# Botworld 1.1
(Technical Report)

Nate Soares, Benja Fallenstein

Machine Intelligence Research Institute
2030 Addison St. #300
Berkeley, CA 94704, USA

{nate,benja}@intelligence.org

April 10, 2014

# Contents

# Chapter 1

# Introduction

This report introduces *Botworld*, a cellular automaton that provides a toy environment for studying self-modifying agents.

The traditional agent framework, used for example in Markov Decision Processes [8] and in Marcus Hutter's universal agent AIXI [4], splits the universe into an agent and an environment, which interact only via discrete input and output channels.

Such formalisms are perhaps ill-suited for real self-modifying agents, which are embedded within their environments [5]. Indeed, the agent/environment separation is somewhat reminiscent of Cartesian dualism: any agent using this framework to reason about the world does not model itself as part of its environment. For example, such an agent would be unable to understand the concept of the environment interfering with its internal computations, e.g. by inducing errors in the agent's RAM through heat [3].

Intuitively, this separation does not seem to be 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. However, the concrete non-Cartesian models that have been proposed (such as Orseau and Ring's formalism for space-time embedded intelligence [5], Vladimir Slepnev's models of updateless decision theory [6, 7], and Yudkowsky and Herreshoff's *tiling agents* [9]) depart significantly from their Cartesian counterparts.

Botworld is a toy example of the type of universe that these formalisms are designed to reason about: it provides a concrete world containing agents ("robots") whose internal computations are a part of the environment, and allows us to study what happens when the Cartesian barrier between an agent and its environment breaks down. Botworld allows us to write decision problems where the Cartesian barrier is relevant, program actual agents, and run the system.

As it turns out, many interesting problems arise when agents are embedded in their environment. For example, agents whose source code is readable may be subjected to Newcomb-like problems [1] by entities that simulate the agent and choose their actions accordingly.

Furthermore, certain obstacles to self-reference arise when non-Cartesian agents attempt to achieve confidence in their future actions. Some of these issues are raised by Yudkowsky and Herreshoff [9]; 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 concrete decision problem ("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.

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.

## 1.1 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 consists of a grid of cells, each of which is either a *square* or an impassable *wall*. Each square may contain an arbitrary number of *robots* and *items*. Robots can navigate the grid and possess tools for manipulating items. Some items are quite useful: for example, *shields* can protect robots from attacks by other robots. 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 items carried by all robots that are in the player's home square at the end of the game. (We may imagine these robots being airlifted and the items in their possession being given to the player.)

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 is one way in which the Cartesian boundary can break down: It may not be enough to choose an optimal *action*, if the way in which this action is *computed* can matter.

For example, imagine a robot which tries to execute an action that it can prove will achieve a certain minimum expected utility $u_{\min}$. In the traditional agent framework, this can imply an optimality property: if there is *any* program $p$ our robot could have run such that our robot can prove that $p$ would have received expected utility $\geq u_{\min}$, then our robot will receive expected utility $\geq u_{\min}$ (because it can always do what that other program would have done). But suppose that this robot is placed into an environment where another robot reads the contents of the first robot's register machine, and gives the first robot a reward *if and only if the first robot runs the program "do nothing ever"*. Then, since this is not the program our robot runs, it will not receive the reward.

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 list of items lying around in the square 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. Newly 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).

## 1.2 Cartesianism in Botworld

Though we have stated that we mean to study non-Cartesian formalizations of intelligence, Botworld does in fact have a "Cartesian" boundary. The robot parts are fundamental objects, the machine registers are non-reducible. The important property of Botworld is not that it lacks a Cartesian boundary, but that the boundary is *breakable*.

In the real world the execution of a computer program is unaffected by the environment *most* of the time (except via the normal input channels). While the contents of a computer's RAM *can* be changed by heating it up with a desk lamp [3], they are usually not. An Artificial General Intelligence (AGI) would presumably make use of this fact. Thus, an AGI may commonly wish to ensure that its Cartesian boundary is not violated in this way over some time period (during which it can make use of the nice properties of Cartesian frameworks). Botworld attempts to model this in a simple way by requiring agents to contend with the possibility that they may be destroyed by other robots.

