Practical Verification of Concurrent Haskell Programs

Michelle Schifferstein
Supervisor(s): Jesper Cockx, Lucas Escot
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

The formal verification of concurrent programs is of particular importance, because concurrent programs are notoriously difficult to test. Because Haskell is a purely functional language, it is relatively easy to reason about the correctness of such programs and write down manual proofs. However, since these methods are still prone to error, this paper investigates how AGDA2HS can be used to automate the verification process in Agda, while keeping the advantages of having our code available in Haskell. This paper shows how AGDA2HS enables the partial verification of a simple Haskell concurrency model. The model is first ported to Agda, staying as close to the original code as possible, and directly compared to the Haskell translation provided by AGDA2HS to showcase the readability of the code it produces. Consequently, it is shown how Agda's proof techniques can be used to provide straightforwards proofs of the presence or absence of deadlocks in simple concurrent programs written in this model. Finally, the model's limitations, in particular its deterministic nature, are discussed.

1 Introduction

As a strongly typed, purely functional language, reasoning about the correctness of Haskell programs is relatively easy. However, we would also like to have a slightly more formal way of *showing* other programmers that the library their code relies upon is free from bugs. While libraries such as Quickcheck[1] allow for thorough and efficient testing, testing techniques are not quite as rigorous as formal verification methods, which aim to systematically prove the correctness of a program [1]. In contrast to Haskell, Agda[2], as a dependently typed language, enables us to write such formal proofs about our programs. However, this language is still relatively experimental, mostly used for research and lacking the large infrastructure of an established programming language like Haskell.

AGDA2HS[3] is a tool that enables Haskell library developers to formally verify their programs. It allows them to write their code initially in Agda, consequently provide formal correctness proofs for them, and finally have AGDA2HS translate them to workable Haskell code. The aim of AGDA2HS is to combine the benefits of both worlds, because the programs can be formally verified in Agda, while deployment and further development, for example by those unfamiliar with Agda, can happen in the more popular Haskell. This makes the verified code available to anyone wishing to use it in their Haskell programs.

At this stage, AGDA2HS is still an experimental tool, and identifying the common language subset of Agda and Haskell is being actively researched. In a previous B.Sc. project at the TU Delft, five students have formally verified several Haskell data structures in order to investigate the potential of AGDA2HS as a tool [2]. All of them were successful in porting substantial sections of the chosen Haskell libraries to Agda, proving a number of formal properties, and generating readable and equivalent Haskell code with AGDA2HS.

In this paper, we extend on the previous work by showing how Haskell concurrency models can be verified with the use of AGDA2HS. The reason for this is that we expect that about concurrent programs in particular, interesting properties may be proven in Agda. Classic properties of such programs include liveness, the absence of deadlocks or program-specific invariants [3]. The formal verification of such properties is of vital importance to ensure the correctness of a concurrent program. If concurrency models can be ported

[1]`https://hackage.haskell.org/package/QuickCheck`
[2]`https://wiki.portal.chalmers.se/agda`
[3]`https://github.com/agda/agda2hs`

to Agda and correctly converted to Haskell with AGDA2HS, we would obtain a potentially valuable and convenient method for formally verifying concurrent Haskell programs.

Formally verifying concurrent programs is nothing new - because testing concurrent programs is very difficult, one generally has to resort to formal verification methods, such as proving properties using the Spin model checker [4], which has the serious disadvantage that one has to rewrite the entire program in a second language before being able to prove anything, and this manual translation is itself again susceptible to bugs (whereas our goal is to write and verify the programs in Agda to begin with, and translate them automatically with AGDA2HS). Work has also been done on writing such proofs in Agda for concurrent programs generally [4]-[5] and for Haskell's `STM` library[5] in particular [6], so if we would be able to automatically translate concurrent Agda programs to valid and readable Haskell using AGDA2HS, this would certainly be a useful addition.

This paper therefore sets out to (1) explain how a Haskell concurrency model can be ported to Agda, such that AGDA2HS produces equivalent and readable Haskell code upon translation; and (2) identify formal properties of the programs using the given model and verify these in Agda. In particular, because of time constraints, we will focus on the occurrence of deadlocks in such programs by proving their (non-)termination.

2 Background

We assume the reader to be generally familiar with Haskell as a programming language. Haskell's concurrency features will be explained, as well as the particular model used in this paper. For the sake of brevity, we will also consider basic familiarity with Agda, although an unfamiliar reader should be able to grasp the gist of the provided code examples with the explanations given in the paper.[6]Little prior knowledge about concurrent programs is necessary, since the relevant properties and problems will be highlighted here.

2.1 Concurrent Haskell

In a concurrent program, multiple threads work together simultaneously on performing one or more tasks. Each thread is essentially a sequential program, but the effects of the actions of the different threads will appear interleaved in time [7]. It becomes interesting when there are *side effects* involved, i.e. the change of some non-local state rather than just the pure computation of a result [8], since the occurrence of such an effect will influence the outcome of future actions. The execution order of the threads' actions then becomes relevant. A common example of this is when the different threads operate on some kind of mutable shared state, e.g. to communicate with each other.

In Haskell, a mutable state can be represented in the `IO` monad with an `IORef`[7], a mutable variable that can be read from and written to. To deal with concurrency, we can place such a variable in a *mutex* (mutually exclusive) box, that can only be accessed by one thread at a time: this is an `MVar`[8]. An `MVar` is a box that is either full, containing a variable, or empty. When full, its value can be read and removed (`takeMVar`) or simply read (`readMVar`); when empty it can be written to (`putMVar`). When either action is not

[4]`https://spinroot.com`

[5]`https://hackage.haskell.org/package/stm`

[6]See `https://wiki.portal.chalmers.se/agda/` for Agda's documentation as well as tutorial suggestions.

