not apply transformations that break in the presence of our library.

Sabry [1998, Fact 3.7] points out that merely changing the set of observational equivalences does *not*, in general and on its own, imply that the extended language is no longer pure.

5.2 Equivalence of Sets

The purity of this library rests on the fact that the equations have a unique solution, and therefore it does not matter how the result is calculated. This argument has a weakness in the case of `RSet` because the underlying `Set` type, implemented as a weight-balanced binary search tree [Adams 1992; Nievergelt and Reingold 1972], does not have a unique representation, and a different evaluation order may lead to results that are differently represented. This is only observable by intentionally breaking the abstraction via the `Data.Set.Internal` module, but it is possible.

5.3 Unproven Claims

Our claims above about the safety and purity of the exposed interface are not yet backed by a formal proof. Executing such a proof would first require a formal notion of what "pure" even means, which is not quite clear. Formal criteria have been proposed by Sabry [1998], Longley [1999] and [Hofmann et al. 2010], but they do not easily apply to our setting. Additionally, we'd need a formal framework that allows us to reason about lazy programs with `unsafePerformIO`, `MVar`, laziness and concurrency.

The maybe most similar work are the two mechanized proofs that `runST :: (forall s. ST s a) -> a` is safe [Jacobs et al. 2022; Timany et al. 2018]: The contextual equivalences of the pure language still hold when extending the language with `runST`, and thus the extension is fully abstract. However, their languages are not call-by-need (which is crucial in our case) and not concurrent (which is also relevant), and as discussed in the previous section, our extension preserves some, but not all contextual equivalences.

5.4 Denotational Semantics

We know how to describe pure functional programming languages elegantly using denotational semantics and fixed-point theorems. Domain theory and fixed-point theorems also underpin the data structures presented here. We would therefore expect that our extended language has a nice denotational semantics as well, but it does not seem to be straight forward, for two reasons:

- As discussed in Section 5.1, sharing now affects termination. How can we give a denotational semantics so that it distinguishes tied knots (`let xs = 1 : xs in xs`) from general recursion (`let f x = x : f x in f 1`)?
- We use `let` to describe both recursion in Haskell's domain, e.g. for recursive functions and knot-tied data structures, as well as for the equations to be solved by our library. This entanglement of two seemingly unrelated partial orders so far resisted disentanglement.

Hackett and Hutton [2019] give a denotational semantics with a cost model of lazy evaluation, and in their semantic domain knot-tying can be recognized. We are currently working on a denotational semantics for our library based on that and welcome interested readers to ask us about it.

6 RELATED WORK

The present work draws inspiration from and connects to various directions.

6.1 Fixpoint Solvers and Propagation Networks

Section 4.2 describes the "imperative core" of our library: An underlying library that allows us to declare the cells of a possibly cyclic graph of values with their relations and finds the solution. For the purposes of this paper we can assume this to be a black box, to avoid getting distracted by questions related to the theory and implementation of this black box, as tempting as these questions may be.

All the work in making fixpoint solvers and data-flow analyses efficient is relevant here [Kam and Ullman 1976; Kildall 1973]. An important difference to typical data-flow analyses is that in order to stay pure, we cannot afford to make conservative approximations while solving the equations (e.g. aborting with a safe guess after a certain number of iterations).

Also related is the concept of *propagator network* [Sussman and Radul 2009], although our use-case is a bit simpler, as information only flows in one direction along an edge, and once we queried the value of one vertex, we never add more constraints that would afterwards influence that vertex and invalidate the value we read.

6.2 LVars

Kuper and Newton [2013] introduce *LVars*, concurrent data structures that can be read from and written to in parallel, but thanks to restrictions on writes (monotonically increasing) and reads ("threshold reads"), a deterministic result is guaranteed in the end. Therefore, the concurrent, effectful, monadic code can be executed in a pure context, via `runPar :: (forall s. Par Det s a) -> a`.

There are some similarities to our work (uses `unsafePerformIO` under the hood, relies on monotonicity to guarantee a unique result), but also significant differences: LVars come with a monadic interface while we avoid monads; our library allows to pull out many pure values, while `runPar` discharges the `Par` monad only once; with LVar one can do more than just solve equations. It thus does not seem to be possible to implement our library using just their public, safe interface.

