Botworld 1.0
(Technical Report)

Nate Soares, Benja Fallenstein

April 9, 2014

Contents

1 Motivation

This report introduces *botworld*, a cellular automaton used for studying self-modifying agents.

Most formal frameworks for studying self-modifying agents split the universe into an agent and an environment. The agent interacts with the environment only via discrete input and output channels.

Such formalisms are perhaps ill-suited for real self-modifying agents, which are embedded within their environments. Indeed, the agent/environment separation is somewhat reminiscent of cartesian dualism: any agent built using such a framework does not model itself as part of its environment.

Intuitively, this separation is not a fatal flaw, but merely a tool for simplifying the discussion. We should be able to remove this "cartesian" assumption from formal models of intelligence. Botworld is a tool for probing this intuition: it provides a concrete world containing agents that we wish to act intelligently, and allows us to study what happens when the cartesian barier between an agent and its environment begins to break down.

As it turns out, many interesting obstacles arise when agents are embedded in an environment. For example, agents whose source code may be read may be subjected to Newcomb-like problems (with entities that simulate the agent's actions and choose their actions accordingly).

Descision theoretical tools for solving such problems alrdeady exist (such as Vladimir Slepnev's formalism of updateless decision theory); botworld provides an environment where we can actually build the games, program the agents, and run the system.

Furthermore, certain obstacles to self-reference arise when non-cartesian agents attempt to achieve condfidence in their future actions. Some of these issues are raised in the *Tiling Agents* paper by Yudkowsky and Herreshoff; botworld gives us a concrete environment in which we can examine them.

One of the primary benefits of botworld is *concreteness*: when working with abstract problems of self-reference, it is often very useful to see a discrete game in a fully specified world that directly exhibits the obstacle under consideration. Botworld makes it easier to visualize these obstacles.

Conversely, botworld also makes it easier to visualize suggested agent architectures, which in turn makes it easier to visualize potential problems and probe the architecture for edge cases.

Finally, botworld is a tool for communicating. It is our hope that botworld will help others understand the varying formalisms for self-modifying agents by giving them a concrete way to visualize such architectures being implemented. Furthermore, botworld gives us a concrete way to illustrate various obstacles, by implementing botworld games in which the obstacles arise.

For example, consider an agent that is searching for a strategy in some game, and wants to do at least as well as some fallback strategy. In a traditional cartesian framework, the agent may adopt a simple architecture that searches for strategy/proof pairs where the proof proves that the strategy does better than the fallback.

In a non-cartesian environment, such a proof will not suffice. Consider, for example, a *stupidity rewarding* agent which reads source code and dispenses large rewards to agents with the program "do nothing ever".

An agent in the same world as the stupidity rewarder with a fallback strategy of "do nothing ever" will fail to self-modify into an agent that does nothing ever, because it falsely believes that its proof-searching strategy will do at least as well as its fallback strategy. This agent fails to realize that the stupidity rewarder can distinguish between robots that *actually* do nothing ever and robots that search for strategies (with "do nothing ever" as a fallback).

This problem is somewhat abstract and perhaps difficult to visualize—but in botworld, we can *actually build a game* with a stupidity rewarder and a proof-searching robot, and see how the proof searcher goes wrong.

Botworld has helped us gain a deeper understanding of varying formalisms for self-modifying agents and the obstacles they face. It is our hope that botworld will help others more concretely understand these issues as well.

2 Overview

Botworld is a high level cellular automaton: the contents of each cell can be quite complex. Indeed, cells may house robots with register machines, which are run for a fixed amount of time in each cellular automaton step. A brief overview of the cellular automaton follows. Afterwards, we will present the details along with a full implementation in Haskell.

Botworld is a cellular automaton with *robots* and *items*. The robots navigate a grid of cells (some of which may be impassable walls) and manipulate the items. Some items are quite useful: for example, shield items can protect robots from aggressors. Other items are intrinsically valuable, though the values of various items depends upon the game being played.

Among the items are *robot parts*, which the robots can use to construct other robots. Robots may also be broken down into their component parts (hence the necessity for shields). Thus, robots in botworld are quite versatile: a well-programmed robot can reassemble its enemies into allies or construct a robot horde.

Because robots are transient objects, it is important to note that players are not robots. Many games begin by allowing each player to specify the initial state of a single robot, but clever players will write programs that soon distribute themselves across many robots or construct fleets of allied robots. Thus, botworld games are not scored depending upon the actions of the robot. Instead, each player is assigned a home square (or squares), and botworld games are scored according to the contents of the home squares at the end of the game.

Robots cannot see the contents of robot register machines by default, though robots *can* execute an inspection to see the precise state of another robot's register machine. This can lead to interesting games, particularly when the inspecting robot is powerful enough to simulate the targeted robot in full.

It is important to note that there are two different notions of time in botworld. The cellular automaton evolution proceeds in discrete steps according to the rules described below. During each cellular automaton step, the machines inside the robots are run for some finite number of ticks.

Like any cellular automaton, botworld updates in discrete *steps* which apply to every cell. Each cell is updated using only information from the cell and its immediate neighbors. Roughly speaking, the step function proceeds in the following manner for each individual square:

1. The output register of the register machine of each robot in the square is read to determine the robot's *command*. Note that robots are expected to be initialized with their first command in the output register.
2. The commands are used in aggregate to determine the robot *actions*. This involves checking for conflicts and invalid commands.
3. The item list is updated according to the robot actions. Items that have been lifted or used to create robots are removed, items that have been dropped are added.
4. Robots incoming from neighboring squares are added to the robot list.
5. Created robots are added to the robot list.
6. The input registers are set on all robots. Robot input includes a list of all robots in the square (including exiting, entering, destroyed, and created robots), the actions that each robot took, and the updated item list.
7. Robots that have exited the square or that have been destroyed are removed from the robot list.
8. All remaining robots have their register machines executed (and are expected to leave a command in the output register.)

These rules allow for a wide variety of games, from NP-hard knapsack packing games to difficult Newcomb-like games such as a variant of the Parfit's hitchhiker problem (wherein a robot will drop a valuable item only if it, after simulating your robot, concludes that your robot will give it a less valuable item).