[7]`https://hackage.haskell.org/package/base-4.16.1.0/docs/Data-IORef.html`

[8]`https://hackage.haskell.org/package/base-4.16.1.0/docs/Control-Concurrent-MVar.html`

possible, a thread *blocks* on it when trying to perform the action, and waits until the action becomes available. We will show how `MVars` can be modeled in Agda.

An elaborate and accessible introduction to concurrent programming in Haskell can be found in [7]. It includes a chapter on Haskell's Software Transactional Memory (STM), an interesting and important concurrency model that is outside the scope of this paper.

2.2 Properties of Concurrent Programs: Deadlocks

Two kinds of properties characterize good concurrent programs: safety properties and liveness properties [3]. Safety properties ensure the program behaves correctly - this includes the absence of deadlocks, correct results, or program-specific invariants such as that x is never smaller than 0. For safety properties, we can always point out a specific point in the program where they fail, if they do. Liveness properties ensure a program makes progress, e.g. they might specify what happens after performing an action a or that a program eventually always returns to state s.

In this paper, we will focus on one of the safety properties: the absence of deadlocks. According to the Encyclopedia of Parallel Programming, "A deadlock is said to occur when two or more processes are waiting for each other to release a resource. None of the processes can make any progress" [9].

```
mVarDeadlock :: IO ()
mVarDeadlock = do
        a <- newEmptyMVar
        b <- newEmptyMVar
        forkIO $ do
                    takeMVar a
                    putMVar b ()
        takeMVar b
        putMVar a ()
```

```
mVarDeadlock :: IO ()
mVarDeadlock = do
        a <- newEmptyMVar
        b <- newEmptyMVar
        forkIO $ do
                    takeMVar a
                    putMVar b ()
        putMVar a ()
        takeMVar b
```

Listing 1: A simple concurrent Haskell program that deadlocks because the two threads each first try to read an empty `MVar` before writing to the other (left) and a non-deadlocking alternative (right).

For our verification, we will consider a simple example of a deadlocking program. The aim is to detect through proofs in Agda that this program contains deadlocks. The program in Listing 1 (left) illustrates a simple deadlock arising from the use of `MVars`. This program creates two empty `MVars`, and then a new thread is forked from the main thread that reads a and writes b, while the main thread continues to read b and write a. It meets the four necessary conditions for a deadlock, as explained in [9]: (1) we have two *mutex* resources, the `MVars` a and b; (2) both resources are waited upon by one of the threads, because `takeMVar` is called while the `MVars` are empty; (3) the resources can only be released (i.e. written to) by the other thread; (4) the threads are waiting for each other. The program thus deadlocks. If both threads were to act on the variables in the same order (Listing 1, right), condition 4 would be unsatisfied, because the two threads are never waiting at the same time - thus that program is free from deadlocks.

At present, GHC's run-time system (RTS) can automatically detect deadlocks through its garbage collector, so this feature is built into Haskell.[9] However, this detection is not flawless and may in some cases warn for deadlocks were there are none, as well as fail to detect some [7]. Thus, this is for debugging purposes only and it is worthhile to investigate whether we can obtain more certainty through Agda's proof techniques.

[9] `https://hackage.haskell.org/package/base-4.16.1.0/docs/Control-Concurrent.html`

3 A Simple Concurrency Model

Haskell's `Control.Concurrent` library[10] forms a collection of concurrency abstractions that rely on low-level primitives that are built into Haskell, which makes it difficult to translate the definitions of e.g. `MVar` or `STM` (another concurrency model) directly to Agda. Therefore, for a first attempt to model Haskell's concurrency in Agda, we will work with a particular functional specification, presented in [10], that does not make use of any additional primitives. An alternative, more elaborate such model is presented in [11]; we will not use it as our basis because it was unknown to us at the early stages of the present work, but we will adopt crucial parts of it along the way (see section 5).

The main idea of the model presented in [10] is to represent concurrency as a monad transformer, which can 'lift' the operations in any arbitrary monad to a concurrent setting. A monad transformer is defined as:[11]

```
class MonadTrans t where
    lift :: Monad m => m a -> (t m) a
```

Since we will not make use of actual threads, concurrent processes are interleaved - we run the first part of a computation and suspend the rest, allowing another process to run in the meantime. We do this through what is called *continuation passing style*[12], which allows for partial computations. We can then define a type `C` to represent concurrency:

```
type C m a = (a -> Action m) -> Action m
```

This type takes a monad `m` over `a`; additionally, it takes another argument, a function, that specifies what should be done with the result (of type `a`) of the current computation (`Action m`) - namely performing another action over `m`.

`Action` is defined as a data type that can be either an atomic operation, a 'fork' that creates new processes, or an indication to stop a process:

```
data Action m = Atom (m (Action m))
              | Fork (Action m) (Action m)
              | Stop
```

The model additionally defines the `action` function to convert an expression of type `C` into an action, and functions to turn any computation in `m` into an atomic action in `C m`[13], as well as to fork and to stop a process:

```
action :: Monad m => C m a -> Action m
action m = m (\_ -> Stop)

atom :: Monad m => m a -> C m a
atom m = \c -> Atom (fmap c m)

stop :: Monad m => C m a
stop = \_ -> Stop
```

[10]`https://hackage.haskell.org/package/base-4.16.1.0/docs/Control-Concurrent.html`

[11]As of GHC 8.8, a type class constraint was added to the class declaration to enforce that (t m) is a monad. Its documentation can be found at `https://hackage.haskell.org/package/transformers-0.6.0.4/docs/Control-Monad-Trans-Class.html`. For our purposes the simpler definition will do.

[12]Read more about continuation passing style in Haskell here: `https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style`.

```
fork :: Monad m => C m a -> C m ()
fork m = \c -> Fork (action m) (c ())
```

Originally, the model defined the `atom` function as `atom m = \c -> Atom (fmap c m)`, but since Functor is a superclass of Monad since GHC 7.10 [12], this definition can be significantly simplified with the use of `fmap`.