6.3 Shallow Graph Embeddings and Observable Sharing

Claessen and Sands [1999] made sharing observable in Haskell, so that they can very elegantly describe logic circuits in Haskell. They extend pure Haskell with operations that one might dismiss as impure (observable object identity),. They point out that this breaks some equational equivalences of Haskell, but argue that this may be acceptable, given that some of these equivalences are not benign code transformations to begin with, as they can duplicate arbitrary amount of work anyways – a line of reasoning you may recognize from Section 5.1. Gill [2009] proposed a variant with a more focused primitive `reifyGraph` that turns a knot-tied value into an explicit graph data structure.

Both extensions to Haskell are bolder: They break more laws (even referential transparency!) in order to make the structure of the embedded graph observable, as necessary for their poster use-case. With our library, you *cannot* observe the structure of the graph; you only get your hands on the unique solution to the equations. This means it is unsuitable for some use-cases, but also less disruptive.

We could have implemented our library on top of their observable sharing mechanism, reifying the graph of values as a data structure with explicitly named vertices. We chose to do it differently: Haskell's laziness and the guarantee that a binding `let x = unsafePerformIO act` is executed at most once makes sharing of the value `x` observable. For each shared value we can thus create a unique mutable cell, and let them refer to each other. The graph is never actually turned into a (Haskell-level) data structure and the cells are not (explicitly) numbered or named; instead the graph is represented by the pointers on the heap, cells are implicitly identified by their object identity, and the bookkeeping data for each cell is stored within its mutable fields. One advantage is that reading of values at multiple nodes of the (implicit) graph will share the solver's work, instead of re-reifying the graph and re-solving it.

Looking beyond Haskell we want to point out CoCaml [Jeannin et al. 2017], which allows the OCaml programmer to observe the structure of knot-tied (coinductive) data, and even process such data while preserving the sharing (so that a `map` over a cyclic list can return a cyclic list).

6.4 Logic Programming

For the kind of use-cases presented here, logic programming languages like Prolog and Datalog are certainly on their home turf. What we bring to the table is the seamless integration into an existing purely functional language.

6.4.1 Datafun. Particularly close to our work, and straddling the functional and logic programming paradigms, is the *Datafun* language [Arntzenius and Krishnaswami 2016], a pure and total functional programming language generalizing Datalog, which can declaratively express and compute fixed points of monotone maps on semilattices – exactly what we are trying to do. Since they tailor their language around this idea, their type system can recognize *monotonic* function definitions

Datafun comes with a denotational semantics, which we still struggle with. In their case it helped that as a total language, they do not have to worry about arbitrary recursion and non-termination, avoiding the complications mentioned in Section 5.4.

Monotonicity is crucial for the existence and uniqueness of a solution (Section 2.5). We ensure monotonicity by simply restricting the interface (Figure 2) to only expose monotonic functions. This works, but is rather limiting when it comes to higher-order functions. Assume we would want to support recursively-definable finite *maps* as well. A natural order on finite maps is the point-wise ordering. But with that ordering, higher-order operations like `map :: (a -> b) -> RMap k a -> RMap k b` are only monotonic if their argument is monotonic, and Haskell's type system does not allow us to express that constraint. Without these higher-order functions, however, the map API would be quite impoverished, so we do not support finite maps with the point-wise ordering in our library.[3]

[3]You might notice the `Data.Recursive.Map` module in the `rec-def` package. This implements maps with the *discrete* ordering on values, where all higher order function arguments are vacuously monotonic, thus avoiding this issue.

Datafun's type system has a separate function arrow $\xrightarrow{+}$ to characterize monotonic functions, and thus supports this use-case quite well.

6.4.2 Hatafun. Could we have such a function arrow $\xrightarrow{+}$ in Haskell? This idea is explored by Zhang [2020] with *Hatafun*, an experiment embedding Datafun's type system into Haskell. It defines a **newtype** a -+> b = MFun (a -> b) for monotonic functions and uses type-level computation and Higher Order Abstract Syntax in the style of Polakow [2015] to allow safe and flexible definitions of such functions.