3 Implementation

This report is a literate Haskell file, so we must begin the code with the module definition and the Haskell imports.

```
module BotWorld where
import Control.Applicative ((<$>), (<*>))
import Control.Monad (join)
import Control.Monad.Reader (Reader, asks)
import Data.List (delete, elemIndices, intercalate, sortBy)
import Data.List.Split (chunksOf)
import Data.Maybe (catMaybes, isJust, fromMaybe, mapMaybe)
import Data.Ord (comparing)
import Text.Printf (printf)
```

Botworld cells may be either walls (which are immutable and impassible) or *squares*, which may contain both *robots* and *items* which the robots carry and manipulate. We represent cells using the following type:

```
type Cell = Maybe Square
```

The interesting parts of botworld games happen in the squares.

```
data Square = Square
  {robotsIn :: [Robot]
  , itemsIn :: [Item]
  } deriving (Eq, Show)
```

The ordering is arbitrary, but is used by robots to specify the targets of their actions: a robot executing the command *Lift* 3 will attempt to lift the item at index 3 in the item list of its current square.

Botworld, like any cellular automaton, is composed of a grid of cells.

```
type BotWorld = Grid Cell
```

We do not mean to tie the specification of botworld to any particular grid implementation: botworld grids may be finite or infinite, wrapping (pacman style) or non-wrapping. The specific implementation used in this report is somewhat monotonous, and may be found in Appendix A.

3.1 Robots

Each robot can be visualized as a little metal construct on wheels, with a little camera on the front, lifter-arms on the sides, a holding area atop, and a register machine ticking away deep within.

```
data Robot = Robot
  {frame :: Frame
  , inventory :: [Item]
  , processor :: Processor
  , memory :: Memory
  } deriving (Eq, Show)
```

The robot frame is colored (the robots are painted) and has a *strength* which determines the amount of weight that the robot can carry in its inventory.

```
data Frame = F {color :: Color, strength :: Int} deriving (Eq, Show)
```

The color is not necessarily unique, but may help robots distinguish other robots. In this report, colors are represented as a simple small enumeration. Other implementations are welcome to adopt a more fully fledged datatype for representing robot colors.

```
data Color = Red | Orange | Yellow | Green | Blue | Violet | Black | White
  deriving (Eq, Ord, Enum)
```

The frame strength limits the total weight of items that may be carried in the robot's inventory. Every item has a weight, and the combined weight of all carried items must not exceed the frame's strength.

```
canLift :: Robot → Item → Bool
canLift r item = strength (frame r) ⩾ sum (map weight $ item : inventory r)
```

Robots also contain a register machine, which consists of a *processor* and a *memory*. The processor is defined purely by the number of instructions it can compute per botworld step, and the memory is simply a list of registers.

```
newtype Processor = P {speed :: Int} deriving (Eq, Show)
type Memory = [Register]
```

In this report, the register machines use a very simple instruction set which we call the *constree language*. A full implementation can be found in Appendix B. However, when modelling concrete decision problems in botworld, we may choose to replace this simple language by something easier to use. (In particular, many robot programs will need to reason about botworld's laws. Encoding botworld into the constree language is no trivial task.)

3.2 Items

Botworld squares contain *items* which may be manipulated by the robots. Items include *robot parts* which can be used to construct robots, and *shields* which can be used to protect a robot from aggressors, and various types of *cargo*, a catch-all term for items that have no functional significance inside botworld but that players try to collect to increase their score.

At the end of a botworld game, a player is scored on the value of all items carried by robots in the player's *home square*. We may imagine these robots being airlifted and the items in their possession being given to the player. The value of different items varies from game to game; see Section 3.5 for details.

Robot parts are either *processors*, *registers*, or *frames*.

```
data Item
  = Cargo {cargoType :: Int, cargoWeight :: Int}
  | ProcessorPart Processor
  | RegisterPart Register
  | FramePart Frame
  | Shield
  deriving (Eq, Show)
```

Every item has a weight. Shields, registers and processors are light. Frames are heavy. The weight of cargo is variable.

```
weight :: Item → Int
weight (Cargo _ w) = w
weight Shield = 1
weight (RegisterPart _) = 1
weight (ProcessorPart _) = 1
weight (FramePart _) = 100
```

Robots can construct other robots from component parts. Specifically, a robot may be constructed from one frame, one processor, and any number of registers.[1]

```
construct :: [Item] → Maybe Robot
construct parts = do
  FramePart f ← singleton $ filter isFrame parts
  ProcessorPart p ← singleton $ filter isProcessor parts
  let robot = Robot f [] p [r | RegisterPart r ← parts]
  if all isPart parts then Just robot else Nothing
```

[1]The following code introduces the helper function *singleton* :: [*a*] → *Maybe a* which returns *Just x* when given [*x*] and Nothing otherwise, as well as the helper functions *isFrame*, *isProcessor*, *isPart* :: *Item* → *Bool*, all of which are defined in Appendix C.

Robots may also shatter robots into their component parts. As you might imagine, each robot is deconstructed into a frame, a processor, and a handful of registers.

```
shatter :: Robot → [Item]
shatter r = FramePart (frame r) : ProcessorPart (processor r) : rparts where
  rparts = map (RegisterPart ∘ forceR Nil) (memory r)
```

3.3 Commands and actions

Robot machines have a special *output register* which is used to determine the action taken by the robot in the step. Robot machines are run at the *end* of each botworld step, and are expected to leave a command in the output register. This command determines the behavior of the robot in the following step.

Available commands are:

- *Move*, for moving around the grid.
- *Lift*, for lifting items.
- *Drop*, for dropping items.
- *Inspect*, for reading the contents of another robot's register machine.
- *Destroy*, for destroying robots.
- *Build*, for creating new robots.
- *Pass*, which has the robot do nothing.

```
data Command
  = Move Direction
  | Lift Int
  | Drop Int
  | Inspect Int
  | Destroy Int
  | Build [Int] Memory
  | Pass
  deriving Show
```

Depending upon the state of the world, the robots may or may not actually execute their chosen command. For instance, if the robot attempts to move into a wall, the robot will fail. The actual actions that a robot may end up taking are given below. Their meanings will be made explicit momentarily (though you can guess most of them from the names).

```
data Action
  = Created
  | Passed
  | MoveBlocked Direction
  | MovedOut Direction
  | MovedIn Direction
  | CannotFit Int
  | GrappledOver Int
  | Lifted Int
  | Dropped Item
  | InspectTargetFled Int
  | InspectBlocked Int
  | Inspected Int Robot
  | DestroyTargetFled Int
  | DestroyBlocked Int
  | Destroyed Int
  | BuildInterrupted [Int]
  | Built [Int] Robot
  | Invalid
  deriving (Eq, Show)
```

3.4 The step function

Botworld cells are updated given only the current state of the cell and the states of all surrounding cells. Wall cells are immutable, and thus we need only define the step function on squares.

```
step :: Square → [(Direction, Cell)] → Square
```

We begin by computing what each robot would like to do. We do this by reading from (and then zeroing out) the output register of the robot's register machine.