Finally, we can then make `C` a monad transformer instance, such that lifting a monadic computation to `C` will make it an atomic operation in the concurrent setting:

```
instance MonadTrans C where
    lift = atom
```

To schedule the computations performed by the processes, the model makes use of a simple *round robin* scheduler, which takes a list of actions, lets a process perform the first part of a computation, and places the continuation at the end of the list. When a process is forked, both of its actions are added to the end of the list. This procedure continues until the list is empty (possibly indefinitely).

This model allows us to represent simple concurrent programs; it will also enable us to construct the examples of faulty programs from the previous section, because we can use it to represent Haskell's `MVars` - we will cover the details of this as part of the implementation.

4 Implementation

Our first priority is to translate the model to Agda in such a way that AGDA2HS converts it to readable Haskell code as close as possible to the definitions provided in the previous section. Not all code can be presented here due to a lack of space, so only the most notable parts will be discussed in detail.[14]

Our code will make extensive use of the standard library of AGDA2HS, that provides definitions for useful types such as `List`, `Maybe` or `Monad`.[15] Using these rather than importing them from Agda directly ensures they are translated correctly to Haskell.

```
{-# NO_POSITIVITY_CHECK #-}
data Action (m : Set → Set) : Set where
    Atom : m (Action m) → Action m
    Fork : Action m → Action m → Action m
    Stop : Action m
{-# COMPILE AGDA2HS Action #-}
```

```
data Action m = Atom (m (Action m))
              | Fork (Action m) (Action m)
              | Stop
```

Listing 2: The `Action` data type in Agda (left) with the corresponding Haskell translation by AGDA2HS (right).

In order to define `C`, we will first need a data type `Action`. As shown in Listing 2, it is possible to define this in such a way that it translates exactly to the Haskell code we want. `m`, representing a monad (although not restricted to monads here) is of type `Set → Set` because a monad is parameterized over some `a : Set`.

However, note the pragma here that switches off the positivity checker. Agda requires all its (recursive) types to be strictly positive, because otherwise one can write non-terminating

[14] All code can be found at `https://github.com/mschifferstein/concurrent-haskell-verification/`.
[15] See `https://github.com/jespercockx/agda2hs/tree/master/lib/Haskell/Prim`.

functions with them - which, since Agda is a total language, is not allowed.[16] In our case, the `Atom` constructor is not strictly positive because `Action` occurs as an argument to `m`.

When we turn off the positivity checker, we basically work *under the assumption that* our data type is positive. That is, our proofs are valid given that this assumption holds - meaning this part needs to be manually verified. There are cleaner, alternative solutions thinkable which could however not be fully implemented within the scope of this project; the interested reader may find partial implementation and documentation of related problems in the repository.

With our `Action` type in place, we can move on to a definition of `C`. In [10], `C` is defined as a type class. However, Haskell does not allow us to define instances of `(C m)` where `C` is a type class, even with the `TypeSynonymInstances` flag, because `C` is supposed to take two arguments rather than one. In Haskell, most monad transformers (e.g. `MaybeT`[17]) are instead defined with `newtype` declarations, if not as data types, but these are not currently accepted by AGDA2HS. We thus use have to opt for the more cumbersome but equivalent alternative given in Listing 3, which AGDA2HS can translate without problems.

```
record C (m : Set → Set) (a : Set) : Set where
    constructor Conc
    field
        act : (a → Action m) → Action m
open C public
{-# COMPILE AGDA2HS C #-}
```

```
data C m a =
    Conc{act :: (a -> Action m) -> Action m}
```

Listing 3: A definition of `C` as a record type (left) with the corresponding Haskell translation by AGDA2HS (right).

While [10] only provides a monad instance for `C`, we need to provide functor and applicative instances as well, since these are superclasses of `Monad` as of GHC 7.10 [12]. Fortunately, `fmap` and `<*>` can be defined in terms of bind (>>=).[18] For the sake of brevity, we leave out these instance definitions as well as the straightforward definitions of the functions that accompany our model.

```
round_robin : {{Monad m}} → List (Action m) → MyNat → m ⊤
round_robin _ Zero = return tt
round_robin [] (Suc n) = return tt
round_robin (Atom x ∷ xs) (Suc n) = do
                            x1 ← x
                            round_robin (xs ++ (x1 ∷ [])) n
round_robin (Fork x y ∷ xs) (Suc n) = round_robin (xs ++ (x ∷ y ∷ [])) n
round_robin (Stop ∷ xs) (Suc n) = round_robin xs n
```

Listing 4: The `round_robin` function for interleaving actions. It takes a custom defined natural number as argument as 'fuel' to ensure termination.

The last function that is worth discussing is the `round_robin` function from Listing 4, which is responsible for interleaving the concurrent actions. Originally, this function is

[16] Read more about Agda's strict positivity requirements here: https://agda.readthedocs.io/en/latest/language/data-types.html#strict-positivity.

[17] https://hackage.haskell.org/package/transformers-0.6.0.4/docs/Control-Monad-Trans-Maybe.html

[18] See https://wiki.haskell.org/Monad.

essentially non-terminating, since new actions may always be added to the list. However, Agda requires all its functions to be terminating.[19]Thus we have it take a (custom defined, to be able to pattern match) natural number as argument, serving as 'fuel', ensuring that the function terminates once the fuel runs out (reaches zero). This will unfortunately also appear in the Haskell translation, but in practice we can initialize it with some very large value (the maximum integer value in Haskell would allow for almost 10^{16} recursive calls) that is practically equivalent to looping infinitely.