For example, the reflexive transitive closure from the introduction can be calculated using Hatafun as follows (code provided by Yihong Zhang):

```
type Graph = M.Map Int (S.Set Int)

rTrans :: Graph -> Graph
rTrans = eval $ lam $ \edge_graph ->
  let reaches = mlam $ \graph ->
        edge_graph `lub` mapWithKey (
          lam $ \k -> lam $ \vs -> insert `app` k `mapp` mbind vs (lam $ \v' -> graph ! v')
        ) graph
  in fix reaches
```

We see that some functions are defined using explicit combinators for monotonic functions (mlam, mapp, fix) instead of using Haskell's syntax (\, juxtaposition, **let**).

Especially the need to thread recursion through a single fix is significant here: The problem has to be described as a *single* equation in a suitable partial order. So here, the map of reachable nodes is repeatedly recalculated as a whole, until a fixed-point is found.

Contrast that to our code, where only a single map exists, and serves to declare *multiple* equations in the partial order of sets, using knot-tying. The fixed-point iteration then happens on these sets, which can be significantly more efficient.

The main aim of Hatafun is to explore an embedding of datafun's type system for monotonic functions in Haskell, using the HOAS approach, and our library could well benefit from that approach. This is rather orthogonal to our idea of declaring mutually recursive equations via Haskell's **let**, which might be useful for Hatafun, too.

Appendix D.1 contains an Hatafun implementation of our case study.

6.5 Data Flow Libraries in Haskell

Graf [2021] also observes that compiler analyses implemented in pure functional languages tend to mix specifying the data flow equations with the actual solving, and that ideally solving ought to be independent of the syntactic nesting structure of the input program, echoing our concerns in Section 3.4. His *datafix* library explores a monadic solution and can handle our case study, but only with significant refactoring:

- The Exp data type needs to be re-phrased using open recursion:

```
data ExpF r = VarF V | LamF V r | AppF r r | ThrowF | CatchF r | LetRecF [(V, r)] r
data Exp = Fix (ExpF Exp)
```

- A generic traversal for this data type needs to be written. It can then be used for all analyses and passes over Exp.
- Then our actual analysis/transformation pass can be implemented. The snippet corresponding to Section 3.3 would be

```
transferFunctionAlg :: TransferAlgebra RWE
transferFunctionAlg _ _ env e = case e of
  ...
  CatchF e -> do
    MkRWE can_throw e <- e
    let new_e | RTrue <- can_throw = Fix $ CatchF e
              | otherwise          = e
    pure (MkRWE RFalse new_e)
```

where RWE is a tuple of a Boolean and an expression.

The complete code, provided by Sebastian Graf, is listed in Appendix D.2.

Compared to our library, datafix provides more flexibility: It is not restricted to equations and solving strategies that lead to a unique solution, independent of the evaluation strategy, but is at liberty to pragmatically make conservative approximations. Also, the non-recursive description of passes based on an openly-recursive data type may enable the fusion of consecutive compiler passes.

However, it requires significant refactoring (changing the expression data type, for example), and one has to depart from the arguably elegant purely functional style that we started with.

7 CONCLUSION AND FURTHER WORK

We saw that we can extend Haskell with the ability to solve recursive equations involving Booleans and sets, that this extension can be implemented as a library, and that it – arguably – preserves the nice properties of the language.

Of course there is more to be done. On the practical side there are more data types and operations to be included in the library (natural numbers in various ordering, maps). On the theoretical side, a more rigorous, formal approach to the question of whether this is actually safe and pure is worth pursuing, as is finding a denotational description of what the library does.

ACKNOWLEDGMENTS

This paper would be worse without the constructive feedback from Sebastian Graf, Claudio Russo, Andreas Lochbihler, Peter Thiemann as well as the anonymous reviewers. We also thank Yihong Zhang and Sebastian Graf for the code snippets in the related work section.