More problematically, in the real world, the internals of a computer program will always affect the environment—for example, through waste heat emitted by the computer—but it seems likely that these effects are usually unpredictable enough that an AGI will not be able to improve its performance by carefully choosing e.g. the pattern of waste heat it emits. However, an AGI will need to ensure that these unavoidable violations of its Cartesian boundary will *in fact* not make an expected difference to its goals. Botworld sidesteps this issue and only requires robots to deal with a more tractable issue: Contending with the possibility that their source code might be read by another agent.

Our model is not realistic, but it is simple to reason about. For all that the robot machines are not reducible, the robots are still embedded in their environment, and they can still be read or destroyed by other agents. We hope that this captures some of the complexity of naturalistic agents, and that it will serve as a useful test bed for formalisms designed to deal with this complexity. Although being able to deal with the challenges of Botworld is presumably not a good indicator that a formalism will be able to deal with *all* of the challenges of naturalistic agents, it allows us to see in concrete terms how it deals with some of them.

In creating Botworld we tried to build something implementable by a lower-level system, such as Conway's *Game of Life* [2]. It is useful to imagine such an implementation when considering Botworld games.

Future versions of Botworld may treat the robot bodies as less fundamental objects. In the meantime, we hope that it is possible to picture an implementation where the Cartesian boundary is much less fundamental, and to use Botworld to gain useful insights about agents embedded within their environment. Our intent is that when we apply a formalism for naturalistic agents to the current implementation of Botworld, then there will be a straightforward translation to an application of the same formalism to an implementation of Botworld in (say) the Game of Life.

# Chapter 2

# 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 Prelude hiding (lookup)
import Control.Applicative ((<$>), (<*>))
import Control.Monad (join)
import Data.List (delete, elemIndices)
import Data.Map (Map, assocs, fromList, lookup, mapWithKey)
import Data.Maybe (catMaybes, isJust, mapMaybe)
```

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 (Pac-Man style) or non-wrapping. The specific implementation used in this report is somewhat monotonous, and may be found in Appendix A.

## 2.1 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, Show)
```

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 (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.)

## 2.2 Items

Botworld squares contain *items* which may be manipulated by the robots. Items include *robot parts* which can be used to construct robots, *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*. The value of different items varies from game to game; see Section 2.5 for details.

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

```
data Item
  = Cargo {cargoType :: Int, cargoWeight :: Int}
  | ProcessorPart Processor
  | RegisterPart Register
  | FramePart Frame
  | InspectShield
  | DestroyShield
  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 (FramePart _) = 100
weight _ = 1
```

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
```

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.[2]

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

[2]The following code introduces the function *forceR* :: *Constree* $\rightarrow$ *Register* $\rightarrow$ *Register*, which sets the contents of a register. It is defined in Appendix B.

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

## 2.3 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.

Robots specify the items they want to manipulate or the robots they want to target by giving the index of the target in the appropriate list. The *Int*s in *Lift* and *Build* commands index into the square's item list. The *Int*s in *Inspect* and *Destroy* commands index into the square's robot list. The *Int*s in *Drop* commands index into the inventory of the robot which gave the command.

```
data Command
  = Move Direction
  | Lift {itemIndex :: Int}
  | Drop {inventoryIndex :: Int}
  | Inspect {targetIndex :: Int}
  | Destroy {victimIndex :: Int}
  | Build {itemIndexList :: [Int], initialState :: 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
  | CannotLift Int
  | GrappledOver Int
  | Lifted Int
  | Dropped Int
  | InspectTargetFled Int
  | InspectBlocked Int
  | Inspected Int Robot
  | DestroyTargetFled Int
  | DestroyBlocked Int
  | Destroyed Int
  | BuildInterrupted [Int]
  | Built [Int] Robot
  | Invalid
  deriving (Eq, Show)
```

## 2.4 The step function

Botworld cells are updated in two alternating phases. First, in the *environment phase*, robot commands are read from each robot's register machine's output register and these are used to affect the world. This generates an *Event*, which describes the action that each robot performed and the way in which each item was manipulated.

