Worker/wrapper for a Better Life

Extended Abstract

Brad Torrence Mike Stees Andrew Gill

Information and Telecommunication Technology Center
The University of Kansas
{brad.torrence,mstees,andygill}@ittc.ku.edu

Abstract

In Software Engineering, an implementation, and its model, can fall out of step. If we can connect the implementation and model, then development of both artifacts can continue, while retaining confidence in the overall design and implementation. In this paper, we show it is possible in practice to connect together an executable specification of Conway's Game of Life, and a number of implementations, using the worker/wrapper transformation. In particular, we use the rewrite tool HERMIT to apply post-hoc transformations to replace a linked-list based description with other data structures. Directed optimizations allow for highly-efficient executable specifications, where the model becomes the implementation. This work is the first time programmer-directed worker/wrapper has been attempted on a whole application, rather than simply on individual functions. Beyond data representation improvement, we translate our model such that we can execute on a CPU/GPU hybrid, by targeting the accelerate DSL.

1. Introduction

The concepts of the worker/wrapper methodology introduced by Gill and Hutton in [6] are the driving force behind the examples demonstrated in this paper. We have implemented the worker/wrapper methodology in the Haskell Equational Reasoning Model to Implementation Tunnel (HERMIT) system. HERMIT has already demonstrated several examples of its capability of using the worker/wrapper theory [3, 4, 11]. However, these examples have only consisted of small transformations usually over a single function. In this paper, we scale the methodology and tool support to a larger example – an implementation of the Game of Life.

1.1 Worker/Wrapper

The worker/wrapper transformation is a technique for improving the performance of a program by changing the underlying implementation. The transformation generates two components: the "worker", which performs using the new implementation, and the "wrapper", which conceals this new implementation under the original API.

When using worker/wrapper, the programmer provides two functions, `abs` and `rep` that translate from the new representation to the original representation, and from the original representation to the new representation. When given some specific preconditions on the relationship between `abs` and `rep`, we can assume another condition that creates a provably correct worker/wrapper transformation [10]. Specifically, given a fix-point of the original computation,

```
comp = fix work
```

as well as `abs` and `rep`, and the worker/wrapper preconditions, we can rewrite comp into a worker and wrapper.

```
comp = abs worker
worker = fix (rep . work . abs)
```

Critically, the worker is now acting over a new representation. By applying laws that relate `rep`, `work`, and `abs`, we can optimize the `worker`, giving a more efficient program. The user intervention is the choice of `abs` and `rep`, and demonstrating the pre-conditions, and we want to use HERMIT to perform both of these steps, as well as optimize the result.

1.2 HERMIT

HERMIT is a framework that provides tools for interacting with and transforming GHC Core programs. The primary means of facilitating this interaction is through the HERMIT Shell, a REPL interface that allows the user to traverse a GHC Core abstract syntax tree. It is inside this REPL that the user issues commands that construct rewrites from AST to AST [3, 11]. Use of the 'unfold-rule' command in particular, in conjunction with GHC RULES pragma [8], allows the user to construct a rewrite from the given rule [3].

Adding to these capabilities are the new worker/wrapper related commands, 'split-1-beta' and 'split-2-beta'. These new commands perform the worker/wrapper split using the safe correctness conditions discussed in the previous section. These commands take a function argument, which is the target of the split, and two more arguments that comprise a transformation-pair. This pair of functions are referred to as the 'abs' and 'rep' functions [10] or 'work' and 'wrap' functions [6] in previous discussions about the worker/wrapper theory. If this transformation pair is created to meet the necessary preconditions to make 1β or 2β true, then they can be given to a "split" command. The product of a split is a worker, which implements new functionality, and a wrapper, which simply calls the new function, maintaining the original interface.

The creation of these commands have given HERMIT users the ability to transform an entire program using the worker/wrapper methodology, provided the theory can be applied effectively using automated mechanisms such as HERMIT to transform an implementation without changing the source code.

