# Hello World

Cornelius Diekmann, Lars Hupel

March 17, 2025

**Abstract**

In this article, we present a formalization of the well-known "Hello, World!" code, including a formal framework for reasoning about IO. Our model is inspired by the handling of IO in Haskell. We start by formalizing the 🌍 and embrace the IO monad afterwards. Then we present a sample `main :: IO ()`, followed by its proof of correctness.

## Contents



**theory** *IO*
**imports**
*Main*
*HOL−Library.Monad-Syntax*
**begin**

# 1 IO Monad

Inspired by Haskell. Definitions from https://wiki.haskell.org/IO_inside

## 1.1 Real World

We model the real world with a fake type.

WARNING: Using low-level commands such as **typedecl** instead of high-level **datatype** is dangerous. We explicitly use a **typedecl** instead of a **datatype** because we never want to instantiate the world. We don't need a constructor, we just need the type.

The following models an arbitrary type we cannot reason about. Don't reason about the complete world! Only write down some assumptions about parts of the world.

**typedecl** *real-world* (‹🌍›)

For examples, see `HelloWorld_Proof.thy`. In said theory, we model `STDIN` and `STDOUT` as parts of the world and describe how this part of the wold can be affected. We don't model the rest of the world. This allows us to reason about `STDIN` and `STDOUT` as part of the world, but nothing more.

## 1.2 IO Monad

The set of all functions which take a 🌍 and return an $'\alpha$ and a 🌍.

The rough idea of all IO functions is the following: You are given the world in its current state. You can do whatever you like to the world. You can produce some value of type $'\alpha$ and you have to return the modified world.

For example, the `main` function is Haskell does not produce a value, therefore, `main` in Haskell is of type `IO ()`. Another example in Haskell is `getLine`, which returns `String`. It's type in Haskell is `IO String`. All those functions may also modify the state of the world.

**typedef** $'\alpha$ *io* = *UNIV* :: (🌍 ⇒ $'\alpha$ × 🌍) *set*
**proof** −
  **show** $\exists x.\ x \in UNIV$ **by** *simp*
**qed**

Related Work: *Programming TLS in Isabelle/HOL* by Andreas Lochbihler and Marc Züst uses a partial function (⇀). **typedecl** *real-world* **typedef** $'\alpha$ *io* = *UNIV* :: (🌍 ⇀ $'\alpha$ × 🌍) *set* **by** *simp* We use a total function. This implies the dangerous assumption that all IO functions are total (i.e., terminate).

The **typedef** above gives us some convenient definitions. Since the model of $'\alpha$ *io* is just a mode, those definitions should not end up in generated code.

**term** *Abs-io* — Takes a 🌍 $\Rightarrow$ $'\alpha \times$ 🌍 and abstracts it to an $'\alpha$ *io*.
**term** *Rep-io* — Unpacks an $'\alpha$ *io* to a 🌍 $\Rightarrow$ $'\alpha \times$ 🌍

## 1.3 Monad Operations

Within an $'\alpha$ *io* context, execute $action_1$ and $action_2$ sequentially. The world is passed through and potentially modified by each action.

**definition** *bind* :: $'\alpha$ *io* $\Rightarrow$ $('\alpha \Rightarrow{} '\beta$ *io*$)$ $\Rightarrow$ $'\beta$ *io* **where** [*code del*]:
*bind* $action_1$ $action_2$ = *Abs-io* ($\lambda world_0$.
*let* ($a$, $world_1$) = (*Rep-io* $action_1$) $world_0$;
($b$, $world_2$) = (*Rep-io* ($action_2$ $a$)) $world_1$
*in* ($b$, $world_2$))