This leaves us both with a list of robots (which have had their machine's output register zeroed out) and a corresponding list of robot outputs.

```
step sq neighbors = Square robots' items' where
  (robots, intents) = unzip $ map takeOutput $ robotsIn sq
```

Notice that we read the robot's output register at the beginning of each botworld step. (We run the robot register machines at the end of each step.) This means that robots must be initialized with their first command in the output register.

Before we can compute the actions that are actually taken by each robot, we need to compute some data that will help us identify failed actions.

Items may only be lifted or used to build robots if no other robot is also validly lifting or using the item. In order to detect such conflicts, we generate a list of items which corresponds by index to the cell's item list, except with contested items missing.

```
itemTargets :: [Maybe Item]
itemTargets = map contest uses where
  uses = validLifts ++ concat validBuilds
```

We determine the indices of items that robots want to lift by looking at all lift orders that the ordering robot could in fact carry out:[2]

[2]The following code introduces the helper function (!!?) :: [a] → *Int* → *Maybe* a, used to safely index into lists, which is defined in Appendix C.

```
validLifts = [i | (r, Lift i) ← orders, isValidLift r i]
isValidLift r i = maybe False (canLift r) (itemsIn sq !!? i)
orders = [(r, cmd) | (r, Just cmd) ← zip robots intents]
```

We then determine the indices of items that robots want to use to build other robots by looking at all build orders that actually do describe a robot:

```
validBuilds = [is | Build is _ ← catMaybes intents, isValidBuild is]
isValidBuild = maybe False (isJust ∘ construct) ∘ mapM (itemsIn sq!!?)
```

We may then determine which items are in high demand, and generate our item list with those items removed.

```
contest i = if i ∈ delete i uses then Nothing else itemsIn sq !!? i
```

Robots may only be destroyed or inspected if they do not possess adequate shields. Every attack (*Destroy* or *Inspect* command) targeting a robot destroys one of the robot's shields. So long as the robot possesses more shields than attackers, the robot is not affected. However, if the robot is attacked by more robots than it has shields, then all of its shields are destroyed *and* all of the attacks succeed (in a wild frenzy, presumably).

To implement this behavior, we generate first a list corresponding by index to the robot list which specifies the number of attacks that each robot receives in this step:

```
attacks :: [Int]
attacks = map numAttacks [0 ..] where
  numAttacks i = length $ filter (≡ i) allAttacks
  allAttacks = mapMaybe (getAttack=≪) intents
  getAttack (Inspect i) = Just i
  getAttack (Destroy i) = Just i
  getAttack _ = Nothing
```

We then generate a list corresponding by index to the robot list which for each robot determines whether that robot is adequately shielded in this step[3]:

```
shielded :: [Bool]
shielded = zipWith isShielded [0 ..] robots where
  isShielded i r = (attacks !! i) ⩽ length (filter isShield $ inventory r)
```

Any robot that exits the square in this step cannot be attacked in this step. Moving robots evade their pursuers. The shields of moving robots are not destroyed. Note that this is not a foolproof defense: a robot cornered by walls had best have some shields handy. Nevertheless, we define a function that determines whether a robot has fled. This function makes use of the fact that movement commands into non-wall cells always succeed.

[3]This function introduces the helper function *isShield* :: *Item* → *Bool* defined in Appendix C.

```
fled :: Maybe Command → Bool
fled (Just (Move dir)) = isJust $ join $ lookup dir neighbors
fled _ = False
```

We may now map robot commands onto the actions that the robots actually take. We begin by noting that any robot with invalid output takes the *Invalid* action.

```
resolve :: Robot → Maybe Command → Action
resolve robot = maybe Invalid act where
```

As we have seen, *Move* commands fail only when the robot attempts to move into a wall cell.

```
  act (Move dir) = (if isJust cell then MovedOut else MoveBlocked) dir
    where cell = join $ lookup dir neighbors
```

Lift commands can fail in three different ways:

1. If the item index is out of range, the command is invalid.
2. If the item is not an available target then multiple robots have attempted to use the same item.
3. If the robot lacks the strength to hold the item, the lift fails.

Otherwise, the lift succeeds.

```
  act (Lift i) = maybe Invalid tryLift $ itemTargets !!? i where
    tryLift = maybe (GrappledOver i) pickUp
    pickUp item = (if canLift robot item then Lifted else CannotFit) i
```

Drop commands always succeed so long as the robot actually possesses the item they attempt to drop.

```
  act (Drop i) = maybe Invalid Dropped (inventory robot !!? i)
```

Inspect commands, like *Lift* commands, may fail in three different ways:

1. If the specified robot does not exist, the command is invalid.
2. If the specified robot moved away, the inspection fails.
3. If the specified robot had sufficient shields this step, the inspection is blocked.

Otherwise, the inspection succeeds.

```
  act (Inspect i) = maybe Invalid tryInspect (robots !!? i) where
    tryInspect target
        | fled (intents !! i) = InspectTargetFled i
        | shielded !! i = InspectBlocked i
        | otherwise = Inspected i target
```

Destroy commands are similar to inspect commands: if the given index actually specifies a victim in the robot list, and the victim is not moving away, and the victim is not adequately shielded, then the victim is destroyed.