1.3 The Game of Life

To show that worker/wrapper transformations under HERMIT can be accomplished for complex programs composed of multiple source files, a suitable program had to be selected that was complex enough to have merit, but not so complex as to provide an overly involved example. An example program that contains these features is the Game of Life.

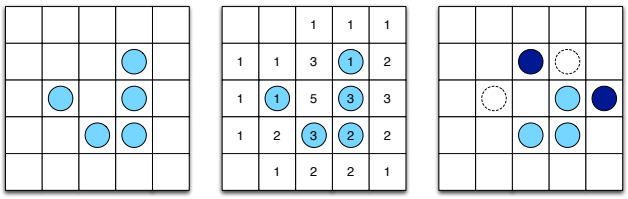
The Game of Life was created by the British mathematician John Horton Conway [5]. Technically, it's a simulation. A player creates the initial state of the board and the rules of the game determine how the board evolves, the player can only observe once the game has begun.

The game board is made of cells arranged in a two dimensional grid. A cell can either be in an alive or dead state, and depending on the state of neighboring cells a cell may change its state. The neighbors of a cell are simply those immediately adjacent to the the target cell in any direction. The rules of the game are simple and as follows:

1. Under-population – A living cell dies if it has fewer than two living neighbors.
2. Stable-population – A living cell remains alive if it has 2 or 3 living neighbors.
3. Over-population – A living cell dies if it has more than three living neighbors.
4. Reproduction – A new cell is born if it is empty and has exactly three living neighbors.

The combination of these simple rules allows for surprisingly interesting and complex patterns to emerge from the game.

The first image in the following figure shows a popular pattern in the game known as the Glider. This pattern replicates through generations. Due to the rules of the game, the pattern moves across the game board in a specific direction. The second image shows the pattern with each cell imprinted with the number of living neighbor cells. These numbers determine the pattern of the next generation, displayed in the third image. The white cells die due to under-population, the dark cells are newly born due to reproduction, and the lightly colored cells remain from the previous generation due to stable-population numbers. These five cells represent the new generation in the game and are used to calculate the next pattern.



The game has been implemented many times. Graham Hutton implemented the Game of Life in Haskell using a simple list-based implementation in a terminal console[7, p. 94-97]. It was from Hutton's original implementation that our implementation was derived.

The implementation used in this experiment still uses Hutton's original design, however it has been slightly modified. The new design restricts the implementation to use a sorted-list. This created an isomorphic structure to transform, making the task of proving code equivalence simpler. The sorted-list is made of two-tuples of integers. Each pair represents a location on the two-dimensional board structure and has a type synonym, `Pos`. The pairs featured in the list represent the position of living cells in each generation of the game. Therefore, when a cell dies that position is removed from the list, and the reverse is true when a new cell is born.

Another difference in Hutton's Life is that the dimensions of the game board were hard coded into the source code. The program has been modified in a way to allow the board configuration to be altered by user input. Through these modifications, the user can change the board dimensions, represented as an integer pair. The user can also dictate whether or not the edges of the board wrap around to the opposite edge, represented as a boolean value. This information is stored into a new type created called a `Config`. It is simply a two-tuple of an integer pair and a boolean. This configuration and the data structure containing the game data are contained in a new structure called a `LifeBoard`, and it is a new data type with a constructor and accessing functions for the board and configuration fields. It also aided in another change to the original program. Hutton's Life source code was also divided. The part of the code that calculates each generation of the game (the engine) has been separated from the part of the code that visualizes each generation (the display). The reason was to allow new engine and display mechanisms to be created and connected. The class which merges the engine and the display elements is the Life class. This abstract interface contains functions that allow the combination of any engine that implements the Life class functions with any display that utilizes the Life class interface. This section shows the module Life.Types which contains the types described.

```
type Pos = (Int,Int)
type Size = (Int,Int)
type Config = (Size,Bool)

class Life b where
    empty :: Config -> b
    diff :: b -> b -> b
    next :: b -> b
    inv :: Pos -> b -> b
    dims :: b -> Size
    alive :: b -> [Pos]

scene :: Life board => Config -> [Pos] -> board
scene = foldr inv . empty

data LifeBoard c b = LifeBoard{ config :: c, board :: b }
    deriving Show
```