In Haskell, the definition for `bind` (>>=) is:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
(action1 >>= action2) world0 =
   let (a, world1) = action1 world0
       (b, world2) = action2 a world1
   in (b, world2)
```

**hide-const** (**open**) *bind*
**adhoc-overloading** *bind* $\rightleftharpoons$ *IO.bind*

Thanks to **adhoc-overloading**, we can use monad syntax.

**lemma** *bind* (*foo* :: $'\alpha$ *io*) ($\lambda a.$ *bar* $a$) = *foo* $\ggg$= ($\lambda a.$ *bar* $a$)
**by** *simp*

**definition** *return* :: $'\alpha \Rightarrow{} '\alpha$ *io* **where** [*code del*]:
*return* $a$ $\equiv$ *Abs-io* ($\lambda world.$ ($a$, *world*))

**hide-const** (**open**) *return*

In Haskell, the definition for `return` is::

```
return :: a -> IO a
return a world0  =  (a, world0)
```

## 1.4 Monad Laws

**lemma** *left-id*:
**fixes** $f$ :: $'\alpha \Rightarrow{} '\beta$ *io* — Make sure we use our ($\ggg$=).
**shows** (*IO.return* $a$ $\ggg$= $f$) = $f$ $a$
**by**(*simp add*: *return-def IO.bind-def Abs-io-inverse Rep-io-inverse*)

**lemma** *right-id*:
  **fixes** $m :: {}'\alpha\ io$ — Make sure we use our ($\gg=$).
  **shows** $(m \gg= IO.return) = m$
  **by**(*simp add*: *return-def IO.bind-def Abs-io-inverse Rep-io-inverse*)

**lemma** *bind-assoc*:
  **fixes** $m :: {}'\alpha\ io$ — Make sure we use our ($\gg=$).
  **shows** $((m \gg= f) \gg= g) = (m \gg= (\lambda x.\ f\ x \gg= g))$
  **by**(*simp add*: *IO.bind-def Abs-io-inverse Abs-io-inject fun-eq-iff split*: *prod.splits*)

## 1.5 Code Generator Setup

We don't expose our ($\gg=$) definition to code. We use the built-in definitions of the target language (e.g., Haskell, SML).

**code-printing constant** *IO.bind* ⇀ (*Haskell*) - >>= -
  **and** (*SML*) *bind*
  | **constant** *IO.return* ⇀ (*Haskell*) *return*
  **and** (*SML*) (() => -)

SML does not come with a bind function. We just define it (hopefully correct).

**code-printing code-module** *Bind* ⇀ (*SML*) ‹
*fun bind x f* () = *f* (*x* ()) ();
›
**code-reserved** (*SML*) *bind return*

Make sure the code generator does not try to define $'\alpha\ io$ by itself, but always uses the one of the target language. For Haskell, this is the fully qualified Prelude.IO. For SML, we wrap it in a nullary function.

**code-printing type-constructor** *io* ⇀ (*Haskell*) *Prelude.IO* -
  **and** (*SML*) *unit* −> -

In Isabelle, a *string* is just a type synonym for *char list*. When translating a *string* to Haskell, Isabelle does not use Haskell's `String` or `[Prelude.Char]`. Instead, Isabelle serializes its own `data Char = Char Bool Bool Bool Bool Bool Bool Bool Bool`. The resulting code will look just ugly.

To use the native strings of Haskell, we use the Isabelle type *String.literal*. This gets translated to a Haskell `String`.

A string literal in Isabelle is created with *STR ''foo''*.

We define IO functions in Isabelle without implementation. For a proof in Isabelle, we will only describe their externally observable properties. For code generation, we map those functions to the corresponding function of the target language.

Our assumption is that our description in Isabelle corresponds to the real behavior of those functions in the respective target language.

We use **axiomatization** instead of **consts** to axiomatically define that those functions exist, but there is no implementation of them. This makes sure that we have to explicitly write down all our assumptions about their behavior. Currently, no assumptions (apart from their type) can be made about those functions.

```
axiomatization
  println :: String.literal ⇒ unit io and
  getLine :: String.literal io
```

A Haskell module named `StdIO` which just implements `println` and `getLine`.