A THE REFLEXIVE TRANSITIVE CLOSURE, PEDESTRIAN-STYLE

In the introduction we mentioned that without the data structure presented in this paper, a programmer likely has to write their reflexive-transitive-closure code with an explicit loop (a tail-recursive go function), explicitly keeping track of seen vertices to avoid running in circles:

```
rTrans3 :: Graph -> Graph
rTrans3 g = M.fromList [ (v, S.toList (go S.empty [v])) | v <- M.keys g ]
  where
    go :: S.Set Int -> [Int] -> S.Set Int
    go seen [] = seen
    go seen (v:vs) | v `S.member` seen = go seen vs
                   | otherwise         = go (S.insert v seen) (g M.! v ++ vs)
```

B FUSED PROGRAM ANALYSIS AND TRANSFORMATION

In Section 3.3 we pointed out that one advantage of our library is that it not only allows defining mutually recursive equations, but (lazily) query that graph as we go. In our example, we rewrite Catch e to e when the analysis proves that e cannot throw. The full code of the analysis and transformation pass is a bit unwieldy, as it returns both the analysis result and the modified code, so we put it into this appendix.

```
removeCatch :: Exp -> Exp
removeCatch e = snd (go M.empty e)
  where
    go :: M.Map V RBool -> Exp -> (RBool, Exp)
    go env (Var v)            = (env M.! v, Var v)
    go env Throw              = (RB.true, Throw)

    go env (Catch e)          = (RB.false, new_e)
      where
        (can_throw, e')      = go env e
        new_e | RB.get can_throw = Catch e'
              | otherwise          = e'

    go env (Lam v e)          = (can_throw, Lam v e')
      where
        env'                 = M.insert v RB.false env
        (can_throw, e')      = go env' e

    go env (App e1 e2)        = (can_throw1 RB.|| can_throw2, App e1' e2')
      where
        (can_throw1, e1') = go env e1
        (can_throw2, e2') = go env e2

    go env (Let v e1 e2)      = (can_throw2, Let v e1' e2')
      where
        (can_throw1, e1') = go env e1
        env_bind             = M.singleton v can_throw1
        env'                 = M.union env_bind env
        (can_throw2, e2') = go env' e2

    go env (LetRec binds e) = (can_throw, LetRec binds' e')
      where
        (env_bind, binds') = unzip (map (goBind env') binds)
        env'                 = M.union (M.fromList env_bind) env
        (can_throw, e')      = go env' e

    goBind :: M.Map V RBool -> (V,Exp) -> ((V,RBool), (V,Exp))
    goBind env (v,e) = ((v, RB.id can_throw), (v, e'))
      where
        (can_throw, e')      = go env e
```

C THE IMPERATIVE CORE'S IMPLEMENTATION

In Section 4.2 we assumed a module exporting the following API:

```
data Cell a
newCell       ::                                       IO (Cell a)
defCellInsert :: Ord a => Cell a -> a -> Cell a -> IO ()
getCell       ::              Cell a ->                IO (Set a)
```

A simple implementation could look like this:

```
module Cell (Cell, newCell, defCellInsert, getCell) where

import Control.Monad (join, unless)
import Control.Concurrent.MVar
import qualified Data.Set as S

data Cell a = MkCell { val :: MVar (S.Set a), onChange :: MVar (IO ()) }

newCell :: IO (Cell a)
newCell = do
        m <– newMVar S.empty
        notify <– newMVar (pure ())
        pure $ MkCell m notify

getCell :: Cell a –> IO (S.Set a)
getCell (MkCell m _) = readMVar m

setCell :: Eq a => Cell a –> S.Set a –> IO ()
setCell (MkCell m notify) x = do
        old <– swapMVar m x
        unless (old == x) $ join (readMVar notify)

watchCell :: Cell a –> IO () –> IO ()
watchCell (MkCell m notify) act = modifyMVar_ notify (\a –> pure (act >> a))

defCellInsert :: Ord a => Cell a –> a –> Cell a –> IO ()
defCellInsert c1 x c2 = watchCell c2 update >> update
    where
        update = do
            s <– getCell c2
            setCell c1 (S.insert x s)
```

D THE CASE STUDY IN RELATED WORK

To complete our comparison with related work, in particular Hatafun (Section 6.4.2) and datafix (Section 6.5), we show how the case study from Section 3 can be solved with these libraries. This is not an explanation of these libraries, but we hope that the listings can still give a useful impression

The implementations were provided by the authors of the respective libraries, for which we are grateful. The following listings have been modified from their original version. They have been formatted to fit this paper.