Now that we have a working concurrency model, we can use it to define `MVar`s. [10] defines `MVar`s and their operations in terms of Haskell's built-in `Var`, which is nowadays the `IO` monad's `IORef` (see subsection 2.1). Given our concurrency model, we will follow this format, although in practice `IORef` is defined in terms of `MVar` instead. The model assumes the `IO` monad as a primitive; for now we will therefore *postulate*[20]it in Agda, along with `IORef` and its operations, and have our Haskell code import the existing `IORef` - we are thereby able to obtain Haskell code using these primitives that compiles correctly and behaves as we would expect. This allows us to first finish our model by adding `MVar`s to it; in section 5 we will see how to actually model IO.

```
MVar : (a : Set) → Set
MVar a = IORef (Maybe a × Bool)
{-# COMPILE AGDA2HS MVar #-}

newEmptyMVar : { @0 a : Set } → C IO (MVar a)
newEmptyMVar = lift (newIORef (Nothing , True))
{-# COMPILE AGDA2HS newEmptyMVar #-}

newMVar : a → C IO (MVar a)
newMVar a = lift (newIORef (Just a , True))
{-# COMPILE AGDA2HS newMVar #-}
```

```
type MVar a = IORef (Maybe a, Bool)

newEmptyMVar :: C IO (MVar a)
newEmptyMVar = lift (newIORef (Nothing, True))

newMVar :: a -> C IO (MVar a)
newMVar a = lift (newIORef (Just a, True))
```

Listing 5: A definition of `MVar` in terms of `IORef` in Agda (left) with the corresponding Haskell translation by AGDA2HS (right). Operations for creating new `IORef`s are lifted in their entirety to atomic actions in the concurrent setting.

Listing 5 displays the definition of `MVar` with operations for creating a new `MVar`, which is basically an `IORef` (a regular, non-concurrent mutable variable) that contains a tuple of `Maybe a` (which contains a value if and only if the `MVar` is full) and a boolean (which we will need for the write operations). `MVar`s can be created either empty of full. The latter is not presented in [10], but we have added it to our model since it is part of Haskell's `MVar` definition. Function definitions have been adjusted since `newIORef` is defined to create an `IORef` with a given value, not an empty one. Creating new `MVar`s must be an atomic operation in the concurrent setting, thus is `lift`ed in its entirety.

We have also defined read and write operations for `MVar`s. The latter and most interesting one is given in Listing 6. While [10] chooses a convenient definition of `MVar` that simply always writes the given value to the `MVar`, we opted for porting the actual semantics, according to which the operation should block if the `MVar` is full.[22]To this aim, we should (1) check that no other thread is currently busy writing to the `MVar` (i.e. the boolean tuple value should be `True`); and (2) check that the `MVar` is currently empty. If both conditions

[19]Read more about Agda's termination checker here: `https://agda.readthedocs.io/en/latest/language/termination-checking.html`.

[20]`https://agda.readthedocs.io/en/v2.6.2.2/language/postulates.html`

[21]This corresponds to `Control.Concurrent`'s `putMVar`.

```agda
checkWriteOk : MVar a → IO (Maybe a × Bool)
checkWriteOk v = do
    v1 ← readIORef v
    writeIORef v (fst v1 , False)
    return v1
{-# COMPILE AGDA2HS checkWriteOk #-}

endWrite : MVar a → a → IO ⊤
endWrite v a = writeIORef v (Just a , True)
{-# COMPILE AGDA2HS endWrite #-}

writeMVar : MVar a → a → MyNat → C IO (Maybe ⊤)
writeMVar v a Zero = return Nothing
writeMVar v a (Suc fuel) = do
    v1 ← lift (checkWriteOk v)
    case v1 of λ
        {(Nothing , True) → (do
                    x <- lift (endWrite v a)
                    return (Just x));
        (Just b , True) →  lift (endWrite v b)
                        >> writeMVar v a fuel;
        (_ , False) → writeMVar v a fuel}
{-# COMPILE AGDA2HS writeMVar #-}
```

```haskell
checkWriteOk :: MVar a -> IO (Maybe a, Bool)
checkWriteOk v
  = readIORef v >>=
      \ v1 -> writeIORef v (fst v1, False)
              >> return v1

endWrite :: MVar a -> a -> IO ()
endWrite v a = writeIORef v (Just a, True)

writeMVar :: MVar a -> a -> MyNat -> C IOs (Maybe ())
writeMVar v a Zero = return Nothing
writeMVar v a (Suc fuel)
  = lift (checkWriteOk v) >>=
      \ v1 -> case v1 of
        (Nothing, True) -> lift (endWrite v a)
                                  >>= \ x -> return (Just x)
        (Just b, True) -> lift (endWrite v b)
                                  >> writeMVar v a fuel
        (_, False) -> writeMVar v a fuel
```

Listing 6: The definition of `writeMVar`[21]that blocks through busy waiting when the `MVar` is full, in Agda (left) with the corresponding Haskell translation by AGDA2HS (right).

are met, we can write the new value to it, otherwise the operation has to block. In this case, blocking is accomplished by having the function call itself recursively, which is known as 'busy waiting' - a method that is very inefficient but is also the simplest implementation to serve our purposes.[23]

The busy waiting makes the function essentially non-terminating; as with the `round_robin` function, we solve this by adding 'fuel' and have the return type be `C IO (Maybe ⊤)` rather than `C IO ⊤`, so that we can return `Nothing` when running out of fuel. Ideally we would want to use a wrapper type to hide these implementation details for the user, so they will not have to pattern match on the result whenever reading from or writing to an `MVar`.

We need to go through the implementation in a bit more detail to convince ourselves of its correctness. When `writeMVar` is called, it first checks that no other thread is currently writing the `MVar`. It performs this check *and* sets the boolean value to `False` in one atomic operation, so that no two threads can do this at the same time. If the check fails, the function recurses. If it succeeds and the `MVar` is empty, it writes the new value *and* sets the boolean value to `True` in one atomic operation. If the `MVar` is full, it first restores the boolean value to indicate no thread is currently writing, after which it recurses.

Finally, `MVar`s can be used to create unbounded channels, as in Haskell's `Chan`[24] data type [7], which is also possible in our model - although again we need to work around issues with Agda's positivity checker. An implementation may be found in the repository, but is not discussed here because verification of it is not yet possible, due to time constraints.