The module contains the abstract definition of the Life class mentioned as well as the LifeBoard structure. In addition, the `scene` function is also defined there. It provides a standard way to transform a [Pos] into an implementation-specific board. And the following section shows the Life.Engine.Hutton module which is derived from Hutton's Life.

```
type Board = LifeBoard Config [Pos]

neighbs :: Config -> Pos -> Board
neighbs c@((w,h),warp) (x,y) = LifeBoard c $ sort $ if warp
    then map (\(x,y) -> (x `mod` w, y `mod` h)) neighbors
    else filter
        (\(x,y) -> (x >= 0 && x < w) && (y >= 0 && y < h))
        neighbors
    where neighbors =[(x-1,y-1), (x,y-1), (x+1,y-1), (x-1,y),
                      (x+1,y), (x-1,y+1), (x,y+1), (x+1,y+1)]

isAlive :: Board -> Pos -> Bool
isAlive b p = elem p $ board b

isEmpty :: Board -> Pos -> Bool
isEmpty b = not . (isAlive b)

liveneighbs :: Board -> Pos -> Int
```

```
liveneighbs b =
  length . filter (isAlive b) . board . (neighbs (config b))

survivors :: Board -> Board
survivors b = LifeBoard (config b) $
    filter (\p -> elem (liveneighbs b p) [2,3]) $
        board b

births :: Board -> Board
births b = LifeBoard (config b) $
   filter (\p -> isEmpty b p && liveneighbs b p == 3) $
    nub $ concatMap (board . (neighbs (config b))) $
        board b

nextgen :: Board -> Board
nextgen b = LifeBoard (config b) $
    sort $ board (survivors b) ++ board (births b)

instance Life Board where
    next b = nextgen b
    alive b = board b
    empty c = LifeBoard c []
    dims b = fst $ config b
    diff b1 b2 = LifeBoard (config b1) $
            board b1 \\ board b2
    inv p b = LifeBoard (config b) $
        if isAlive b p
        then filter ((/=) p) $ board b
        else sort $ p : board b
```

Although the top-level program module is loaded into HERMIT, only this engine module is targeted for transformation, leaving the original display mechanisms unchanged and still effective through worker/wrapper methodology.

2. HERMIT Worker/Wrapper Examples

The goal of this experiment is to confirm that an entire program can be transformed with HERMIT using the worker/wrapper methodology. The examples take a slighly-modified version Hutton's original implementation of the game, which uses lists as the data structure for representing the game board, and modify it using HERMIT. This was done by applying the worker/wrapper concept through HERMIT in an effort to change the underlying data structures used in the program.

The following sections describe the examples and the methods used in HERMIT to accomplish a worker/wrapper transformation. Each experiment involved changing Hutton's Life (the list-based implementation) into an implementation using another primary data structure, such as a QuadTree, a Set, and an Unboxed Vector. Along with changing the representation of the game, we found it is also possible to change the hardware used by the program with the inclusion of certain DSLs (like Accelerate, which gives program access to a GPU). First, lets explore the preparation process required to do these transformations within HERMIT.

As outlined in previous worker/wrapper example transformations using HERMIT [11], the process requires the creation of specific GHC rules and a set of transformation functions.

The module that contains this information is referred to as the transformation-module. This module must be tailored specifically to each conversion. A transformation-module requires one pair of conversion functions for each function targeted. When approaching the problem, it is best to start by creating the most basic transformations first. Transformation pairs can be stacked, using more basic pairs in their definitions.

The transformation-module is also where a series of GHC Rules will be written to allow HERMIT to equate sections of Core code between old and new implementations. This is accomplished via the use of GHC rules that are added to the conversion file using the GHC RULES pragma and compiled into the HERMIT session.

Once a worker/wrapper split is made within HERMIT, the process requires making transformations to segments of code in an effort to match the AST with a GHC rule that swaps equivalent code statements.