```
code-printing code-module StdIO ⇀ (Haskell) ‹
module StdIO (println, getLine) where
import qualified Prelude (putStrLn, getLine)
println = Prelude.putStrLn
getLine = Prelude.getLine
›                                  and (SML) ‹
(* Newline behavior in SML is odd.*)
fun println s () = TextIO.print (s ^ \n);
fun getLine () = case (TextIO.inputLine TextIO.stdIn) of
                  SOME s => String.substring (s, 0, String.size s − 1)
                | NONE => raise Fail getLine;
›

code-reserved (Haskell) StdIO println getLine
code-reserved (SML) println print getLine TextIO
```

When the code generator sees the functions *println* or *getLine*, we tell it to use our language-specific implementation.

```
code-printing constant println ⇀ (Haskell) StdIO.println
                           and (SML) println
            | constant getLine ⇀ (Haskell) StdIO.getLine
                           and (SML) getLine
```

Monad syntax and *println* examples.

```
lemma bind (println (STR ''foo''))
           (λ-.  println (STR ''bar'')) =
      println (STR ''foo'') ≫= (λ-. println (STR ''bar''))
  by simp
lemma do { - ← println (STR ''foo'');
           println (STR ''bar'')} =
     println (STR ''foo'') ≫ (println (STR ''bar''))
  by simp
```

## 1.6 Modelling Running an *'α io* Function

Apply some function *iofun* to a specific world and return the new world (discarding the result of *iofun*).

**definition** *exec* :: $'\alpha$ *io* ⇒ 🌍 ⇒ 🌍 **where**
*exec iofun world* = *snd* (*Rep-io iofun world*)

Similar, but only get the result.

**definition** *eval* :: $'\alpha$ *io* ⇒ 🌍 ⇒ $'\alpha$ **where**
*eval iofun world* = *fst* (*Rep-io iofun world*)

Essentially, *exec* and *eval* extract the payload $'\alpha$ and 🌍 when executing an $'\alpha$ *io*.

**lemma** *Abs-io* ($\lambda$*world*. (*eval iofun world*, *exec iofun world*)) = *iofun*
**by**(*simp add*: *exec-def eval-def Rep-io-inverse*)

**lemma** *exec-Abs-io*: *exec* (*Abs-io f*) *world* = *snd* (*f world*)
**by**(*simp add*: *exec-def Abs-io-inverse*)

**lemma** *exec-then*:
*exec* ($io_1$ ≫ $io_2$) *world* = *exec* $io_2$ (*exec* $io_1$ *world*)
**and** *eval-then*:
*eval* ($io_1$ ≫ $io_2$) *world* = *eval* $io_2$ (*exec* $io_1$ *world*)
**by** (*simp-all add*: *exec-def eval-def bind-def Abs-io-inverse split-beta*)

**lemma** *exec-bind*:
*exec* ($io_1$ ≫= $io_2$) *world* = *exec* ($io_2$ (*eval* $io_1$ *world*)) (*exec* $io_1$ *world*)
**and** *eval-bind*:
*eval* ($io_1$ ≫= $io_2$) *world* = *eval* ($io_2$ (*eval* $io_1$ *world*)) (*exec* $io_1$ *world*)
**by**(*simp-all add*: *exec-def eval-def bind-def Abs-io-inverse split-beta*)

**lemma** *exec-return*:
*exec* (*IO.return a*) *world* = *world*
**and**
*eval* (*IO.return a*) *world* = *a*
**by** (*simp-all add*: *exec-def eval-def Abs-io-inverse return-def*)

**end**
**theory** *HelloWorld*
**imports** *IO*
**begin**

## 2 Hello, World!

The idea of a *main* function is that, upon start of your program, you will be handed a value of type 🌍. You can pass this world through your code and modify it. Be careful with the 🌍, it's the only one we have.

The main function, defined in Isabelle. It should have the right type in Haskell.

**definition** *main* :: *unit io* **where**
*main* ≡ *do* {
- ← *println* (*STR* ''*Hello World*! *What is your name?*'');
*name* ← *getLine*;
*println* (*STR* ''*Hello*, '' + *name* + *STR* ''!'')
}

# 3 Generating Code

Checking that the generated code compiles.

**export-code** *main* **checking** *Haskell? SML*

## 3.1 Haskell

The generated code in Haskell (including the prelude) is shown below.