[23]See Chapter 3 of [13].

[23]According to the original definition from [14], `writeMVar` would instead throw an error when the `MVar` is already full. These are the semantics referred to by [10].

[24]`https://hackage.haskell.org/package/base-4.16.1.0/docs/Control-Concurrent-Chan.html`

5 Verification: Proving (the Absence of) Deadlocks

Now that we have an implementation of `MVars` in Agda that AGDA2HS can translate to correct Haskell code, we can return to the example program in subsection 2.2 and verify its correctness. In particular, we will be addressing the problem of deadlock detection: can we show that a concurrent program is free from deadlocks, or, in the case of the example, that it is indeed faulty and deadlocks?

Before we can proceed with writing proofs, there is one more thing we need. At present, we postulate `IO`, `IORef` and its operations in our Agda code, and have the translated Haskell code use Haskell's own built-in counterparts. However, Agda knows nothing about how postulates behave and therefore cannot normalize expressions using them. So instead, we will need to model their behavior: we need a functional specification that allows us to define the impure `IO` operations we need in a pure way. [11] provides the details of such a model that allows us to work with mutable state. Here we will make minimal adjustments to obtain this model in Agda.

```
Loc = Nat
{-# COMPILE AGDA2HS Loc #-}

record IORef (a : Set) : Set where
    constructor MkIORef
    field
        mem : Loc
open IORef public
{-# COMPILE AGDA2HS IORef #-}

Data = Maybe Nat × Bool
{-# COMPILE AGDA2HS Data #-}

data IOs (a : Set) : Set where
    NewIORef : Data → (Loc → IOs a) → IOs a
    ReadIORef : Loc → (Data → IOs a) → IOs a
    WriteIORef : Loc → Data → (IOs a) → IOs a
    Return : a → IOs a
{-# COMPILE AGDA2HS IOs #-}
```

```
type Loc = Natural

data IORef a = MkIORef{mem :: Loc}

type Data = (Maybe Natural, Bool)

data IOs a = NewIORef Data (Loc -> IOs a)
           | ReadIORef Loc (Data -> IOs a)
           | WriteIORef Loc Data (IOs a)
           | Return a
```

Listing 7: A definition of the `IOs` data type with constructors for all `IORef` operations in Agda (left) with the corresponding Haskell translation by AGDA2HS (right).

Listing 7 presents the main data type of our `IO` model, `IOs`. It has constructors for all `IORef` operations we earlier postulated, as well as a `Return` constructor to wrap an arbitrary value inside `IOs`. Each `IORef` includes a pointer to a (fictive) unique memory location (through the `Loc` type) at which its contents can be found. In our model, `IORefs` will have content of type `Data`, which is simply a type synonym for `Maybe Nat × Bool`, so that we can define `MVar = IORef Data`. We will limit ourselves to natural numbers as content; [11] suggests how a generalized version may be implemented.

There are functions for `newIORef`, `readIORef` and `writeIORef` that instantiate the respective constructors. Additionally `Functor`, `Applicative` and `Monad` instances are defined for `IOs`. Definitions can be found in [11], and as their conversion to Agda is straightforward the code is not presented here.

Finally, we have a `Store` that keeps track of the next free memory location as well as the data corresponding to each location. A `runIOs` function evaluates the stateful computation of a program in `IOs`, initialized with an empty store - this computation is given by the function `runIOState` and has cases for each of the `IOs` constructor cases - adding new data to the store for `NewIORef`, retrieving data from it for `ReadIORef`, updating its data for `WriteIORef` and ending the computation for `Return`. Again our Agda implementation, left out here for the sake of brevity, corresponds closely to the definitions given in [11].

```
prop : runIOs (do
                ref <- newIORef (Just 5 , True)
                empt <- newIORef (Nothing , True)
                readIORef ref)
        ≡
        (Just 5 , True)
prop = refl
```

Listing 8: A simple proof for an `IORef` program that the result of the final *read* operation evaluates to what was originally put in.

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Listing 9: Agda 's equality data type with the reflexivity constructor that indicates that every value is equal to itself.[25]

We are now able to write proofs concerning the results of programs that operate on `IORef`s. For example, the proof in Listing 8 verifies that a program which creates two `IORef`s and reads the contents from the first one returns the value we actually put in. It makes use of Agda's built-in equality data type, defined as in Listing 9. Because Agda can reduce the left and right hand side of the equivalence to the same value, we can simply use the `refl` constructor to prove the equivalence.

The simplest way to make this model work with `MVar`s is by defining `MVar = IORef Data` and simply let our `MVar` operations run in the `IOs` monad rather than the postulated `IO`. With this approach, however, we lose the ability to write proofs about the content of our memory for programs running in a concurrent setting; for this we would need to adopt Swierstra's $\texttt{IO}_c$ model that has constructors for all `MVar` operations as well as an additional constructor to represent forking threads [11]. Because of time constraints, however, we have opted to retain our `MVar` definition in terms of `IORef` as according to [10], so that we can directly use it with the `IOs` model in the way just specified.

Now that we can run IO operations in a concurrent setting, we can write the example program from subsection 2.2 in Agda. We slightly adjust the `round_robin` function from Listing 4 to return a boolean value, as shown in Listing 10. The boolean value signifies termination: it returns `True` if it stops because there are no actions left to perform, and `False` if it runs out of fuel before all actions have been performed. Recall that the fuel was added to enforce termination and that we will in practice make the value of this argument so large that it will represent infinity. This setup then practically enables us to check a program against deadlocks: if our program deadlocks, it will continue infinitely, i.e. until the `round_robin` function runs out of fuel and returns `False`.