It is most likely that a HERMIT user will not know all the needed rules to accomplish the transformation before the process begins. At this stage in HERMIT development, it is easiest to take a more organic approach when performing a worker/wrapper conversion. By that, one should create the needed transformation-pairs and a few rules that will be known to be useful to start, then add new rules as they are needed. Adding a new GHC rule to a HERMIT session requires exiting HERMIT and reentering to add any newly created rules to the environment. This was the process used to complete these transformations. That being said the finished product is a HERMIT script that can perform the entire transformation. This script makes a perfect guide to lead through the following examples.

These examples were performed with GHC 7.8 in combination with the latest version of HERMIT. All of them target the same program described in the previous section.

2.1 Example: List-to-Set implementation transform

The first example uses the same data representation for the board structure. The goal of the transformation was to implement the game engine using the Set data structure featured in the standard library Data.Set. The transformation replaces the use of the standard Haskell list with the use of the set in the engine module. This transformation is isomorphic because we have restricted the use of the Set to maintain order of its elements. Therefore, relationship of a sorted-list to a sorted-set is trivially equivalent, swapping the containers used by the engine produces an equivalent program. The representation remains the same, both containers contain pairs of integers that correspond to living cells on the board. The containing LifeBoard structure is maintained except it is morphed to contain a set rather than a list.

2.1.1 Transform Preparation

The transformation-module should contain all the transformation functions necessary to complete the process. To make the type definitions shorter and relate to the list-based engine, the module also includes type synonyms for the `Board` (copied from the source) and `Board'` (the transformed type). These type definitions as well as all the necessary transformation pairs are displayed here for reference.

```
type Board = LifeBoard Config [Pos]
type Board' = LifeBoard Config (Set Pos)

{-# NOINLINE absb #-}
absb :: Set Pos -> [Pos]
absb = toAscList

{-# NOINLINE repb #-}
repb :: [Pos] -> Set Pos
repb = fromDistinctAscList

{-# NOINLINE absB #-}
absB :: Board' -> Board
absB b = LifeBoard (config b) $ absb (board b)

{-# NOINLINE repB #-}
repB :: Board -> Board'
repB b = LifeBoard (config b) $ repb (board b)

absBx :: (Board' -> a) -> Board -> a
absBx f = f . repB
```

```
repBx :: (Board -> a) -> Board' -> a
repBx f = f . absB

absxB :: (a -> Board') -> a -> Board
absxB f = absB . f

repxB :: (a -> Board) -> a -> Board'
repxB f = repB . f

absCPB :: (Config -> Pos -> Board') -> Config -> Pos -> Board
absCPB f = absxB . f

repCPB :: (Config -> Pos -> Board) -> Config -> Pos -> Board'
repCPB f = repxB . f

absBB :: (Board' -> Board') -> Board -> Board
absBB = absBx . absxB

repBB :: (Board -> Board) -> Board' -> Board'
repBB = repBx . repxB

absPBB :: (Pos -> Board' -> Board')-> Pos -> Board -> Board
absPBB f = absBB . f

repPBB :: (Pos -> Board -> Board) -> Pos -> Board' -> Board'
repPBB f = repBB . f

absBBB :: (Board' -> Board' -> Board') -> Board -> Board -> Board
absBBB f = absBB . f . repB

repBBB :: (Board -> Board -> Board) -> Board' -> Board' -> Board'
repBBB f = repBB . f . absB
```

The purpose of the transformation is to replace the use of lists with the use of sets. The most basic transformation function pair, and the first that should be created, is the pair that transforms the data structure. In staying true to the worker/wrapper methodology the function names begin with `abs` and `rep`. These abs-rep pairs need to be designed to perform bidirectional transformations, where the abs function returns the original form and the rep function returns the new form.

The base pair are named `absb` and `repb`. Both make use of the Data.Set functions that transform a set to/from an ordered list, the `fromDistinctAscList` and `toAscList` functions. Each function of the pair should reverse the results of the other. The compiler directives above these functions are directions that notify GHC to not automatically inline these functions. GHC will do this automatically for some code in an optimization effort. Without these directives the absb and repb functions would not appear in HERMIT session.