```
data Event = Event
  {robotActions :: [(Robot, Action)]
  , untouchedItems :: [Item]
  , droppedItems :: [Item]
  , fallenItems :: [ItemCache]
  } deriving Show
```

This data structure makes it easy for programs (which get to see the *Event*) to differentiate beteween items that were untouched, items that were willingly dropped, and items which fell from a destroyed robot. In the last category, fallen robot parts are differentiated from fallen robot possessions.

```
data ItemCache = ItemCache
  {components :: [Item]
  , possessions :: [Item]
  } deriving Show
```

When observing Botworld games, it is sometimes useful to hop directly from *Event* to *Event*. For this, we define a convenience type.

```
type EventGrid = Grid (Maybe Event)
```

After the environment phase there is a *computation phase*, during which all remaining robots have their register machine's input register set (according to the *Event*) and then run (according to the host robot's processor). Each register machine is expected to leave a command in the output register at the end of the computation phase, for use in the next environment phase.

A single Botworld step thus consists of one environment phase followed by one computation phase:

$$\begin{aligned}&\mathit{step} :: (\mathit{Square}, \mathit{Map\ Direction\ Cell}) \rightarrow \mathit{Square}\\&\mathit{step} = \mathit{computationPhase} \circ \mathit{environmentPhase}\end{aligned}$$

We will now define the environment phase and the computaition phase in turn.

The environment phase begins by determining 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.[3]

### 2.4.1 Environment Phase

$$\begin{aligned}&\mathit{environmentPhase} :: (\mathit{Square}, \mathit{Map\ Direction\ Cell}) \rightarrow \mathit{Event}\\&\mathit{environmentPhase}\ (\mathit{sq}, \mathit{neighbors}) = \mathit{event}\ \textbf{where}\\&\quad (\mathit{robots}, \mathit{intents}) = \mathit{unzip}\ (\mathit{takeOutput} <\$> \mathit{robotsIn\ sq})\end{aligned}$$

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.

#### Resolving conflicts

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 compute whether each individual item is contested, and store the result in a list of items which corresponds by index to the cell's item list.

$$\begin{aligned}&\mathit{contested} :: [\mathit{Bool}]\\&\mathit{contested} = \mathit{isContested} <\$> [0 \mathinner{..} \mathit{pred}\ \$\ \mathit{length}\ \$\ \mathit{itemsIn\ sq}]\ \textbf{where}\end{aligned}$$

[3]The following code introduces the function $\mathit{takeOutput} :: \mathit{Decodable}\ o \Rightarrow \mathit{Robot} \rightarrow (\mathit{Robot}, \mathit{Maybe}\ o)$, defined in Appendix B.1, which reads a robot's output register, decodes the contents into a Haskell object, and clears the register.

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:[4]

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

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:

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

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

```
uses = allLifts ++ concat allBuilds
isContested i = i ∈ delete i uses
```

**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 destroy or inspect attempts that each robot receives in this step:

```
destroyAttempts :: [Int]
destroyAttempts = numAttempts <$>[0 . . pred $ length $ robotsIn sq] where
  numAttempts i = length [n | Just (Destroy n) ← intents, n ≡ i]
inspectAttempts :: [Int]
inspectAttempts = numAttempts <$>[0 . . pred $ length $ robotsIn sq] where
  numAttempts i = length [n | Just (Inspect n) ← intents, n ≡ i]
```

We then generate a list corresponding by index to the robot list which for each robot determines whether that robot is adequately shielded (againts various attacks) in this step[5]:

```
inspectShielded :: [Bool]
inspectShielded = zipWith isShielded [0 . .] robots where
  isShielded i r = (inspectAttempts !! i) ⩽ numInspectShields r
```

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

[5]This function introduces the helper functions *isInspectShield*, *isDestroyShield* :: *Item* → *Bool* defined in Appendix C.