D.1 Program Analysis Using Hatafun

Yihong Zhang ported the code from Section 3 as follows:

```
type V = String
type Env = M.Map V Bool
data Exp = Var V | Lam V Exp | App Exp Exp | Throw | Catch Exp
           | Let V Exp Exp | LetRec (M.Map V Exp) Exp

canThrow :: Exp –> Bool
canThrow e = eval (go `mapp` lift M.empty `app` lift e)

go :: Defn (M.Map V Bool -+> (Exp –> Bool))
go = mlam $ \env –> lam $ \x –> case unlift x of
```

```
    Var v -> when (mapMember `app` lift v `mapp` env)
                  (env ! lift v)
    Throw -> tt
    Catch e -> ff
    Lam v e -> go `mapp` lub (lift $ M.singleton v False) env `app` lift e
    App e1 e2 -> (go `mapp` env `app` lift e1) `lub` (go `mapp` env `app` lift e2)
    Let vUnsafe e1Unsafe e2Unsafe -> go `mapp` env' `app` e2
      where
        v :: Defn V
        v = lift vUnsafe
        e1 :: Defn Exp
        e1 = lift e1Unsafe
        e2 :: Defn Exp
        e2 = lift e2Unsafe

        env_bind = mapSingleton `app` v `mapp` (go `mapp` env `app` e1)
        env' = lub env_bind env
    LetRec bindsUnsafe eUnsafe -> go `mapp` env' `app` e
      where
        e :: Defn Exp
        e = lift eUnsafe
        binds :: Defn (M.Map V Exp)
        binds = lift bindsUnsafe

        step :: Defn (Env -+> Env)
        step = mlam $ \env ->
          let env_bind = mapWithKey (lam $ \v -> lam $ \e -> go `mapp` env `app` e) binds
          in  lub env env_bind
        env' = fix step
```

He explained that the use of unlift on the expression to be analyzed is unsafe, but ok here, because the expression is invariant here; correctly tracking the monotonicity of user-defined data types like Exp is possible, but requires auxiliary safe API definitions, eventually may generated generically.

More relevant is that, similar as in Section 6.4.2, this does not set up a system of mutually recursive equations of boolean values, with one variable per AST node, but at each LetRec it declares and solves a single equation for the environment as a whole. With nested LetRecs, this can blow up the complexity, as explained in Section 3.4, which brings us to the next alternative worth considering.