Robots *can* destroy themselves. Programs should be careful to avoid unintentional self-destruction.

```
act (Destroy i) = maybe Invalid tryDestroy (robots !!? i) where
  tryDestroy _
    | fled (intents !! i) = DestroyTargetFled i
    | shielded !! i = DestroyBlocked i
    | otherwise = Destroyed i
```

Build commands must also pass three checks in order to succeed:

1. All of the specified indexes must specify actual items.
2. All of the specified items must not be contested.
3. The items must together specify a robot.

```
act (Build is m) = maybe Invalid tryBuild parts where
  parts = mapM (itemTargets!!?) is
  tryBuild = maybe (BuildInterrupted is) buildUsing ∘ sequence
  buildUsing = maybe Invalid (Built is ∘ initialize m) ∘ construct
```

Pass commands always succeed.

```
act Pass = Passed
```

With the *resolve* function in hand it is trivial to compute the actions actually executed by the robots in the square:

```
localActions :: [Action]
localActions = zipWith resolve robots intents
```

With this data we can determine which robots left the square and which robots were destroyed. It is convenient to know, for each robot which began in this square, whether that robot is still in this square and (if they are) whether they survived. We store that data in the following list (which corresponds by index to the original robot list):

```
survived :: [Maybe Bool]
survived = zipWith check [0 ..] localActions where
  check _ (MovedOut _) = Nothing
  check n _ = Just $ n ∉ [i | Destroyed i ← localActions]
```

We then compute the updated inventories of all robots who began in this square. (The inventories of moving robots are not changed, so we need not update the inventory of robots entering this square.)

Robot inventories are updated whenever the robot executes a *Lift* action, executes a *Drop* action, or experiences an attack (in which case shields may be destroyed.) Notice that shields are destroyed in order according to the ordering of the targeted robot's inventory.[4]

[4]The following code introduces the helper function *dropN* :: *Int* → (*a* → *Bool*) → [*a*] → [*a*], which drops the first *n* items matching the given predicate. It is defined in Appendix C.

```
updateInventory :: Int → Action → Robot → Robot
updateInventory i a r = case a of
  MovedOut _ → r
  Lifted n → r {inventory = (itemsIn sq !! n) : defended}
  Dropped item → r {inventory = delete item defended}
  _ → r {inventory = defended}
  where defended = dropN (attacks !! i) isShield $ inventory r
```

We use this function to update the inventories of all robots that were originally in this square. Notice that the inventories of destroyed robots are updated as well: destroyed robots get to perform their actions before they are destroyed.

```
veterans :: [Robot]
veterans = zipWith3 updateInventory [0 ..] localActions robots where
```

Next, we identify which robots enter this square from other squares. We compute this by looking at the intents of the robots in neighboring squares. Remember that move commands always succeed if the robot is moving into a non-wall square. Thus, all robots in neighboring squares which intend to move into this square will successfully move into this square.

```
incomingFrom :: (Direction, Cell) → [(Robot, Direction)]
incomingFrom (dir, neighbor) = mapMaybe movingThisWay cmds where
  cmds = maybe [] (map takeOutput ∘ robotsIn) neighbor
  movingThisWay (robot, Just (Move dir′))
    | dir ≡ opposite dir′ = Just (robot, dir)
  movingThisWay _ = Nothing
```

We compute both a list of entering robots and a corresponding list of the directions which those robots entered from.

```
(travelers, origins) = unzip $ concatMap incomingFrom neighbors
```

We also determine the list of robots that have been created in this timestep:

```
children = [r | Built _ r ← localActions]
```

All remaining robots will have their register machines run before the next step. Before they may be run, however, their input registers must be updated. Each robot recieves five inputs:

1. The host robot's index in the following list
2. The list of all robots in the square, including robots that exited, entered, were destroyed, and were created.
3. A list of actions for each robot, corresponding to the list above.
4. The updated item list¿
5. Some private input.

We have already largely computed the list of all robots. It is worth noting here that when this robot list is converted into machine input, some information will be lost: processors and memories are not visible to other robots (except via *Inspect* commands). This data-hiding is implemented by the constree encoding code; see Appendix B.2 for details.

```
allRobots :: [Robot]
allRobots = veterans ++ travelers ++ children
```

Computing the list of all actions is similarly simple. As with the robot list, some of this data will be lost when it is converted into machine input. Specifically, robots cannot distinguish between *Passed* and *Invalid* actions. Also, the results of an *Inspect* command are visible only to the inspecting robot. Again, this data-hiding is implemented by the constree encoding code; see Appendix B.2 for details.

```
allActions :: [Action]
allActions = localActions ++ travelerActions ++ childActions where
  travelerActions = map MovedIn origins
  childActions = replicate (length children) Created
```

We now compute the item list. It is given in three groups.
The items that were unaffected:

```
unaffected :: [Item]
unaffected = removeIndices (lifts ++ concat builds) (itemsIn sq) where
  lifts = [i | Lifted i ← localActions]
  builds = [is | Built is _ ← localActions]
```

The items that were willingly dropped by robots:

```
dropped :: [Item]
dropped = [item | Dropped item ← localActions]
```

And the fallen items from destroyed robots, which is given in groups of part/inventory pairs:

```
fallen :: [([Item], [Item])]
fallen = [itemsOf r | (r, Just False) ← zip veterans survived] where
  itemsOf r = (shatter r, filter (¬ ∘ isShield) (inventory r))
```

The item list retains some structure when it is encoded as robot input, which helps robots determine what happened to which items.

The final piece of robot input is private. If the robot executed a successful *Inspect* command then the private input includes information about the inspected robot's machine.

Also, the private input differentiates between *Invalid* and *Passed* actions in a private fashion, so that each individual machine can know whether *it itself* gave an invalid command in the previous step. (All other robots cannot distinguish between *Invalid* and *Passed* actions.)

```
privateInput :: Action → Constree
privateInput Invalid = encode (1 :: Int)
privateInput (Inspected _ r) = encode
  (processor r, length $ memory r, memory r)
privateInput _ = encode (0 :: Int)
```

With these inputs in hand, we can run any given robot by updating their input register appropriately and then running the robot's register machine:

```
run :: Int → Action → Robot → Robot
run index action robot = runMachine $ setInput robot input where
  input = (index, allRobots, allActions, items, privateInput action)
  items = (unaffected, dropped, fallen)
```

The register machines are run as described in the following function. It makes use of the constree register machine; refer to Appendix B for details.

```
runMachine :: Robot → Robot
runMachine robot = case runFor (speed $ processor robot) (memory robot) of
  Right memory′ → robot {memory = memory′}
  Left _ → robot {memory = map (forceR Nil) (memory robot)}
```

We only run robots that both stayed in the square and were not destroyed. We figure out which robots stayed and survived according to their index in the list of all robots.