```
  numInspectShields = length ∘ filter isInspectShield ∘ inventory
destroyShielded :: [Bool]
destroyShielded = zipWith isShielded [0 ..] robots where
  isShielded i r = (destroyAttempts !! i) ⩽ numDestroyShields r
  numDestroyShields = length ∘ filter isDestroyShield ∘ inventory
```

**Any robot that exits the square in this step cannot be attacked in this step.** Moving robots evade their pursuers, and the shields of moving robots are not destroyed. We define a function that determines whether a robot has successfully fled. This function makes use of the fact that movement commands into non-wall cells always succeed.

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

**Determining actions**

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.

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

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

```
  act :: Command → Action
  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 robot lacks the strength to hold the item, the lift fails.
3. If the item is contested, then multiple robots have attempted to use the same item.

Otherwise, the lift succeeds.

```
  act (Lift i) = maybe Invalid tryLift $ itemsIn sq !!? i where
    tryLift item
      | ¬ $ canLift robot item = CannotLift i
      | contested !! i = GrappledOver i
      | otherwise = Lifted i
```

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

```
act (Drop i) = maybe Invalid (const $ Dropped i) (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
    | inspectShielded !! 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
    | destroyShielded !! i = DestroyBlocked i
    | otherwise = Destroyed i
```

Build commands must also pass three checks in order to succeed:[6]

1. All of the specified indices must specify actual items.
2. None of the specified items may be contested.
3. The items must together specify a robot.

```
act (Build is m) = maybe Invalid tryBuild $ mapM (itemsIn sq!!?) is where
  tryBuild = maybe Invalid checkBuild ∘ construct
  checkBuild blueprint
    | any (contested!!) is = BuildInterrupted is
    | otherwise = Built is $ setState m blueprint
```

Pass commands always succeed.

[6]The following code introduces the function *setState* :: *Memory* → *Robot* → *Robot*, defined in Appendix B.1.

```
    act Pass = Passed
```

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

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

### Generating the event

With the local actions in hand, we can start updating the robots and items. We begin by computing which items were unaffected and which items were willingly dropped.[7]

```
untouched :: [Item]
untouched = removeIndices (lifts ++ builds) (itemsIn sq) where
  lifts = [i | Lifted i ← localActions]
  builds = concat [is | Built is _ ← localActions]
dropped :: [Item]
dropped = [item r i | (r, Dropped i) ← zip robots localActions] where
  item r i = inventory r !! i
```

We cannot yet compute the new item state entirely, as doing so requires knowledge of which robots were destroyed. The items and parts of destroyed robots will fall into the square, but only *after* the destroyed robot carries out their action.

We now turn to robots that began in the square, and update their inventories. (Note that because the inventories of moving robots cannot change, we do not need to update the inventories of robots entering the 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.)[8]

```
updateInventory :: Int → Action → Robot → Robot
updateInventory i a r = let stale = inventory r in case a of
  MovedOut _ → r
  Lifted n → r {inventory = (itemsIn sq !! n) : defend stale}
  Dropped n → r {inventory = defend $ removeIndices [n] stale}
  _ → r {inventory = defend stale}
  where
    defend = breakDestroyShields ∘ breakInspectShields
    breakDestroyShields = dropN (destroyAttempts !! i) isDestroyShield
    breakInspectShields = dropN (inspectAttempts !! i) isInspectShield
```

[7]The following code introduces the helper function *removeIndices* :: [*Int*] → [*a*] → [*a*] which is defined in Appendix C.

[8]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.