Listing 11 presents the deadlocking program from subsection 2.2 - recall that both the forked thread and the main thread are stuck waiting for the other thread to write to the

[25]For more information on this data type, see `https://plfa.github.io/Equality/`.

```
round_robin : {{Monad m}} → List (Action m) → MyNat → m Bool
round_robin [] _ = return True
round_robin xs Zero = return False
round_robin (Atom x ∷ xs) (Suc n) = do
                              x1 ← x
                              round_robin (xs ++ (x1 ∷ [])) n
round_robin (Fork x y ∷ xs) (Suc n) = round_robin (xs ++ (x ∷ y ∷ [])) n
round_robin (Stop ∷ xs) (Suc n) = round_robin xs n
```

Listing 10: The adjusted `round_robin` function. Returns `True` if all actions are completed or `False` if it runs out of fuel first.

```
fuel = natToMyNat 9223372036854775800
{-# COMPILE AGDA2HS fuel #-}

mVarDeadlock : Bool
mVarDeadlock = runIOs (run (do
                a <- newEmptyMVar
                b <- newEmptyMVar
                fork (do
                        takeMVar a fuel
                        writeMVar b 1 fuel)
                takeMVar b fuel
                writeMVar a 2 fuel
                ) (natToMyNat 100000))
{-# COMPILE AGDA2HS mVarDeadlock #-}


deadlock-proof : mVarDeadlock ≡ False
deadlock-proof = refl
```

```
fuel :: MyNat
fuel = natToMyNat 9223372036854775800

mVarDeadlock :: Bool
mVarDeadlock = runIOs
  (run
     (newEmptyMVar >>=
        \ a ->
          newEmptyMVar >>=
            \ b ->
              fork (takeMVar a fuel >> writeMVar b 1 fuel) >>
                (takeMVar b fuel >> writeMVar a 2 fuel))
     (natToMyNat 100000))
```

Listing 11: The deadlocking program from subsection 2.2 with corresponding (trivial) proof of deadlock in Agda (left) with the corresponding Haskell translation by AGDA2HS (right).

respective `MVars` they are trying to read. Running this program will thus result in the `round_robin` function (initiated by `run`) running out of fuel, which thereupon returns `IOs False`; the `runIOs` function in turn extracts that boolean value. Proving that this program does not terminate can be done by a simple `refl`, because Agda is itself able to reduce the expression to a boolean value through normalization. We do not translate the proof to Haskell. Note that we have to take care that the value used as fuel for `takeMVar` and `writeMVar` is significantly larger than that given to `run`, because otherwise we risk those functions terminating (and thus the action list becoming empty) before the `round_robin` function runs out of fuel (with the alternative model from [11] we could likely be able to avoid this because we can take into account the return value of those functions). Furthermore, we deliberately only use 100000 as fuel to `run` because the larger the value, the longer it takes to type check, and this is clearly plenty for such a short program.

In a similar fashion, we can prove that a version of the program in which the `takeMVar` and `writeMVar` operations of the main thread are reversed, is free of deadlocks (evaluates to `True`).

However, we should also point out that our current model cannot detect *all* possibilities of deadlock: when there are race conditions involved, a deadlock may pass through unnoticed. This is illustrated in Listing 12 - according to the proof, this program terminates in our model. In a genuine concurrent setting, however, *we do not know* whether it terminates:

```
fuel = natToMyNat 9223372036854775800

failDetect : Bool
failDetect = runIOs (run (do
                a <- newEmptyMVar
                mutex <- newMVar 0
                fork (do
                        takeMVar mutex fuel
                        writeMVar a 2 fuel
                        writeMVar mutex 0 fuel)
                takeMVar mutex fuel
                takeMVar a fuel
                writeMVar mutex 0 fuel
                ) (natToMyNat 100000))

fail-proof : failDetect ≡ True
fail-proof = refl
```

Listing 12: Example of a program that may or may not deadlock depending on the order in which the thread operations are interleaved. Because round-robin interleaving is deterministic, we can 'prove' this program to be free of deadlocks, which in a real concurrent setting it is not.

indeed, if the forked thread gets hold of the *mutex* variable first, it will write to *a* and make the *mutex* available for the main thread so that it can retrieve *a*; this is the order of operations of our `round_robin` implementation. But if instead the main thread gets hold of the *mutex* variable first, it will wait forever for the other thread to write to *a*, which it cannot because the *mutex* has already been taken - in this situation a deadlock occurs. The behavior of the program thus depends on the order in which actions are performed, i.e. it has *race conditions* [15]. Because our model is deterministic, we cannot trust the verification of programs that contain such race conditions. It would be interesting to investigate whether this determinism could be circumvented by some form of randomized interleaving or considering multiple interleaving options at once.

6 Related Work

Iris. Iris[26] is a separation logic framework that can be used to verify concurrent programs [16].[27]Although we are not aware of any work done with Iris on concurrent Haskell specifically, as a higher-order framework, Iris can be instantiated with a range of different programming languages, so we would expect it to be usable with e.g. the model discussed in this paper. It is integrated in the Coq proof assistant, so it might be possible to use it in combination with *hs-to-coq*, a tool for automatically translating existing Haskell code to Coq [17]-[18], to obtain proofs for concurrent Haskell programs. Using this approach would have the advantage that an entire framework for verifying concurrent programs already exists in Coq, but not in Agda; on the other hand, the order of translation (from Haskell to Coq, rather than Agda to Haskell) has some disadvantages, because the developer needs to be aware of the limitations of Coq while programming in Haskell. [19] also presents an alternative Coq library for verifying concurrent programs.

[26]`https://iris-project.org/`

[27]Separation logic is an extension of Hoare logic that can be used for reasoning about concurrent programs [20].

Concurrent Haskell Debugger. The Concurrent Haskell Debugger[28] (CHD) is a graphical user interface that helps programmers debug their concurrent programs, by allowing them to go through their program step by step and adjust the execution order. One integrated extension is automatic deadlock detection [21], which redefines the IO monad and detects deadlocks in different possible schedules (other than the one selected by the user) through iterative deepening. As the search is non-exhaustive, it is not guaranteed to discover a deadlock present in the program - but it is thereby also efficient and thus a useful debugging tool.