We can look this data up in the *survived* list created previously, remembering that all indices which don't show up in the list denote robots that either entered or were created (and that all such robots are present).

```
present :: Int → Bool
present = maybe True (fromMaybe False) ∘ (survived!!?)
```

We then construct the new robot list by running all present robots.

```
robots′ :: [Robot]
robots′ = [run i a r | (i, a, r) ← triples, present i] where
  triples = zip3 [0 ..] allActions allRobots
```

It remains only to specify the updated item list. This is the same as the updated item list that was passed to the robots as input, with the additional structure removed.

```
items′ :: [Item]
items′ = unaffected ++ dropped ++ concat [xs ++ ys | (xs, ys) ← fallen]
```

This fully specifies the step function for botworld cells. To reiterate:

1. Robot machine output registers are read to determine robot intents.
2. Robot actions are computed from robot intents.
3. Robot inventories are updated.
4. Incoming robots are computed.
5. Unmoved, dropped, and fallen items are computed.
6. Destroyed robots are removed, constructed robots are added.
7. Machine input registers are set according to the updated state.
8. Robot register machines are executed (and are expected to leave a command in the output register for the next step).
9. The updated item list is constructed.

3.5 Games

Botworld games can vary widely. A simple game that botworld lends itself to easily is a knapsack game, in which players attempt to maximize the value of the items collected by robots which they control. (This is an NP-hard problem in general.)

Remember that *robots are not players*: a player may only be able to specify the initial program for a single robot, but players may well attempt to acquire whole fleets of robots with code distributed throughout.

As such, botworld games are not scored according to the possessions of any particular robot. Rather, each player is assigned a *home square*, and the score of a player is computed according to the items possessed by all robots in the player's home square at the end of the game. (The robots are airlifted out and their items are extracted for delivery to the player.) Thus, a game configuration also needs to assign specific values to the various items.

Formally, we define a game configuration as follows:

```
data GameConfig = GameConfig
  {players :: [(Position, String)]
  , valuer :: Item → Int
  }
```

With a game configuration in hand, we can compute how many points a single robot has achieved:

```
points :: Robot → Reader GameConfig Int
points r = (λvalue → sum (map value $ inventory r)) <$> asks valuer
```

Then we can compute the total score in any particular square:

```
score :: BotWorld → Position → Reader GameConfig Int
score g = maybe (return 0) (fmap sum ◦ mapM points ◦ robotsIn) ◦ at g
```

We do not provide any example games in this report. Some example games are forthcoming.

4 Concluding notes

Botworld allows us to study self-modifying agents in a world where the agents are embedded *within* the environment. Botworld admits a wide variety of games, including games with Newcomb-like problems and games with NP-hard tasks.

Botworld provides a very concrete environment in which to envision agents. This has proved quite useful when considering obstacles of self-reference: the concrete model often makes it easier to envision difficulties and probe edge cases.

Furthermore, botworld allows us to constructively illustrate issues that we come across by providing a concrete game in which the issue presents itself. This can often help make the abstract problems of self-reference easier to visualize.

Forthcoming papers will illustrate some of the discoveries that we've made using botworld.

A Grid Manipulation

This report uses a quick-and-dirty *Grid* implementation wherein a grid is represented by a flat list of cells. This grid implementation specifies a wraparound grid (pacman style), which means that every position is valid.

Botworld is not tied to this particular grid implementation: non-wrapping grids, infinite grids, or even non-euclidean grids could house botworld games. We require only that squares agree on who their neighbors are: if square A is north of square B, then square B must be south of square A.

```
type Dimensions = (Int, Int)
type Position = (Int, Int)
data Grid a = Grid
  { dimensions :: Dimensions
  , cells :: [a]
  } deriving Eq
locate :: Dimensions -> Position -> Int
locate (x, y) (i, j) = (j `mod` y) * x + (i `mod` x)
indices :: Grid a -> [Position]
indices (Grid (x, y) _) = [(i, j) | j <- [0 .. pred y], i <- [0 .. pred x]]
at :: Grid a -> Position -> a
at (Grid dim xs) p = xs !! locate dim p
change :: (a -> a) -> Position -> Grid a -> Grid a
change f p (Grid dim as) = Grid dim $ alter (locate dim p) f as
generate :: Dimensions -> (Position -> a) -> Grid a
generate dim gen = let g = Grid dim (map gen $ indices g) in g
```

A.1 Directions

Each square has eight neighbors (or up to eight neighbors, in finite non-wrapping grids). Each neighbor lies in one of eight directions, termed according to the cardinal directions. We now formally name those directions and specify how directions alter grid positions.

```
data Direction = N | NE | E | SE | S | SW | W | NW
  deriving (Eq, Ord, Enum, Show)
opposite :: Direction -> Direction
opposite d = iterate (if d < S then succ else pred) d !! 4
towards :: Direction -> Position -> Position
towards d (x, y) = (x + dx, y + dy) where
  dx = [0, 1, 1, 1, 0, -1, -1, -1] !! fromEnum d
  dy = [-1, -1, 0, 1, 1, 1, 0, -1] !! fromEnum d
```

A.2 Botworld Grids

Finally, we define a function that updates an entire botworld grid by one step:

```
update :: BotWorld → BotWorld
update g = g {cells = map doStep $ indices g} where
  doStep pos = flip step (fellows pos) <$> at g pos
  fellows pos = map (walk pos) [N ..]
  walk p d = (d, at g $ towards d p)
```

B Constree Language

Robots contain register machines, which run a little Turing complete language which we call the *constree language.* There is only one data structure in constree, which is (unsurprisingly) the cons tree:

```
data Constree = Cons Constree Constree | Nil deriving (Eq, Show)
```

Constrees are stored in registers, each of which has a memory limit.

```
data Register = R {limit :: Int, contents :: Constree} deriving (Eq, Show)
```

Each tree has a size determined by the number of conses in the tree. It may be more efficient for the size of the tree to be encoded directly into the *Cons*, but we are optimizing for clarity over speed, so we simply compute the size whenever it is needed.