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
```

Now that we know the updated states of the robots, we can compute what items fall from the destroyed robots.

```
fallen = [cache r | (i, r) ← zip [0 ..] veterans, died i] where
  cache r = ItemCache (shatter r) (inventory r)
  died n = n ∈ [i | Destroyed i ← localActions]
```

Computing the updated robot states is somewhat more difficult. Before we can, we must 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]
incomingFrom dir neighbor = mapMaybe movingThisWay cmds where
  cmds = maybe [] (fmap takeOutput ∘ robotsIn) neighbor
  movingThisWay (robot, Just (Move dir′))
    | dir ≡ opposite dir′ = Just robot
  movingThisWay _ = Nothing
```

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

```
immigrations = assocs $ mapWithKey incomingFrom neighbors
(travelers, origins) = unzip [(r, d) | (d, rs) ← immigrations, r ← rs]
```

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

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

This allows us to compute a list of all robots that either started in the square, entered the square, or were created in the square in this step. Note that this list also contains robots that exited the square and robots that have been destroyed. This is intentional: the list of all robots (and what happened to them) is sent to each remaining robot as program input.

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

We nest generate the corresponding list of actions.

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

We have now computed the updated robots (and the corresponding actions) and the updated items (in three groups: untouched items, dropped items, and fallen items). This is all of the data that we need to complete the environment phase of the step function:

```
event = Event (zip allRobots allActions) untouched dropped fallen
```

### 2.4.2 Computaiton phase

We now proceed to the computation phase of the step function. This function turns an *Event* into an updated *Square*, by generating a new robot list and a new item list. The new robot list is generated by removing robots that exited or were destroyed, and running the register machines on the remaining robots. The new item list is generated by simply flattening the untouched, dropped, and fallen item lists into a single list.

```
computationPhase :: Event → Square
computationPhase = Square <$> newRobotList <*> newItemList where
  newRobotList :: Event → [Robot]
```

The new robot list is generated by running the register machines on each remaining robot after updating that robot's register machine's input register.[9]

```
newRobotList event = runMachine <$> prepped where
  prepped = [setInput r (createInput i a) | (i, r, a) ← triples]
  triples = [(i, r, a) | (i, r, a) ← zip3 [0 ..] robots actions, isHere i a]
  isHere i a = ¬ (isExit a ∨ i ∈ [x | Destroyed x ← actions])
  (robots, actions) = unzip $ robotActions event
```

Before being run, each robot receives three inputs:

1. The host robot's index in the robot/action list.
2. The *Event* object.
3. Some private input.

This data is encoded into the constree language, and the encoding is lossy: the contents of each robot's register machine are not included in the robot list, and robots cannot distinguish between *Passed* and *Invalid* actions taken by other robots. Also, the results of an *Inspect* command are only visible to the

[9]The following code introduces the function *setInput* :: *Encodable* $i \Rightarrow$ *Robot* $\to i \to$ *Robot*, defined in Appendix B.1, which encodes a Haskell object into Constree and sets the robot's input register accordingly.

inspecting robot. This data-hiding is implemented by the constree encoding code; see Appendix B.2 for details.

The following function creates the input object for each robot (if that robot remains in the square and survived):

```
    createInput :: Int → Action → (Int, Event, Constree)
    createInput n a = (n, event, private a)
```

A robot's private input either contains the results of a successful *Inspect* command or lets a robot know when its previous command was *Invalid*. Otherwise, the private input is empty.

```
  private :: Action → Constree
  private (Inspected _ r) = encode (processor r, length $ memory r, memory r)
  private Invalid = encode True
  private _ = Nil
```

Robot register machines are run using the *runFor* :: *Int* → *Memory* → *Either Error Memory* Constree function defined in Appendix B. Notice that a robot with invalid code has all of its registers cleared.[10]

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

Finally, we compute the new item list by simply discarding the additional item structure that was kept around for the purposes of robot input.

```
  newItemList :: Event → [Item]
  newItemList event = untouched ++ dropped ++ fallen where
    untouched = untouchedItems event
    dropped = droppedItems event
    fallen = concat [xs ++ ys | ItemCache xs ys ← fallenItems event]
```

This completes the computation phase.

