Functional Pearl: More fixpoints!

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. Functional Pearl: More fixpoints!. In . ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

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 :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

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

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


ICFP'23, September 04–09,2023, Seattle, WA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/XXXXXXX.XXXXXXX

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:

```
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:

```
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 recursive 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 we should: The declarative specification, from which we have derived `rTrans1`, holds for recursive graphs as well, so it does not seem too unreasonable to expect the above code to handle recursive 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 `Set` from `Data.Set`. 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 recursive 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, 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), which calls for a discussion of whether calling this *pure* is actually warranted (Section 5). Finally we take a brief glance at related approaches (Section 6).

The main contributions of this paper are:

- We demonstrate that by making more data types recursively definable, more problems can be solved elegantly and declaratively. They are *safe*, *pure* and are a natural fit for a language like Haskell.
- We show how to implement this as a regular library, using GHC's `unsafePerformIO` primitive under the hood.[1]
- We discuss the difficulties of answering the question whether a language extension like this can still be considered *pure*, contributing questions rather than conclusive answers.

ACKNOWLEDGMENTS

We like to thanks Sebastian Graf and Claudio Russo for their helpful comments.

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 vanilla `Set`, and to whet the appetite for the look at its implementation in Section 4. Figure 1 gives a comprehensive overview of the API.

2.1 Just an isomorphic copy?

At the first glance, `RSet` looks like a 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 vanilla data type. But there must be a difference, else we would not be writing this paper.

2.2 Recursion!

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

[1] The library can be found as `rec-def` on Hackage; an anonymized copy is included in this submission.

```
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 ->                 Bool
  mk           :: Bool ->                      RDualBool
  true, false  ::                              RDualBool
  (&&), (||)   :: RDualBool -> RDualBool ->    RDualBool
  and, or      :: [RDualBool] ->               RDualBool
  not          :: RDualBool ->                 RBool
  id           :: RDualBool ->                 RDualBool
```

Fig. 1. The API of recursively definable sets and Booleans

```
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 result. But is it the right result? If we look at the last example above we can see that the vanilla 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
```

That is good, because 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 vanilla set, 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

Our library of recursively definable values does not only provide sets, but also other data types, in particular Booleans, as seen in Figure 1. Again, we have a copy of the usual operations (literals, conjunction and disjunction), and as before, we expect a (possibly recursive) expression of RBool values to evaluate to the Boolean that solves these equation.

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 proves 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 1 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 null returns a RDualBool? And why is there no function 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 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 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 $f : A \to A$ 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 (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.

This also explains why some functions from the underlying vanilla data type (such as Data.Set's difference) are not available, as they are not monotonic.

2.6 Termination

Another function from Data.Set that we do not have in Data.Recursive.Set is the function 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 plays into the "A has finite height" requirement in the theorem above. 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 finite 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 equation 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 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, as Haskell programmers are used to programming around non-termination anyways.

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 :: Set a -> Set 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 conclude the section with a slight 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 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. 2. An AST for a functional language with mutual recursion and exceptions

To set the stage, Figure 2 contains a typical Haskell datatype for its abstract syntax. The `LetRec` constructor takes a list of bound variables along with their definition's right-hand side, 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.

3.3 Results everywhere

To keep it simple for this paper the analysis does not update the AST (e.g. remove redundant calls to `Catch`, or remember analysis results in the AST as GHC would do), as the necessary plumbing would obscure our point. But it would work: 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 Alternatives

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 found 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 thought that this approach frees the solving algorithm from having to follow the syntactic nesting structure of the program, and that the repeated passes of the fixpoint iteration do not have to traverse and re-analyze the AST over and over. Instead is 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.[2]

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.[3]

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.

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.

[2] See https://hackage.haskell.org/package/datafix-0.0.1.0/docs/Datafix-Denotational.html#v:datafix for an attempt.
[3] At least we believe it is not possible, as we explain in Section 5.6.

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*, (2) define their relationships and (3) finally read their values. 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 a mutually recursive set equation, could 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 B 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 3, 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 the `RSet`, or on another `RSet` that depends on this one.
- A flag to remember if this deferred action has run already.