A tree can only be placed in a register if the size of the tree does not exceed the size limit on the register.

```
size :: Constree → Int
size Nil = 0
size (Cons t1 t2) = succ $ size t1 + size t2
```

Constrees are trimmed from the right. This is important only when you try to shove a constree into a register where the constree does not fit.

```
trim :: Int → Constree → Constree
trim _ Nil = Nil
trim x t@(Cons front back)
   | size t ⩽ x = t
   | size front < x = Cons front $ trim (x − succ (size front)) back
   | otherwise = Nil
```

There are two ways to place a tree into a register: you can force the tree into the register (in which case the register gets set to nil if the tree does not fit), or you can fit the tree into the register (in which case the tree gets trimmed if it does not fit).

```
forceR :: Constree → Register → Register
forceR t r = if size t ⩽ limit r then r {contents = t} else r {contents = Nil}
fitR :: Encodable i ⇒ i → Register → Register
fitR i r = forceR (trim (limit r) (encode i)) r
```

The constree language has only four instructions:

1. One to make the contents of a register nil.
2. One to cons two registers together into a third register.
3. One to deconstruct a register into two other registers.
4. One to conditionally copy one register into another register, but only if the test register is nil.

```
data Instruction
  = Nilify Int
  | Construct Int Int Int
  | Deconstruct Int Int Int
  | CopyIfNil Int Int Int
  deriving (Eq, Show)
```

A machine is simply a list of such registers. The first register is the program register, the second is the input register, the third is the output register, and the rest are workspace registers.

The following code implements the above construction set on a constree register machine:

```
data Error
  = BadInstruction Constree
  | NoSuchRegister Int
  | DeconstructNil Int
  | OutOfMemory Int
  | InvalidOutput
  deriving (Eq, Show)
getTree :: Int → Memory → Either Error Constree
getTree i m = maybe (Left $ NoSuchRegister i) (Right ∘ contents) (m !!? i)
setTree :: Constree → Int → Memory → Either Error Memory
setTree t i m = maybe (Left $ NoSuchRegister i) go (m !!? i) where
  go r = if size t > limit r then Left $ OutOfMemory i else
    Right $ alter i (const r {contents = t}) m
execute :: Instruction → Memory → Either Error Memory
execute instruction m = case instruction of
  Nilify tgt → setTree Nil tgt m
  Construct fnt bck tgt → do
    front ← getTree fnt m
    back ← getTree bck m
    setTree (Cons front back) tgt m
  Deconstruct src fnt bck → case getTree src m of
    Left err → Left err
    Right Nil → Left $ DeconstructNil src
    Right (Cons front back) → setTree front fnt m ≫= setTree back bck
  CopyIfNil tst src tgt → case getTree tst m of
    Left err → Left err
    Right Nil → getTree src m ≫= (λt → setTree t tgt m)
    Right _ → Right m
```

```
runFor :: Int → Memory → Either Error Memory
runFor 0 m = Right m
runFor _ [ ] = Right [ ]
runFor _ (r : rs) | contents r ≡ Nil = Right $ r : rs
runFor n (r : rs) = tick ≫= runFor (pred n) where
  tick = maybe badInstruction doInstruction (decode $ contents r)
  badInstruction = Left $ BadInstruction $ contents r
  doInstruction (i, is) = execute i (r {contents = is} : rs)
```

B.1 Robot/machine interactions

Aside from executing robot machines, there are three ways that botworld changes a robot's register machines:

A robot may have its machine written. This happens whenever the machine is constructed.

```
initialize :: Memory → Robot → Robot
initialize m robot = robot {memory = fitted} where
  fitted = zipWith (forceR ∘ contents) m (memory robot) ++ padding
  padding = map (forceR Nil) (drop (length m) (memory robot))
```

A robot may have its output register read. Whenever the output register is read, it is set to *Nil* thereafter.

Programs may use this fact to implement a wait-loop that waits until output is read before proceeding: after output is read, input will be updated before the next instruction is executed, so machines waiting for a *Nil* output can be confident that when the output register becomes *Nil* there will be new input in the input register.

A robot's output register is read at the beginning of eac htick.

```
takeOutput :: Decodable o ⇒ Robot → (Robot, Maybe o)
takeOutput robot = maybe (robot, Nothing) go (m !!? 2) where
  go o = (robot {memory = alter 2 (forceR Nil) m}, decode $ contents o)
  m = memory robot
```

A robot may have its machine input register set. This happens just before the machine is executed in every botworld step.

```
setInput :: Encodable i ⇒ Robot → i → Robot
setInput robot i = robot {memory = set1} where
  set1 = alter 1 (fitR i) (memory robot)
```

B.2 Encoding and Decoding

The following section specifies how haskell data structures are encoded into constrees and decoded from constrees. It is largely mechanical, with a few exceptions noted inline.

```
class Encodable t where
  encode :: t → Constree
class Decodable t where
  decode :: Constree → Maybe t
instance Encodable Constree where
  encode = id
instance Decodable Constree where
  decode = Just
instance Encodable t ⇒ Encodable (Maybe t) where
  encode = maybe Nil (Cons Nil ∘ encode)
instance Decodable t ⇒ Decodable (Maybe t) where
  decode Nil = Just Nothing
  decode (Cons Nil x) = Just <$> decode x
  decode _ = Nothing
instance Encodable t ⇒ Encodable [t] where
  encode = foldr (Cons ∘ encode) Nil
instance Decodable t ⇒ Decodable [t] where
  decode Nil = Just [ ]
  decode (Cons t1 t2) = (:) <$> decode t1 <*> decode t2
```

Lisp programmers may consider it more parsimonious to encode tuples like lists, with a Nil at the end. There is some sleight of hand going on here, however: machine inputs are encoded tuples, and the inputs may sometimes need to be trimmed to fit into a register. If a robot has executed an *Inpsect* command, then the entire contents of the inspected robot will be dumped into the inspector's input register. In many cases, the entire memory of the target robot is not likely to fit into the input register of the inspector. In such cases, we would like as many full encoded registers to be fit into the input as possible.