### 2.4.3 Summary

This fully specifies the step function for Botworld cells. To summarize:

#### Environment phase

1. Robot machine output registers are read to determine robot intents.
2. Robot actions are computed from robot intents.

[10]The following code introduces the function *forceR* :: *Constree* → *Register* → *Register* which sets the contents of a constree register, defined in APpendix B.

3. Lifted and dropped items are computed.
4. Robot inventories are updated.
5. Fallen items are computed.
6. Incoming robots are computed.
7. Constructed robots are added.

**Computation phase**

1. Destroyed and exited robots are removed.
2. Register machine input registers are set.
3. Register machines are executed.
4. The item list is flattened.

As noted previously, machine programs are expected to leave a command in the output register for use in the next step.

## 2.5 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. (We imagine that the robots are airlifted out and their items are extracted for delivery to the player.) Each player may have their own assignment of values to items.

```
data Player = Player
  {values :: Item → Int
  ,home :: Position
  }
```

Because the values of items can vary by player, we need to know the player under consideration in order to compute the total value of a robot's inventory.

```
points :: Player → Robot → Int
points player r = sum (values player <$> inventory r)
```

A player's score at the end of a Botworld game is the sum of the values of all items held by all robots in that player's home square at the end of the game.

```
score :: Botworld → Player → Int
score world player = sum (points player <$> robots) where
  robots = maybe [] robotsIn $ at world $ home player
```

Most players use a very simple value function which assigns value only to cargo items in direct correspondence with the cargo type. For convenience, that value function is defined below.

```
standardValuer :: Item → Int
standardValuer (Cargo t _) = t
standardValuer _ = 0
```

Because Botworld steps begin with an environment phase, robots must be pre-loaded with a command to be executed in the initial step. Some games find this inconvenient, and prefer to begin with a robot phase instead of an environment phase. Such games may begin with a *creation phase* instead of an environment phase. The creation phase generates an event in which all robots are marked *Created* and take no actions. Such games may begin with a creation phase followed by alternating computation and environment phases.

```
creationPhase :: Square → Event
creationPhase (Square rs is) = Event (zip rs $ repeat Created) is [] []
```

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

# Chapter 3

# 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 to us 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 publications will illustrate some of the work that we've done based on Botworld.

# Appendix 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 (Pac-Man 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
instance Functor Grid where
  fmap f g = g { cells = fmap f $ cells g }
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
```

The following series of functions are useful for creating grids. The first creates a grid from a generator function:

```
generate :: Dimensions → (Position → a) → Grid a
generate dim gen = let g = Grid dim (gen <$> indices g) in g
```

The next generates a grid from a list, padded with *Nothing*s as neccessary:

```
fillGrid :: Dimensions → [a] → Grid (Maybe a)
fillGrid dim xs = generate dim (λpos → xs !!? locate dim pos)
```

The final three are useful for creating a grid from a list where only one dimension is known: for example, creating a grid of width 3 from a list, using as few rows as possible, padded as necessary.

```
cutGrid :: (Int → Dimensions) → [a] → Grid (Maybe a)
cutGrid cut xs = generate dim get where
  get pos = xs !!? locate dim pos
  dim = cut $ length xs
vGrid :: Int → [a] → Grid (Maybe a)
vGrid maxw = cutGrid (λlen → (min maxw len, (len + pred maxw) `div` maxw))
hGrid :: Int → [a] → Grid (Maybe a)
hGrid maxh = cutGrid (λlen → ((len + pred maxh) `div` maxh, min maxh len))
```

## 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

Next, we define functions that update an entire Botworld grid. The first two functions run a single phase of the two-phase step function on an entire grid:

```
runCreation :: Botworld → EventGrid
runCreation = fmap (fmap creationPhase)
runEnvironment :: Botworld → EventGrid
runEnvironment bw = bw {cells = doEnv <$> indices bw} where
```

```
    doEnv pos = environmentPhase ∘ withNeighbors pos <$> at bw pos
    withNeighbors pos sq = (sq, fromList $ walk pos <$> [N ..])
    walk pos dir = (dir, at bw $ towards dir pos)
  runRobots :: EventGrid → Botworld
  runRobots = fmap (fmap computationPhase)
```

The next updates an entire Botworld grid by one step:

```
  update :: Botworld → Botworld
  update = runRobots ∘ runEnvironment
```

A final convenience function updates from one event directly to the next.

```
  update′ :: EventGrid → EventGrid
  update′ = runEnvironment ∘ runRobots
```

# Appendix 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.

```
setState :: Memory → Robot → Robot
setState m robot = robot {memory = fitted} where
  fitted = zipWith (forceR ∘ contents) m (memory robot) ++ padding
  padding = 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 each tick.

```
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 *Inspect* 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 f ∘ decode where f (a, (b, c)) = (a, b, c)
instance (Encodable a, Encodable b, Encodable c, Encodable d) ⇒
  Encodable (a, b, c, d) where
  encode (a, b, c, d) = encode (a, (b, c, d))
instance (Decodable a, Decodable b, Decodable c, Decodable d) ⇒
  Decodable (a, b, c, d) where
  decode = fmap f ∘ decode where f (a, (b, c, d)) = (a, b, c, d)
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 (Decodable a, Decodable b, Decodable c, Decodable d, Decodable e) ⇒
  Decodable (a, b, c, d, e) where
  decode = fmap f ∘ decode where f (a, (b, c, d, e)) = (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
```

The special token *Cons Nil* (*Cons Nil Nil*) (which cannot appear as an item in an encoded list of *Bool*s) is allowed to appear at the beginning of an encoded *Int*, in which case it denotes a negative sign.

```
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) = negate <$> decode n
  decode t = decode t ≫= unbits where
    unbits [ ] = Just 0
    unbits [False] = Nothing
    unbits (x : xs) = (λy → (if x then 1 else 0) + 2 ∗ y) <$> 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 Decodable Color where
  decode t = ([Red ..]!!?) =≪ decode t
instance Encodable Frame where
  encode (F c s) = encode (c, s)
instance Decodable Frame where
  decode = fmap (uncurry F) ∘ decode
instance Encodable Processor where
  encode (P s) = encode s
instance Decodable Processor where
  decode = fmap P ∘ decode
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 DestroyShield       = encode (4 :: Int, Nil)
    encode InspectShield       = encode (5 :: Int, Nil)
  instance Decodable Item where
    decode t = case decode t :: Maybe (Int, Constree) of
      Just (0, args) → uncurry Cargo <$> decode args
      Just (1, args) → RegisterPart <$> decode args
      Just (2, args) → ProcessorPart <$> decode args
      Just (3, args) → FramePart <$> decode args
      Just (4, Nil)  → Just DestroyShield
      Just (5, Nil)  → Just InspectShield
      _              → Nothing
  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 constree. 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 (2 :: Int, direction d)
    MovedOut d           → encode (3 :: Int, direction d)
    MovedIn d            → encode (4 :: Int, direction d)
    CannotLift i         → encode (5 :: Int, i)
    GrappledOver i       → encode (6 :: Int, i)
    Lifted i             → encode (7 :: Int, i)
    Dropped i            → encode (8 :: Int, i)
    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 ..]
instance Encodable ItemCache where
  encode (ItemCache pt ps) = encode (pt, ps)
instance Decodable ItemCache where
  decode = fmap (uncurry ItemCache) ∘ decode
instance Encodable Event where
  encode (Event ras u d f) = encode (rs, as, (u, d, f)) where (rs, as) = unzip ras
```