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 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.

```
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. 3. Wrapping an imperative propagator library in a pure way

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` API.

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 still far off.

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 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 `rs2 = RS.insert x rs1` we create a new mutable cell for `rs2`, and tell the mutable cell in `rs1` to notify the other cell 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 notice 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 IS THIS STILL HASKELL?

The library presents itself with a innocent looking, pure and simple interface (Figure 1), but the implementation is full of side-effects (as seen in Section 4). Does this leak, or does programming with it still feel like programming Haskell? Is this really pure? Did we create a beautiful abstraction, or are the types a lie?

In this section we discuss these questions, however without always being able to give a definite answer, and invite you to join the discussion.

5.1 What does purity even mean?

We found that this question is not easy to answer, because there does not exist a single, clear and widely accepted definition of "pure" functional programming. This paper is not the place to give an authoritative answer, so we look at various language properties commonly associated with a pure functional language in general and Haskell in particular, and see whether they still hold in the presence of our library

5.2 Type safety

Purity probably means little without memory and type safety, and we certainly want to preserve Haskell's safety properties: So what we get out of `RS.get` should really be a valid `Set`, and not some corrupted memory.

It is not hard to break type safety with `unsafePerformIO`, especially when there are mutable references and polymorphism around. Nevertheless, we are fairly confident that we did not and that our library is type and memory safe.

5.3 No (observable) side effects

Our library uses `IO` inside, and clearly we would not consider it to be pure if inserting `42` into a set would make the computer eject the CD drive. Avoiding such blatant side effects is simple – we just don't do it.

However, `RS.insert 42 rs` *does* destructively modify mutable data structures inside `rs`. It is crucial that these changes remain internal to the library, and are not observable by the programmer. It must not affect the result the programmer gets via `RS.get rs`, nor any other *observable* value. Again, we believe that this is the case, for the same reasons we believe we maintain referential transparency (Section 5.5).

That the space leak issue of the naive implementation described in Section 4.4.3 is *almost* an issue of this kind: If we'd include space leaks in our notion of "observable behavior", then that would be an example of such a side-effect issue.

5.4 Evaluation order independence.

Even more, it should not matter in which order the various subexpressions are evaluated. In