To continue, the base pair only transforms the underlying data structure. There also needs to be a pair of functions that will transform the containing data structure `LifeBoard`. Notice how they simply replace one of the contained structures using of the predefined absb or repb functions. This stacking trend continues through all of the transformation-pairs.

Analyzing the source code of the Hutton's Life helps to know what other transformations are necessary. We start with the Life class function `dims`. This function simply returns the board dimensions that are originally entered by the user. The output of the `dims` function is a Size, which doesn't require any transformation because the new implementation retains this original structure. The absBx-repBx pair is defined is polymorphic because with its simple definition it can be used for several conversions. Since the function-types of the `alive`, `isAlive`, `isEmpty`, and `liveneighbs` functions, are similar to the `dims` function, this pair can also be used in their conversion processes.

Next we turn to the function `empty`. This function simply creates an empty board structure. To convert this function the functions, `absxB` and `repxB` would be used. This pair is polymorphic for reasons similar to the absBx-repBx pair, it can be used in several worker/wrapper splits.

Now consider the `neighbs` function. It is used to create a list of neighboring locations depending on the board configuration. This function will require the absCPB-repCPB pair for conversion. The previous polymorphic functions are used in their definitions. This aids the unfolding and code reduction process inside a HERMIT session.

Probably the most crucial function in the Life class for our purposes is the "next" function. This function is used to calculate the next generation of the game from the current board structure. It requires the functions, `absBB` and `repBB`. Upon inspecting the source code, one will note that this type of transformation will be necessary for the engine functions `nextgen`, `births`, and `survivors`. Reusing transformation pairs is useful, and since these functions have the same type, there is no need to create a polymorphic transformation function.

Now consider the `inv` function of the Life class. It takes a board position and inverts the cell status on the given board. The `absPBB` and `repPBB` functions are required for its transformation.

The last function to consider is `diff`. It is used to compare two boards and get return the differences between them in a new board. The `absBBB` `repBBB` definitions are used to convert this function. It has the most complex definition because all of its arguments are of type `Board` or `Board'`, and so requires the most transformation.

With the function pairs completed, we move on to the transformation process within HERMIT. But first, we can assume a few necessary GHC Rules that will definitely be useful, like the following.

```
{-# RULES "repB-absB" [~] forall b. repB (absB b) = b #-}

{-# RULES "LifeBoard-absb" [~] forall c b.
  LifeBoard c (absb b) = absB (LifeBoard c b) #-}

{-# RULES "config-absB" [~] forall b.
  config (absB b) = config b #-}

{-# RULES "board-absB"  [~] forall b.
  board (absB b) = absb (board b) #-}

{-# RULES "repB-LifeBoard" [~] forall c b.
  repB (LifeBoard c b) = LifeBoard c (repb b) #-}
```

These rules are known to be useful because they all simply perform a code transformation designed to move the transformation function node in the AST. For instance, the 'repB-absB' rule is used to eliminate an transformation pair once they have been syntactically juxtaposed. Since the goal is to remove unnecessary transformations, this rule is almost necessity. The other rules shown above are useful for similar reasons, they either eliminate or move a transformer in the AST. Typically one won't know all the rules needed prior to starting a worker/wrapper conversion due to the fact that one may not know what form the Core syntax will take during the process. However, the following rules were discovered and are necessary to complete this example.

```
{-# RULES "repb-null" [~] forall c.
  LifeBoard c (repb []) = LifeBoard c Set.empty #-}

{-# RULES "not-elem-absb" [~] forall p b.
  not (elem p (absb b)) = notMember p b #-}

{-# RULES "elem-absb" [~] forall p b.
  elem p (absb b) = member p b #-}
```

```
{-# RULES "length-absb" [~] forall b.
  length (absb b) = size b #-}

{-# RULES "filter-absb" [~] forall f b.
  Prelude.filter f (absb b) = absb (Set.filter f b) #-}

{-# RULES "sort-++-absb" [~] forall b1 b2.
  sort (absb b1 ++ absb b2) = absb (union b1 b2) #-}

{-# RULES "ncm-absb" [~] forall f b.
  nub (concatMap (\p -> absb (board (f p))) (absb b)) =
  absb (unions (toList (Set.map (\p -> board (f p)) b))) #-}

{-# RULES "diff-absb" [~] forall b1 b2.
  absb b1 List.\\ absb b2 = absb (b1 Set.\\ b2) #-}

{-# RULES "insertion" [~] forall b p.
  sort (p : absb b) = absb (insert p b) #-}

{-# RULES "deletion" [~] forall b p.
  Prelude.filter ((/=) p) (absb b) = absb (delete p b) #-}
```