# Appendix C

# Helper Functions

This section contains simple helper functions used to implement the Botworld step function. The first few are used to distinguish different types of items and actions:

$$\begin{array}{l}
\mathit{isPart} :: \mathit{Item} \rightarrow \mathit{Bool} \\
\mathit{isPart}\ (\mathit{RegisterPart}\ \_) = \mathit{True} \\
\mathit{isPart}\ \mathit{item} = \mathit{isProcessor}\ \mathit{item} \vee \mathit{isFrame}\ \mathit{item}
\end{array}$$

$$\begin{array}{l}
\mathit{isProcessor} :: \mathit{Item} \rightarrow \mathit{Bool} \\
\mathit{isProcessor}\ (\mathit{ProcessorPart}\ \_) = \mathit{True} \\
\mathit{isProcessor}\ \_ = \mathit{False}
\end{array}$$

$$\begin{array}{l}
\mathit{isFrame} :: \mathit{Item} \rightarrow \mathit{Bool} \\
\mathit{isFrame}\ (\mathit{FramePart}\ \_) = \mathit{True} \\
\mathit{isFrame}\ \_ = \mathit{False}
\end{array}$$

$$\begin{array}{l}
\mathit{isInspectShield} :: \mathit{Item} \rightarrow \mathit{Bool} \\
\mathit{isInspectShield}\ \mathit{InspectShield} = \mathit{True} \\
\mathit{isInspectShield}\ \_ = \mathit{False}
\end{array}$$

$$\begin{array}{l}
\mathit{isDestroyShield} :: \mathit{Item} \rightarrow \mathit{Bool} \\
\mathit{isDestroyShield}\ \mathit{DestroyShield} = \mathit{True} \\
\mathit{isDestroyShield}\ \_ = \mathit{False}
\end{array}$$

$$\begin{array}{l}
\mathit{isExit} :: \mathit{Action} \rightarrow \mathit{Bool} \\
\mathit{isExit}\ (\mathit{MovedOut}\ \_) = \mathit{True} \\
\mathit{isExit}\ \_ = \mathit{False}
\end{array}$$

The rest 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):

$$\begin{array}{l}
\mathit{singleton} :: [\,a\,] \rightarrow \mathit{Maybe}\ a \\
\mathit{singleton}\ [\,x\,] = \mathit{Just}\ x \\
\mathit{singleton}\ \_ = \mathit{Nothing}
\end{array}$$

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

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

One to safely alter a specific item in a list:

```haskell
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:

```haskell
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.

```haskell
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 _ _ [] = []
```

# Bibliography

[1] Alex Altair. A comparison of decision algorithms on newcomblike problems. 2013.

[2] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, (223):120–123, October 1970.

[3] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *In IEEE Symposium on Security and Privacy*, pages 154–165, 2003.

[4] Marcus Hutter. *Universal Artificial Intelligence: Sequential Decisions based on Algorithmic Probability.* Springer, Berlin, 2005.

[5] Laurent Orseau and Mark B. Ring. Space-time embedded intelligence. In Joscha Bach, Ben Goertzel, and Matthew Iklé, editors, *AGI*, volume 7716 of *Lecture Notes in Computer Science*, pages 209–218. Springer, 2012.

[6] Vladimir Slepnev. A model of UDT with a halting oracle. `http://lesswrong.com/lw/8wc/a_model_of_udt_with_a_halting_oracle/`, 2011.

[7] Vladimir Slepnev. A model of UDT without proof limits. `http://lesswrong.com/lw/b0e/a_model_of_udt_without_proof_limits/`, 2012.

[8] Wikipedia. Markov decision process — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Markov_decision_process&oldid=589500076`, 2014. [Online; accessed 10-April-2014].

[9] Eliezer Yudkowsky and Marcello Herreshoff. Tiling agents for self-modifying AI, and the Löbian obstacle. 2013.