D.2 Program Analysis Using Datafix

The datafix library mentioned in Section 6.5 was created in particular due to the desire to free the equations to be solved from the syntactic structure of the input program. Sebastian Graf shows how the case study can be implemented using datafix, and went straight for the harder problem of a combined analysis/transformation pass, corresponding to Appendix B.

```
import Datafix
import Data.Proxy (Proxy (..))
import qualified Data.Map as M
import Datafix.Utils.SemiLattice (JoinSemiLattice (..), BoundedJoinSemiLattice (..))

type V = String
data ExpF r = VarF V | LamF V r | AppF r r | ThrowF | CatchF r | LetRecF [(V, r)] r
data Exp = Fix (ExpF Exp)

data RBool = RFalse | RTrue deriving (Eq, Ord, Show)
instance JoinSemiLattice RBool where
  (\/) = max
instance BoundedJoinSemiLattice RBool where
  bottom = RFalse

data RWE = MkRWE !RBool Exp
instance Eq RWE where
  MkRWE b1 _ == MkRWE b2 _ = b1 == b2
instance JoinSemiLattice RWE where
  MkRWE b1 _ \/ MkRWE b2 e2 = MkRWE (b1 \/ b2) e2
instance BoundedJoinSemiLattice RWE where
  bottom = MkRWE bottom undefined

type TransferAlgebra lattice = forall m. Monad m =>
  Proxy m -> Proxy lattice -> M.Map V (m lattice) -> ExpF (m lattice) -> m lattice

type TF m = m (Domain m)

buildDenotation :: forall domain. Eq domain => IsBase domain ~ True =>
  TransferAlgebra domain -> Exp -> Denotation domain domain
buildDenotation alg exp = go
  where
    go :: forall m. MonadDatafix m => domain ~ Domain (DepM m) => m (TF (DepM m))
    go = buildDenotation' alg exp

buildDenotation' :: forall domain m. MonadDatafix m => domain ~ Domain (DepM m) =>
  Eq domain => IsBase domain ~ True => TransferAlgebra domain -> Exp -> m (TF (DepM m))
buildDenotation' alg' = buildExpr M.empty
  where
    alg = alg' (Proxy :: Proxy (DepM m)) (Proxy :: Proxy domain)
    buildExpr :: M.Map V (TF (DepM m)) -> Exp -> m (TF (DepM m))
    buildExpr !env (Fix expr) =
      case expr of
        VarF id_ -> pure (alg env (VarF id_))
        LamF id_ body -> do
          transferBody <- buildExpr (M.insert id_ (pure bottom) env) body
          pure (alg env (LamF id_ transferBody))
        AppF f a -> do
          transferF <- buildExpr env f
          transferA <- buildExpr env a
          pure (alg env (AppF transferF transferA))
        ThrowF -> pure (alg env ThrowF)
        CatchF e -> do
          transferE <- buildExpr env e
          pure (alg env (CatchF transferE))
        LetRecF bind body -> do
          (env', transferredBind) <- datafixBindingGroup env bind
          transferBody <- buildExpr env' body
          pure (alg env (LetRecF transferredBind transferBody))
        datafixBindingGroup !env binders = case binders of
          [] -> pure (env, [])
          ((id_, rhs):binders') ->
            datafixEq $ \self -> do
              let env' = M.insert id_ self env
              (env'', transferredBind) <- datafixBindingGroup env' binders'
              transferRHS <- buildExpr env'' rhs
              pure ((env'', (id_, self):transferredBind), transferRHS)

  transferFunctionAlg :: TransferAlgebra RWE
  transferFunctionAlg _ _ env e = case e of
    VarF id_ -> do
      MkRWE throws _ <- env M.! id_
      pure (MkRWE throws (Fix $ VarF id_))
    ThrowF -> pure (MkRWE RTrue (Fix $ ThrowF))
    CatchF e -> do
      MkRWE can_throw e <- e
      let new_e | RTrue <- can_throw = Fix $ CatchF e
                | otherwise           = e
      pure (MkRWE RFalse new_e)
    LamF id_ body -> do
      MkRWE can_throw body <- body
      pure (MkRWE can_throw (Fix $ LamF id_ body))
    AppF f a -> do
      MkRWE throw_f f <- f
      MkRWE throw_a a <- a
      let RFalse ||| b = b
          RTrue  ||| _ = RTrue
      pure (MkRWE (throw_f ||| throw_a) (Fix $ AppF f a))
    LetRecF bind body -> do
      MkRWE can_throw body <- body
      let extract_rhs (v,me) = do
              MkRWE _can_throw e <- me
              pure (v, e)
      bind' <- mapM extract_rhs bind
      pure (MkRWE can_throw (Fix $ LetRecF bind' body))

  removeCatch :: Exp -> Exp
  removeCatch e =
    let MkRWE _ e' = evalDenotation @RWE (buildDenotation transferFunctionAlg e) NeverAbort
    in e'
```

See Section 6.5 for a discussion of this approach.

REFERENCES

Stephen Adams. 1992. *Implementing sets efficiently in a functional language,*. Research Report CSTR 92-10. University of Southampton.

Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: a functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 214–227. https://doi.org/10.1145/2951913.2951948

Joachim Breitner. 2023. *Reproduction package for Functional Pearl "More Fixpoints!"*. https://doi.org/10.1145/3580399

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

Koen Claessen and David Sands. 1999. Observable Sharing for Functional Circuit Description. In *Advances in Computing Science - ASIAN'99, 5th Asian Computing Science Conference, Phuket, Thailand, December 10-12, 1999, Proceedings (Lecture Notes in Computer Science, Vol. 1742)*, P. S. Thiagarajan and Roland H. C. Yap (Eds.). Springer, 62–73. https://doi.org/10.1007/3-540-46674-6_7

Andy Gill. 2009. Type-safe observable sharing in Haskell. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, Stephanie Weirich (Ed.). ACM, 117–128. https://doi.org/10.1145/1596638.1596653

Sebastian Graf. 2021. datafix: Fixing data-flow problems. https://github.com/sgraf812/datafix revision 4d5239c.