Because cons trees are trimmed from the right, we get this behavior for free if we forgo the terminal *Nil* when encoding tuple objects. With this implementation, the memory of the inspected robot (which is a list) will be the rightmost item in the cons tree, and if it does not fit, the registers will be lopped off one at a time. (By contrast, if we Nil-terminated tuple encodings and the machine did not fit, then the entire machine would be trimmed.)

```
instance (Encodable a, Encodable b) ⇒ Encodable (a, b) where
  encode (a, b) = Cons (encode a) (encode b)
instance (Decodable a, Decodable b) ⇒ Decodable (a, b) where
  decode (Cons a b) = (, ) <$> decode a <*> decode b
  decode Nil = Nothing
instance (Encodable a, Encodable b, Encodable c) ⇒ Encodable (a, b, c) where
  encode (a, b, c) = encode (a, (b, c))
instance (Decodable a, Decodable b, Decodable c) ⇒ Decodable (a, b, c) where
```

```
  decode = fmap flatten ∘ decode where flatten (a, (b, c)) = (a, b, c)
instance (Encodable a, Encodable b, Encodable c, Encodable d, Encodable e) ⇒
  Encodable (a, b, c, d, e) where
  encode (a, b, c, d, e) = encode (a, (b, (c, (d, e))))
instance Encodable Bool where
  encode False = Nil
  encode True = Cons Nil Nil
instance Decodable Bool where
  decode Nil = Just False
  decode (Cons Nil Nil) = Just True
  decode _ = Nothing
instance Encodable Int where
  encode n
    | n < 0 = Cons (Cons Nil (Cons Nil Nil)) (encode $ negate n)
    | otherwise = encode $ bits n
    where
      bits 0 = [ ]
      bits x = let (q, r) = quotRem x 2 in (r ≡ 1) : bits q
instance Decodable Int where
  decode (Cons (Cons Nil (Cons Nil Nil)) n) = fmap negate $ decode n
  decode t = unbits <$> decode t where
    unbits [ ] = 0
    unbits (x : xs) = (if x then 1 else 0) + 2 ∗ unbits xs
instance Encodable Instruction where
  encode instruction = case instruction of
    Nilify tgt                → encode (0 :: Int, tgt)
    Construct fnt bck tgt     → encode (1 :: Int, (fnt, bck, tgt))
    Deconstruct src fnt bck   → encode (2 :: Int, (src, fnt, bck))
    CopyIfNil tst src tgt     → encode (3 :: Int, (tst, src, tgt))
instance Decodable Instruction where
  decode t = case decode t :: Maybe (Int, Constree) of
    Just (0, arg)  → Nilify <$> decode arg
    Just (1, args) → uncurry3 Construct <$> decode args
    Just (2, args) → uncurry3 Deconstruct <$> decode args
    Just (3, args) → uncurry3 CopyIfNil <$> decode args
    _              → Nothing
    where uncurry3 f (a, b, c) = f a b c
instance Encodable Register where
  encode r = encode (limit r, contents r)
instance Decodable Register where
  decode = fmap (uncurry R) ∘ decode
instance Encodable Color where
  encode = encode ∘ fromEnum
instance Encodable Frame where
  encode (F c s) = encode (c, s)
instance Encodable Processor where
  encode (P s) = encode s
```

```
instance Encodable Item where
  encode (Cargo t w)         = encode (0 :: Int, t, w)
  encode (RegisterPart r)    = encode (1 :: Int, r)
  encode (ProcessorPart p)   = encode (2 :: Int, p)
  encode (FramePart f)       = encode (3 :: Int, f)
  encode Shield              = encode (4 :: Int, Nil)
instance Encodable Direction where
  encode = encode ∘ fromEnum
instance Decodable Direction where
  decode t = ([N ..]!!?) =≪ decode t
```

Note that only the robot's frame and inventory are encoded into contree. The processor and memory are omitted, as these are not visible in the machine inputs.

```
instance Encodable Robot where
  encode (Robot f i _ _) = encode (f, i)
instance Encodable Command where
  encode (Move d)      = encode (0 :: Int, head $ elemIndices d [N ..])
  encode (Lift i)      = encode (1 :: Int, i)
  encode (Drop i)      = encode (2 :: Int, i)
  encode (Inspect i)   = encode (3 :: Int, i)
  encode (Destroy i)   = encode (4 :: Int, i)
  encode (Build is m)  = encode (5 :: Int, is, m)
  encode Pass          = encode (6 :: Int, Nil)


instance Decodable Command where
  decode t = case decode t :: Maybe (Int, Constree) of
    Just (0, d)   → Move <$>(([N ..]!!?) =≪ decode d)
    Just (1, i)   → Lift <$> decode i
    Just (2, i)   → Drop <$> decode i
    Just (3, i)   → Inspect <$> decode i
    Just (4, i)   → Destroy <$> decode i
    Just (5, x)   → uncurry Build <$> decode x
    Just (6, Nil) → Just Pass
    _             → Nothing
```

Note that *Passed* actions and *Invalid* actions are encoded identically: robots cannot distinguish these actions. Note also that *Inspected* actions do not encode the result of the inspection.

```
instance Encodable Action where
  encode a = case a of
    Passed          → encode (0 :: Int, Nil)
    Invalid         → encode (0 :: Int, Nil)
    Created         → encode (1 :: Int, Nil)
    MoveBlocked d   → encode (4 :: Int, direction d)
    MovedOut d      → encode (2 :: Int, direction d)
    MovedIn d       → encode (3 :: Int, direction d)
```

```
CannotFit i          → encode (6 :: Int, i)
GrappledOver i       → encode (7 :: Int, i)
Lifted i             → encode (5 :: Int, i)
Dropped _            → encode (8 :: Int, Nil)
InspectTargetFled i  → encode (9 :: Int, i)
InspectBlocked i     → encode (10 :: Int, i)
Inspected i _        → encode (11 :: Int, i)
DestroyTargetFled i  → encode (12 :: Int, i)
DestroyBlocked i     → encode (13 :: Int, i)
Destroyed i          → encode (14 :: Int, i)
Built is _           → encode (15 :: Int, is)
BuildInterrupted is  → encode (16 :: Int, is)
where direction d = head $ elemIndices d [N ..]
```