Notice that each of these rules replaces some list-based implementation with an equivalent implementation that uses the set data-structure.

Once the transformation functions are created, a HERMIT session can be started and the worker/wrapper conversion attempted. If all of the rules are known prior to start then the conversion can be completed in one session. Most likely one will have to exit a session to create rules to continue the process. This is where HERMIT scripts are useful. Commands in HERMIT can be saved as scripts, which is especially useful for replaying sessions after modifying HERMIT.

2.1.2 Transform Process

The order that the transformation occurs is important, one should focus on the functions that have the no dependence on other target functions first. One must also consider the transformed result when deciding since the dependencies may change during the process. Transforming one function before you have transformed another function on which the first depends will only create more work, and most likely a poor transformation result.

With these facts in mind, we begin with the Life class functions for the Board type. Consider first, the `empty` function, its definition is very simple and it doesn't depend on other functions. The result that is desired would instead use an empty set rather than an empty list. The desired function will also not depend on any of the other module functions. This is a great function to start the transformation. From the worker/wrapper method we know that at some point the definition will be `repB (LifeBoard c [])`, which after unfolding repB will be `LifeBoard c (repb [])`. This indicates that the rule called 'repb-null' is necessary. Since we have a transformation rule ready for the `empty` function the process can begin.

The script used to convert the `empty` function is shown here as an example.

```
binding-of '$cempty
fix-intro
down
split-1-beta $cempty [|absxB|] [|repxB|]
{
  rhs-of 'g
  repeat (any-call (unfold ['repxB, 'repB]))
  bash
  any-call (unfold-rule repb-null)
}
let-subst
alpha-let ['$cempty']
{ let-bind ; nonrec-rhs ; unfold ; bash }
top
innermost let-float
```

The first and last few lines of each conversion script are identical, the first few commands focus HERMIT on the proper function then perform the worker/wrapper split. The last few commands move the new function to the top-level of the program so that it becomes a part of the API. Most of these commands are common HERMIT commands that are used to manipulate the Core AST. For brevity, these commands will not be covered in detail. The interesting command that may differ each script is the `split-1-beta` command.

The arguments for this command will vary with each script, here the command is `split-1-beta $cempty [|absxB|] [|repxB|]`. This command performs a worker/wrapper split on the named function (`$cempty`) using the given transformation-pair (`absxB` and `repxB`). This command is what performs the worker/wrapper split, producing the wrapper, which retains the original name `$cempty`, and the worker, which is always named g. The rest of the commands that are unique to the script are shown between the { and } commands. Except the `rhs-of g`, this command is common to all the scripts.

The first step in all the scripts is to unfold the transformation functions. How far they are unfolded depends on the particular function being converted. The `bash` command is commonly used to reduce code to a form more suitable for using GHC rules. The definition of `empty` is not complex and does not require much manipulation. That is why there are few commands present in this script. Typically following the unfolding of the transformers there are a series of Core manipulations that would put the AST in a form that matches a predefined GHC rule. However, for this conversion the `bash` command was sufficient.

The application of a GHC rule is done with the `unfold-rule` command. As seen in this script, only the 'repb-null' rule was needed and applied. After the application of this script the function `empty'` is available through the API. The equivalent definition in Haskell would be `empty' c = LifeBoard c Data.Set.empty`.