Run-time verification with LTL. An extension to the CHD enables programmers to verify concurrent programs at run-time using linear temporal logic (LTL) formulae [22]. In particular, it can be used to specify program invariants or propositions that must hold at a particular point in the program. As explained by the developers, such properties cannot be verified (efficiently) at compile time because it is impossible to verify all possible paths of program execution (which may be very large or even infinite in number). Thus these need to be verified at run-time. A disadvantage of this type of verification is that, because the properties are checked during a single execution of the program, if no errors are found, there is no guarantee that the properties hold for *all* possible program executions. Thus it is a useful for debugging our Haskell programs, but it does not provide the formal proofs we would like to see - although our current deterministic model does not yet provide any better guarantees.

Model checking with LTL. A classical method of verifying concurrent programs, if not by handwritten proofs, is by using model checking techniques in combination with LTL, which allows us to formulate properties about concurrent programs such as fairness and liveliness. As mentioned in section 1, one such model checker is Spin. Spin works with its own modeling language, PROMELA, based on Dijkstra's guarded command language.[29]Within the model, one can define invariants that are expected to hold at any point during the program. The model checker can be run with several algorithms to verify these invariants, and can additionally check for properties such as liveliness and fairness. In contrast to our approach, the programmer needs to do the work here twice: both write the program in the desired target language, and represent this program as a verifiable Spin model. Additionally, this brings along the uncertainty of translation: how do we know the model corresponds one-to-one with the program we wish to verify? Although some tools have been developed to automatically extract Spin models from existing source code for some mainstream languages like Java and C, these tools are not flawless and still require the programmer to manually specify which parts of the program need to be modeled and to manually check the model [23]. If we would want to use Spin for our Haskell programs, we would need to create the models manually.

Haskell contracts. Contracts, or *refinement types*, can be used to specify pre- and post conditions of a function, in the form of boolean predicates, which can thereby be statically verified (i.e. at compile time) [25].[30]An advantage of this technique is that contracts are written in Haskell and do not require the programmer to learn another language. Additionally, the functions they use can be non-total or non-terminating, in contrast to Agda functions. However, contracts are restricted to verifying pre- and post conditions and cannot say anything about the relation between multiple functions. In [26], it has been shown how

[28]See `https://www.informatik.uni-kiel.de/~fhu/chd/`. Development of this tool seems to have been discontinued after GHC 5.

[29]For the original introduction to guarded commands, see [24].

contracts can be extended to check for race conditions in Concurrent Haskell's STM.

7 Responsible Research

Finally, we will discuss how the present work adheres to and promotes the research principles as formulated in the Netherlands Code of Conduct for Research Integrity [28].

Transparency. An effort has been made to make this work reproducible and verifiable. The investigation of concurrent Haskell programs with AGDA2HS is far from finished, and it is therefore especially important that others can continue this work without having to start from scratch. To this aim, all code is publicly available[31], including archived alternative code and documentation on specific problems. Additionally, this allows others to verify the obtained results with minimum effort.

Honesty. Not only can the results be verified to be correct, but also have we focused on addressing the encountered difficulties and solutions here. We realize the importance hereof so that anyone continuing this research may avoid getting stuck on the same issues and thereby succeed in carrying it further. The discussions of current limitations serve the same purpose.

Independence. Although the present work was supervised by active developers of AGDA2HS, we believe this has not impacted the objectiveness of our assessment of its suitability to verifying concurrent programs, because as researchers they are also primarily interested in improving their tool, rather than promoting it. Another threat to the independence of this work could have been posed by it being a graded B.Sc. thesis - however, as grades were not directly influenced by the obtained results, there was no conflict of interest in this regard either.

Responsibility. The purpose of this research, namely to contribute to the verification of concurrent Haskell programs, and Haskell libraries in general has been explained already. It is part of the larger aim to enrich the Haskell community with another tool to help create error-free code.

Scrupulousness. We hope the present work along with the aforementioned points of focus advertises its own regard for careful design, implementation, evaluation and documentation.

8 Conclusions and Future Work

In this paper, we have shown how a simple Haskell concurrency model can be ported to Agda and successfully translated to Haskell by AGDA2HS. The code generated by AGDA2HS is readable to and usable for Haskell programmers. As expected, the Agda programmer needs to find workarounds for Agda's demands of totality and termination, which lead to a somewhat different Haskell program than we might want - we saw this in particular in the case where 'fuel' needed to be added to intrinsically non-terminating functions. The effects on the behavior of the resulting Haskell programs are however not necessarily substantial.

Another restriction we encountered was having to represent Haskell `newtype`s as records in Agda, which are then translated into data types, which for our purposes was fine in terms of functionality but undesirably led to more verbose code.

[30]Alternatively, contracts can be checked dynamically, at run-time, as done by e.g. [27].

[31]At `https://github.com/mschifferstein/concurrent-haskell-verification`.

More serious are the restrictions posed by Agda's positivity checker. Converting non-positive data types into ones that are strictly positive takes significant effort (we were not able to do it successfully within the scope of this project) and if done leads to more convoluted Haskell code. On the other hand, turning off the positivity checker as we did seems a viable alternative - one only needs to take care to manually verify and make explicit the assumption that the data type in question is indeed positive in all its use cases.

Verification is still primarily a task for future research. We have made a start by porting the mutable state `IO` model from [11] to Agda and showing how this can be used in combination with [10]'s concurrency model to prove the (non-)termination of concurrent programs for a deterministic action sequence based on round-robin. We have however also seen how this method fails to detect deadlock possibilities in some cases in the presence of race conditions. In fact, as pointed out by [22], it might well be impossible to decisively prove (the absence of) deadlocks, because all possible program executions would need to be considered. Nevertheless, if we can prove that a program does not terminate, we can at least be sure a deadlock is possible in a genuine concurrent setting as well, even though the converse does not hold.