```
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)
```

we need to get the same result, no matter whether we evaluate the first component of the resulting tuple before the second, or the other way around, or even all in parallel in separate threads.

Achieving this in our library required some care. Because our computation are triggered via `unsafePerformIO`, we have to expect any of them to happen at almost any time. Simple graph traversal methods, which simply mark a vertex as "done" before visiting its successors, easily go wrong when a second evaluation hits such a supposedly "done" vertex while the first traversal is still processing the graph. Getting this to the point where it seems to be working was non-trivial, as mentioned in Section 4.4.1.

5.5 Referential transparency

Purity is usually understood to also imply referential transparency: If `x` is defined to be `e`, then we can replace an occurrence `x` with `e`, and vice-versa, without changing the meaning of the program. (Care has to be taken when applying this to recursive definitions. Of course we do not expect `let x = 1 in x` to be the same as `let x = x in x`, although we merely replaced `1` with `x`.)

So is our library still referential transparent? By virtue of always calculating a unique fixpoint, it is! The value of `let rs2 = RS.insert 42 rs1` is the (least) set for which the equation `RS.get rs2 == S.insert 42 (RS.get rs1)` is true, and thus `rs2` and `RS.insert 42 rs1` can be used interchangeably.

This restriction to fixpoints, i.e. to solutions of a set of equations, puts a clear limit on the kind of observations we allow the programmer to make about the recursive equations. For example, it would not be pure if the programmer could obtain the number of equations. This rules out using our approach for use-cases where the actual *structure* of the equations (and not just a solution) is of interest, e.g. when generating circuit descriptions [Claessen and Sands 1999].

5.6 Program equivalences and lambda lifting

Referential transparency is a crucial ingredient to equational reasoning: rewriting the code according to a set of rules, while preserving its meaning. Beyond just replacing `x` with its definition `e` (also called *unfolding* or δ-reduction) are further program transformations that we expect to preserve

semantics. They may be applied by optimizing compiler, used in program verification or simply by a programmer trying to understand the meaning of some code.

Ideally, all program transformations that are meaning-preserving in pure Haskell would still meaning-preserving in the presence of our library. Unfortunately, this is not the case: Lambda lifting can turn terminating programs into non-terminating ones.

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. With our library, this 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 1 can solve this problem. (By this reasoning we also deduce that a pure implementation of that interface indeed does not exist.)

So does this invalidate our claims that we built a beautiful, well-behaved, pure abstraction? In the end that depends on your expectation: Do you expect that in a pure and lazy language lambda-lifting preserves meaning, including termination? Or is that an additional property of some languages, which we can maybe do without?

As you ponder this question, note that if you include (asymptotic) resource usage in your expectation of program equivalence, then such lambda-lifting is already rather dangerous in Haskell: 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))`). Because the lambda-lifting transformation is only an equivalence as long as we ignore (asymptotic) complexity, it seems that breaking that equivalence is not too wicked.

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

5.7 Compiler transformations

The compiler tends to share the attitude that (at least asymptotic) complexity is a relevant aspect of the program's semantics, and will thus be careful to preserve (recursive) sharing that is explicit in the source program. Therefore we can be fairly certain that it will not just nilly-willy apply such lambda-lifting to our code.

The same holds for all the other transformations applied by the compiler, such as partial evaluation, case-of-case, worker/wrapper, common subexpression elimination etc. [Peyton Jones and Santos 1998]. Because our library has these nice equational properties as long as sharing is preserved, we can be fairly certain that all these are still meaning-preserving in the presence of our library.

5.8 Sabry's criterion

One possible definition for purity is offered by Sabry [1998]:

> A language is purely functional if (i) it includes every simply typed λ-calculus term, and (ii) its call-by-name, call-by-need, and call-by-value implementations are equivalent (modulo divergence and errors).

It is not straight-forward to apply this criterion to our library, or Haskell itself, because it assumes the existence of these three implementations, and we do not have that for Haskell. Even if we had, in the call-by-value implementation, all our recursive definitions would vacuously loop, and in the call-by-name implementation (quite like after the lambda lifting), our library would not see *finite* graphs, so all interesting programs would become no-terminating. In that sense, our library *is* pure according to Sabry's criterion.

Because it requires multiple implementations we find Sabry's criterion of purity not fully convincing. Maybe there exists a suitable criterion that works with a single language semantics, maybe by listing observable equivalences that must hold for a language to be considered pure? We'll leave this question hanging, hoping it can tempt someone to give a satisfying answer.

5.9 Longley's criterion

Longley [1999] makes the observation that there exist functions with perfectly pure and functional *behavior* that cannot be *implemented* in the pure fragment of the language, but can be implemented in a suitable extension (ML with references in his case). Our library is an example of that: It behaves pure and functional, but needs extensions (like `unsafePerformIO`) to be implemented.

Longley considers a term to be functional if it denotes an element in the mathematical denotation of its type. At function type, this means they are extensional and that they are described by a mathematical function. We struggle applying this criterion directly, since we have not yet managed to find a denotational semantics for Haskell extended with our library. Unlike usual call-by-name semantics for pure functional programming languages, it must be be able to tell knot-tying recursion apart from unrestricted recursion. It remains to be seen if we can find a suitable denotational semantics.

5.10 How to prove it?

So with the exception of preservation under lambda-lifting, we are fairly confident that our library is as pure as advertised. Ideally we'd like to be sure, and we'd be sure if we could perform a rigorous proof. That not only requires a clear definition of what exactly "pure" means, as discussed above, but also a framework that allows us to reason about lazy programs with `unsafePerformIO`, `IORef`, laziness and concurrency.

Unfortunately, we do not know how to do that well. In the single-threaded world we could imagine extending an operational semantics for lazy languages such as Launchbury's [1993] natural semantics for lazy evaluation with a global store for the mutable variables, and spurious evaluations for `unsafePerformIO`, but it seems tedious and we would not yet have threads.

The maybe most similar work is Timany et al.'s 2018 proof that `runST :: ST a -> a` is safe, although it does not yet cover laziness nor concurrency. They used the Iris framework, and for Iris one even finds a mechanized proof of the correctness of a generic fixpoint solver [de Vilhena et al. 2020]. It seems that important building blocks are already there, and we hope that this paper can maybe motivate more work in that direction.

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, which is good, as there is much interesting than can be said and done here.

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 influence that vertex.

6.2 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), point out that this breaks some equational equivalences of Haskell, but 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 5.7.

Their extension to Haskell is bolder: It breaks 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 probably 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, and use Haskell's laziness together with `unsafePerformIO` (which will be executed once per value) to create unique mutable cells and let those cells 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 graphs 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.

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).

6.3 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.

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.

Monotonicity is crucial for the existence and uniqueness of a solution (Section 2.5). We ensure monotonicity by simply restricting the interface (Figure 1) 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.[4] Datafun's type system has a separate function arrow $\xrightarrow{+}$ to characterize monotonic functions, and thus supports this use-case quite well.

7 CONCLUSION AND FURTHER WORK

We saw that we can extend Haskell with the ability to solve recursive equations involving Boolean and sets, that this extension can be implemented as a library, and that this extension – 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 way to describe what the library do.

REFERENCES

Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: a functional Datalog. In *ICFP 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 214–227. https://doi.org/10.1145/2951913.2951948

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP 2000*. ACM. https://doi.org/10.1145/351240.351266

Koen Claessen and David Sands. 1999. Observable Sharing for Functional Circuit Description. In *ASIAN'99 (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

Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. 2020. Spy game: verifying a local generic solver in Iris. *Proc. ACM Program. Lang.* 4, POPL (2020), 33:1–33:28. https://doi.org/10.1145/3371101

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 *POPL'73*, Patrick C. Fischer and Jeffrey D. Ullman (Eds.). ACM Press, 194–206. https://doi.org/10.1145/512927.512945

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *POPL'93*. ACM. https://doi.org/10.1145/158511.158618

John Longley. 1999. When is a Functional Program Not a Functional Program?. In *ICFP'99*, Didier Rémy and Peter Lee (Eds.). ACM, 1–7. https://doi.org/10.1145/317636.317775

Simon Peyton Jones, Simon Marlow, and Conal Elliott. 1999. Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell. In *IFL'99 (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

Simon L. Peyton Jones, Andrew D. Gordon, and Sigbjørn Finne. 1996. Concurrent Haskell. In *POPL'96*, 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 and André L. M. Santos. 1998. A Transformation-Based Optimiser for Haskell. *Sci. Comput. Program.* 32, 1-3 (1998), 3–47. https://doi.org/10.1016/S0167-6423(97)00029-4

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

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

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 *Haskell 2015*, Ben Lippmeier (Ed.). ACM, 141–152. https://doi.org/10.1145/2804302.2804306

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 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 = do
    watchCell c2 update
    update
```

```
    where
      update = do
        s <- getCell c2
        setCell c1 (S.insert x s)
```