Jennifer Hackett and Graham Hutton. 2019. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.* 3, ICFP (2019), 114:1–114:23. https://doi.org/10.1145/3341718

Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. 2010. What Is a Pure Functional?. In *Automata, Languages and Programming, 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 6199)*, Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis (Eds.). Springer, 199–210. https://doi.org/10.1007/978-3-642-14162-1_17

Koen Jacobs, Dominique Devriese, and Amin Timany. 2022. Purity of an ST monad: full abstraction by semantically typed back-translation. *Proc. ACM Program. Lang.* 6, OOPSLA1 (2022), 1–27. https://doi.org/10.1145/3527326

Jean-Baptiste Jeannin, Dexter Kozen, and Alexandra Silva. 2017. CoCaml: Functional Programming with Regular Coinductive Types. *Fundam. Informaticae* 150, 3-4 (2017), 347–377. https://doi.org/10.3233/FI-2017-1473

John B. Kam and Jeffrey D. Ullman. 1976. Global Data Flow Analysis and Iterative Algorithms. *J. ACM* 23, 1 (1976), 158–171. https://doi.org/10.1145/321921.321938

Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973*, Patrick C. Fischer and Jeffrey D. Ullman (Eds.). ACM Press, 194–206. https://doi.org/10.1145/512927.512945

Lindsey Kuper and Ryan R. Newton. 2013. LVars: lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, Boston, MA, USA, FHPC@ICFP 2013, September 25-27, 2013*, Clemens Grelck, Fritz Henglein, Umut A. Acar, and Jost Berthold (Eds.). ACM, 71–84. https://doi.org/10.1145/2502323.2502326

John Longley. 1999. When is a Functional Program Not a Functional Program?. In *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP '99), Paris, France, September 27-29, 1999*, Didier Rémy and Peter Lee (Eds.). ACM, 1–7. https://doi.org/10.1145/317636.317775

Jürg Nievergelt and Edward M. Reingold. 1972. Binary Search Trees of Bounded Balance. In *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*, Patrick C. Fischer, H. Paul Zeiger, Jeffrey D. Ullman, and Arnold L. Rosenberg (Eds.). ACM, 137–142. https://doi.org/10.1145/800152.804906

Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. 1996. Concurrent Haskell. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 295–308. https://doi.org/10.1145/237721.237794

Simon L. Peyton Jones and Simon Marlow. 2002. Secrets of the Glasgow Haskell Compiler inliner. *J. Funct. Program.* 12, 4&5 (2002), 393–433. https://doi.org/10.1017/S0956796802004331

Simon L. Peyton Jones, Simon Marlow, and Conal Elliott. 1999. Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell. In *Implementation of Functional Languages, 11th International Workshop, IFL'99, Lochem, The Netherlands, September 7-10, 1999, Selected Papers (Lecture Notes in Computer Science, Vol. 1868)*, Pieter W. M. Koopman and Chris Clack (Eds.). Springer, 37–58. https://doi.org/10.1007/10722298_3

Jeff Polakow. 2015. Embedding a full linear Lambda calculus in Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 177–188. https://doi.org/10.1145/2804302.2804309

François Pottier. 2009. Functional Pearl: Lazy least fixed points in ML. http://cambium.inria.fr/~fpottier/publis/fpottier-fix.pdf (unpublished).

Amr Sabry. 1998. What is a Purely Functional Language? *J. Funct. Program.* 8, 1 (1998), 1–22. https://doi.org/10.1017/s0956796897002943

Ilya Sergey, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Joachim Breitner. 2017. Modular, higher order cardinality analysis in theory and practice. *J. Funct. Program.* 27 (2017), e11. https://doi.org/10.1017/S0956796817000016

Gerald Jay Sussman and Alexey Radul. 2009. *The Art of the Propagator.* Technical Report MIT/CSAIL Technical Report MIT-CSAIL-TR-2009-002. Massachusetts Institute of Technology, Cambridge, MA. https://dspace.mit.edu/handle/1721.1/44215

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. *Proc. ACM Program. Lang.* 2, POPL (2018), 64:1–64:28. https://doi.org/10.1145/3158152

Michael Walker and Colin Runciman. 2015. Déjà Fu: a concurrency testing library for Haskell. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 141–152. https://doi.org/10.1145/2804302.2804306

Yihong Zhang. 2020. Hatafun. https://github.com/yihozhang/Hatafun/ revision c372b81.

Received 2023-03-01; accepted 2023-06-27