Another unfortunate limitation of our approach is that we cannot be sure that the pure IO model used for our proofs corresponds to the actual behavior of Haskell's IO - we have to simply assume that our model is a correct representation of the Haskell primitives. We can at most compare the behavior of programs run in our model and with Haskell's IO respectively. While such an extra translation step is unavoidable if we want to write proofs in Agda about e.g. IO programs, in practice we would like to replace them by postulates for the AGDA2HS translation, so that the resulting Haskell code works in the environment we expect it to.

The first next step should be to port [11]'s $\texttt{IO}_c$ model and write proofs concerning the results of `MVar` operations, e.g. by inspecting the values read from memory. We expect this to be rather straightforward, as was the modeling of $\texttt{IO}_s$.

There are many opportunities to expand on this work further. It would be interesting to port a model of Haskell's `STM` library to Agda. To this aim, it could be investigated whether the STM model presented in [6] can be translated with AGDA2HS. To extend on the verification of concurrent programs, it might be possible to adapt the proof methods from [4] and [5] to work with (models of) Haskell's concurrency concepts, which would enable us to prove safety and liveness properties with linear-time logic. Finally, we suggest the investigation of models of interleaving operations alternative to the implemented round-robin, to see whether there are any options that perhaps come closer to mimicking genuine concurrency.

References

[1] P. Bjesse, "What is formal verification?," *ACM SIGDA Newsletter*, vol. 35, 12 2005.

[2] S. Anand, D. Sabharwal, J. Chapman, O. Melkonian, U. Norell, L. Escot, and J. Cockx, "Reasonable agda is correct haskell: Writing verified haskell using agda2hs," 2 2022. unpublished.

[3] H. H. Løvengreen, "Basic concurrency theory," 2018. Course notes, version 1.2, unpublished.

[4] E. Bergsten, O. Larsson, T. Rastemo, O. Rutqvist, and A. Standár, "Methods for using agda to prove safety and liveness for concurrent programs," bachelor's thesis, Chalmers University of Technology & University of Gothenburg, 6 2017.

[5] J. Häggström, "Proof checker for extended linear time temporal logic proofs about small concurrent programs," Master's thesis, Chalmers University of Technology, 2018.

[6] L. Hu, *Compiling Concurrency Correctly Verifying Software Transactional Memory.* PhD thesis, University of Nottingham, 6 2012.

[7] S. Marlow, *Parallel and Concurrent Programming in Haskell.* O'Reilly Media, 7 2013.

[8] D. A. Spuler and A. S. M. Sajeev, "Compiler detection of function call side effects," 1 1994.

[9] R. H. Campbell, "Deadlocks," in *Encyclopedia of Parallel Computing* (D. Padua, ed.), pp. 524–527, Boston, MA: Springer US, 2011.

[10] K. Claessen, "A poor man's concurrency monad," *Journal of Functional Programming*, vol. 9, pp. 313–323, 1999.

[11] W. Swierstra, *A functional specification of effects.* PhD thesis, University of Nottingham, 11 2008.

[12] HaskellWiki, "Functor-applicative-monad proposal — haskellwiki„" 2015. `https://wiki.haskell.org/index.php?title=Functor-Applicative-Monad_Proposal` (accessed May 24,2022).

[13] G. R. Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[14] S. Peyton Jones, A. Gordon, and F. Sigbjorn, "Concurrent haskell," in *Proceedings of the 23rd POPL '96*, ACM, 1 1996.

[15] C. von Praun, "Race conditions," in *Encyclopedia of Parallel Computing* (D. Padua, ed.), pp. 1691–1697, Boston, MA: Springer US, 2011.

[16] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Brikedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, 2018.

[17] A. Spector-Zabusky, J. Breitner, C. Rizkallah, and S. Weirich, "Total haskell is reasonable coq," in *CPP 2018: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pp. 14–27, 1 2018.

[18] J. Breitner, Y. Li, J. Wiegley, B. Systems, S. Weirich, A. Spector-Zabusky, and C. Rizkallah, "89 ready, set, verify! applying hs-to-coq to real-world haskell code (experience report)," *Proc. ACM Program. Lang*, vol. 2, p. 16, 2018.

[19] R. Affeldt and N. Kobayashi, "A coq library for verification of concurrent programs," *Electronic Notes in Theoretical Computer Science*, vol. 199, pp. 17–32, 2 2008.

[20] J. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pp. 55–74, 2002.

[21] J. Christiansen and F. Huch, "Searching for deadlocks while debugging concurrent haskell programs," in *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming - ICFP '04*, pp. 28–39, ACM Press, 9 2004.

[22] V. Stolz and F. Huch, "Runtime verification of concurrent haskell programs," *Electronic Notes in Theoretical Computer Science*, vol. 113, pp. 201–216, 1 2005.

[23] G. J. Holzmann, *The spin model checker : primer and reference manual.* Addison-Wesley, 2004.

[24] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, p. 453–457, 8 1975.

[25] D. N. Xu, S. Peyton Jones, and K. Claessen, "Static contract checking for haskell," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '09*, pp. 41–52, ACM Press, 2009.

[26] R. Demeyer and W. Vanhoof, "Verification of transactions in stm haskell using contracts and program transformation," in *Programming Language Approaches to Communication- and Concurrency-cEntric Software (6th International Workshop)* (W. Vanderbauwhede and N. Yoshida, eds.), pp. 47–51, 2013.

[27] R. Hinze, J. Jeuring, and A. Löh, "Typed contracts for functional programming," in *Functional and Logic Programming, 8th International Symposium*, pp. 208–225, 4 2006.

[28] KNAW, NFU, TO2-federatie, Vereniging Hogescholen, and VSNU, "Netherlands code of conduct for research integrity." [Online]. Available: `https://doi.org/10.17026/dans-2cj-nvwu`, 9 2018.