Focusing on the `alive` conversion script, the absBx-repBx pair are used to perform the worker/wrapper split, and repBx is the only function unfolded. The original `alive` function is synonymous with the `board` access function. But in our new program `board` will return a `(Set Pos)`, and `alive` still returns a `[Pos]`. So, it is necessary to leave a transform function in the new definition. This is one instance where one does not wish to eliminate all of the transformation functions from the definition. The `toAscList` function will appear by simply unfolding the `absb` function, when it is in the right position. This produces the definition `alive' b = toAscList (board b)`, completing this transform.

The `dims` function, also uses the absBx-repBx pair. And similar to the previous conversion, it only needs to unfold the repBx function. After the application of the 'config-absB' rule to eliminate the transform-function. The function is in the desired form since the `dims` definition should not change. After applying this rule the conversion is complete, and "dims'" will be accessible.

The `diff` function conversion requires the absBBB-repBBB pair to perform the worker/wrapper split. And after unfolding the repBBB function, a few rules are needed to complete the transformation, which requires swapping the use of `Data.List.\\` to `Data.Set.\\` via the 'diff-absb' rule.

The `neighbs` conversion makes use of the absCPB-repCPB pair for its split. The process is similar to the process for `alive`, in that it requires leaving a transformation in place. For this function, the

implementation is left alone and the structure is converted before being returned.

The worker/wrapper split is performed on the next three functions using the absBx-repBx pair. The function `isEmpty` is considered next. Its definition is simple but it is important that this function be converted before `isAlive` because `isEmpty` depends on it. However, when converted it will no longer have this dependency. The resulting definition of `isEmpty` uses the `Data.Set.notMember` function and is created with the 'not-elem-absb' rule.

Once that is complete the process moves to `isAlive`. The process is similar to `isEmpty`'s but uses the 'elem-absb' rule to produce a function that uses the `Data.Set.member` function.

The `liveneighbs` function uses the 'filter-absb' and 'length-absb' rules to complete its conversion which replaces `Prelude.filter` and `Prelude.length` with `Data.Set.filter` and `Data.Set.size` respectively.

The functions `survivors`, `births`, `nextgen`, and `next` all require the use of the absBB-repBB pair. The `survivors` function only needs to swap its `filter` function from the Prelude to Set implementation. But `births` requires this transformation in addition to the use of the 'ncm-absb' rule, which changes the list-based implementation to a set-based one.

The `nextgen` function requires the use of the 'sort-++-absb' rule, which changes concatenation into a set union. While `next` is a special case an only requires changing the function to call `nextgen'` rather than `nextgen`.

The final function in the conversion is `inv`. This conversion requires that `isAlive` be transformed first so that `isAlive'` is accessible. Although, `inv` is not dependent on `isAlive`, `inv'` will depend on `isAlive'`. This function is a little more complex to convert because it has two branches that must be accounted during the transformation. It requires the use of both 'insertion' and 'deletion' rules to replace the code in each branch.

When all the necessary functions have been converted the HERMIT command `continue` can be given to complete compilation of the new program.

In addition to the example above, the final paper will discuss similar transformations from List to QuadTree, and List to Unboxed Vector.

2.2 Example: List-to-Accelerate implementation transform

The Accelerate language is a Haskell embedded DSL that provides arrays and scalars, and a collection of collective operations applied to arrays. These operations are algorithmic skeletons that target CUDA implementations via the Accelerate code generator [2]. For more information about Accelerate and its implementation, consult [2, 9].