C Helper Functions

This section contains simple helper functions used to implement the botworld step function. Three are used to distinguish different types of items, and one is used to distinguish a specific type of action:

```
isPart :: Item → Bool
isPart (RegisterPart _) = True
isPart item = isProcessor item ∨ isFrame item
isProcessor :: Item → Bool
isProcessor (ProcessorPart _) = True
isProcessor _ = False
isFrame :: Item → Bool
isFrame (FramePart _) = True
isFrame _ = False
isShield :: Item → Bool
isShield Shield = True
isShield _ = False
```

The other four are generic functions that assist with list manipulation: one to extract a single item from a list (or fail if the list has many items):

```
singleton :: [a] → Maybe a
singleton [x] = Just x
singleton _ = Nothing
```

one to safely access items in a list at a given index:

```
(!!?) :: [a] → Int → Maybe a
[] !!? _ = Nothing
(x: _) !!? 0 = Just x
(_ : xs) !!? n = xs !!? pred n
```

one to safely alter a specific item in a list:

```
alter :: Int → (a → a) → [a] → [a]
alter i f xs = maybe xs go (xs !!? i) where
  go x = take i xs ++ (f x : drop (succ i) xs)
```

one to remove a specific set of indices from a list:

```
removeIndices :: [Int] → [a] → [a]
removeIndices = flip $ foldr remove where
  remove :: Int → [a] → [a]
  remove i xs = take i xs ++ drop (succ i) xs
```

and one to selectively drop the first n items that match the given predicate.

```
dropN :: Int → (a → Bool) → [a] → [a]
dropN 0 _ xs = xs
dropN n p (x : xs) = if p x then dropN (pred n) p xs else x : dropN n p xs
dropN _ _ [] = []
```

D Visualization

The remaining code implements a visualizer for botworld grids. This allows you to print out botworld grids and botworld scoreboards (assuming that you have access to a botworld game configuration).

In botworld grid visualizations, colors are given a three-letter code:

```
instance Show Color where
  show Red = "RED"
  show Orange = "RNG"
  show Yellow = "YLO"
  show Green = "GRN"
  show Blue = "BLU"
  show Violet = "VLT"
  show Black = "BLK"
  show White = "WYT"
```

Each cell is shown using three lines: the first for items, the second for item weights, the third for robots (by color). At most two things are shown per row. (This is by no means a perfect visualization, but it works well for simple games.)

```
visualize :: BotWorld → Reader GameConfig String
visualize g = do
  rowStrs ← mapM showRow rows :: Reader GameConfig [String]
  return $ concat rowStrs ++ line
  where
    unpaddedRows = chunksOf r (cells g) where (r, _) = dimensions g
    pad row = row ++ replicate (maxlen − length row) Nothing
    rows = map pad unpaddedRows
    maxlen = maximum (map length unpaddedRows)
    line = concat (replicate maxlen "+---------") ++ "+\n"
```

Items are crudely shown as follows:

```
showValue :: Item → Reader GameConfig String
showValue b = do
  value ← asks valuer
  return $ case b of
    FramePart (F Red _)      → "[R]"
    FramePart (F Orange _)   → "[O]"
    FramePart (F Yellow _)   → "[Y]"
    FramePart (F Green _)    → "[G]"
    FramePart (F Blue _)     → "[B]"
    FramePart (F Violet _)   → "[V]"
    FramePart (F Black _)    → "[K]"
    FramePart (F White _)    → "[W]"
    ProcessorPart _          → "[#]"
    RegisterPart _           → "[|]"
    Shield                   → "\\X/"
    x → printf "$%d" (value x)
showWeight :: Item → String
showWeight item
  | weight item > 99 = "99+"
  | otherwise = printf "%dg" $ weight item
showRow :: [Cell] → Reader GameConfig String
showRow xs = do
  v ← showCells cellValue xs
  w ← showCells cellWeight xs
  r ← showCells (return <$> cellRobots) xs
  return $ line ++ v ++ w ++ r
showCells strify xs = do
  strs ← mapM (maybe (return "//////////") strify) xs
  return $ "|" ++ intercalate "|" strs ++ "|\n"
cellValue sq = do
  value ← asks valuer
  case sortBy (flip $ comparing value) (itemsIn sq) of
    [] → return "          "
    [b] → printf "   %3s    " <$> showValue b
    [b, c] → printf " %3s  %3s " <$> showValue b <*> showValue c
    (b : c: _) → printf " %3s  %3s\x2026" <$> showValue b <*> showValue c
cellWeight sq = do
  value ← asks valuer
  return $ case sortBy (flip $ comparing value) (itemsIn sq) of
    [] → "          "
    [b] → printf "   %3s    " (showWeight b)
    [b, c] → printf " %3s  %3s " (showWeight b) (showWeight c)
    (b : c: _) → printf " %3s  %3s\x2026" (showWeight b) (showWeight c)
cellRobots sq = case sortBy (comparing $ color ∘ frame) (robotsIn sq) of
  [] → "          "
  [f] → printf "    %s    " (clr f)
  [f, s] → printf "  %s  %s  " (clr f) (clr s)
```

```
        (f : s: _) → printf " %s %s\x2026" (clr f) (clr s)
        where clr = show ∘ color ∘ frame
```

Finally, the scoreboard function takes a game configuration and prints out a scoreboard detailing the scores of each player (broken down according to the robots in the player's home square at the end of the game).

```
scoreboard :: BotWorld → Reader GameConfig String
scoreboard g = do
  scores ← mapM scoreCell =≪ sortedPositions
  return $ unlines $ concat scores
  where
    sortedPositions = do
      ps ← map fst <$> asks players
      scores ← mapM (score g) ps
      let comparer = flip $ comparing snd
      return $ map fst $ sortBy comparer $ zip ps scores
    scoreCell p = do
      header ← playerLine p
      let divider = replicate (length header) '-'
      breakdown ← case maybe [] robotsIn $ at g p of
        [] → return ["  No robots in square."]
        rs → mapM robotScore rs
      return $ header : divider : breakdown
    robotScore r = do
      pts ← points r
      let name = printf "   %s robot" (show $ color $ frame r) :: String
      return $ name ++ ": $" ++ printf "%d" pts
    playerLine p = do
      total ← score g p
      name ← lookup p <$> asks players
      let moniker = fromMaybe (printf "Player at %s" (show p)) name
      return $ printf "%s $%d" moniker total
```