2.2.1 Transform Process

For this transformation, we again use the definition of `Board` from our earlier examples, but we choose an interesting type for `Board'`.

```
type Board = LifeBoard Config [Pos]
type Board' = LifeBoard Config (Acc (Array DIM2 Int))
```

Additionally, the types and implementations of `absb` and `repb` are more involved than the previous ones we have seen.

```
repb :: Size -> [Pos] -> Acc (Array DIM2 Int)
repb (w,h) xs =
    A.reshape (A.index2 (lift w) (lift h))
              (A.scatter to def src)
    where     sz = List.length xs
        to = A.map (\pr -> let (x,y) = unlift pr
                    in (x * (lift w)) + y)
              (A.use $ A.fromList (Z :. sz) xs)
        src = A.fill (A.index1 (lift sz)) 1
        def = A.fill (A.index1 (lift $ w * h)) 0

absb :: Size -> Acc (Array DIM2 Int) -> [Pos]
absb (w,h) arr =
    let prs = A.reshape (A.index1 (lift (w * h)))
            $ A.generate (index2 (lift w) (lift h))
                (\ix -> let Z :. i :. j = unlift ix
                in lift (i :: Exp Int, j :: Exp Int))
        res = A.filter (\pr -> let (i,j) =
            unlift pr :: (Exp Int, Exp Int)
            in (arr A.! (index2 i j)) ==* 1) prs
    in toList $ run res
```

These choices, along with a comparison of this implementation to the other implementations will be discussed in the final paper.

3. Related works

The HERMIT toolkit has experienced success in a wide variety of applications. Some of those applications include applying the worker/wrapper transformation to optimize functions like `reverse` and `last` [3, 11]. Additionally, HERMIT has also been used to mechanize an optimization pass for SYB [1], and to enable stream fusion for `concatMap` [4].

The full paper will contain a detailed literature survey of related works.

4. Conclusion

Earlier work in [3, 11] showed that worker/wrapper can be successfully mechanized in the small. We extended that work to include the transformation of an entire program. In choosing our target application, Game of Life, we wanted an application that was complex enough to require several functions and potentially multiple modules, but simple enough that it could be conceptually understood quickly.

Through the course of our investigation into applying worker/wrapper to the Game of Life, we transformed the original List based version into versions that used Sets, Unboxed Vectors, and Quad Trees. In addition to changing the underlying structure, we also wanted to leverage the GPU. By targeting the Accelerate DSL, we were able to transform the List based version into a version that performed all of the population calculations on the GPU. Ultimately, our experiences in this exploration have shown us that application wide worker/wrapper transformations can in fact be mechanized, and that HERMIT is a valuable tool for doing such a mechanization.

Acknowledgments

We would like the thank Andrew Farmer for help with HERMIT. This material is based upon work supported by the National Science Foundation under Grant No. 1117569.

References

[1] M. D. Adams, A. Farmer, and J. P. Magalhães. Optimizing syb is easy! In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 71–82, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. . URL `http://doi.acm.org/10.1145/2543728.2543730`.

[2] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, pages 3–14. ACM, 2011.

[3] A. Farmer, A. Gill, E. Komp, and N. Sculthorpe. The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *Proceedings of*

the ACM SIGPLAN Haskell Symposium, Haskell '12, pages 1–12. ACM, 2012. ISBN 978-1-4503-1574-6. . URL http://doi.acm.org/10.1145/2364506.2364508.

[4] A. Farmer, C. Höner zu Siederdissen, and A. Gill. The hermit in the stream: Fusing stream fusion's concatmap. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*, PEPM '14, pages 97–108, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2619-3. . URL http://doi.acm.org/10.1145/2543728.2543736.

[5] M. Gardner. Mathematical games – the fantastic combinators of john conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[6] A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(02):227–251, 2009.

[7] G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.

[8] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

[9] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional gpu programs. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 49–60. ACM, 2013.

[10] N. Sculthorpe and G. Hutton. Work it, wrap it, fix it, fold it. *Journal of Functional Programming*, 24(1):113–127, 2014. URL http://dx.doi.org/10.1017/S0956796814000045.

[11] N. Sculthorpe, A. Farmer, and A. Gill. The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Proceedings of the 24th Symposium on Implementation and Application of Functional Languages*, volume 8241 of *Lecture Notes in Computer Science*, pages 86–103, 2013. URL http://dx.doi.org/10.1007/978-3-642-41582-1_6.