```
module StdIO (println, getLine) where
import qualified Prelude (putStrLn, getLine)
println = Prelude.putStrLn
getLine = Prelude.getLine
```

```
{-# LANGUAGE EmptyDataDecls, RankNTypes, ScopedTypeVariables #-}

module HelloWorld(main) where {

import Prelude ((==), (/=), (<), (<=), (>=), (>), (+), (-), (*), (/),
(**),
  (>>=), (>>), (=<<), (&&), (||), (^), (^^), (.), ($), ($!), (++), (!!),
Eq,
  error, id, return, not, fst, snd, map, filter, concat, concatMap, reverse,
  zip, null, takeWhile, dropWhile, all, any, Integer, negate, abs, divMod,
  String, Bool(True, False), Maybe(Nothing, Just));
import Data.Bits ((.&.), (.|.), (.^.));
import qualified Prelude;
import qualified Data.Bits;
import qualified StdIO;

main :: Prelude.IO ();
main =
  (StdIO.println
    "Hello World! What is your name?") >>= (\ _ ->
     StdIO.getLine >>= (\ name -> StdIO.println (("Hello, " ++ name) ++
"!"));

}
```

### 3.2 SML

The generated code in SML (including the prelude) is shown below.

```
fun bind x f () = f (x ()) ();



(* Newline behavior in SML is odd.*)
fun println s () = TextIO.print (s ^ "\n");
fun getLine () = case (TextIO.inputLine TextIO.stdIn) of
                  SOME s => String.substring (s, 0, String.size s - 1)
                | NONE => raise Fail "getLine";


structure HelloWorld : sig
  val main : unit -> unit
end = struct

val main : unit -> unit =
  bind (println "Hello World! What is your name?")
    (fn _ => bind getLine (fn name => println ("Hello, " ^ name ^ "!")));

end; (*struct HelloWorld*)
```

**end**
**theory** *HelloWorld-Proof*
**imports** *HelloWorld*
**begin**

# 4 Correctness

## 4.1 Modeling Input and Output

With the appropriate assumptions about *println* and *getLine*, we can even prove something. We summarize our model about input and output in the assumptions of a **locale**.

**locale** *io-stdio* =
— We model `STDIN` and `STDOUT` as part of the 🌍. Note that we know nothing about 🌍, we just model that we can find `STDIN` and `STDOUT` somewhere in there.
**fixes** *stdout-of*::🌍 ⇒ *string list*
**and** *stdin-of*::🌍 ⇒ *string list*

— Assumptions about `STDIN`: Calling *println* appends to the end of `STDOUT` and *getLine* does not change anything.
**assumes** *stdout-of-println*[*simp*]:
*stdout-of* (*exec* (*println str*) *world*) = *stdout-of world*@[*String.explode str*]
**and** *stdout-of-getLine*[*simp*]:
*stdout-of* (*exec getLine world*) = *stdout-of world*

— Assumptions about `STDIN`: Calling *println* does not change anything and *getLine* removes the first element from the `STDIN` stream.

**and** *stdin-of-println*[*simp*]:
*stdin-of* (*exec* (*println str*) *world*) = *stdin-of world*
**and** *stdin-of-getLine*:
*stdin-of world* = *inp*#*stdin* ⟹
*stdin-of* (*exec getLine world*) = *stdin* ∧ *eval getLine world* = *String.implode inp*
**begin**
**end**

## 4.2 Correctness of Hello World

Correctness of *main*: If `STDOUT` is initially empty and only *''corny''* will be typed into `STDIN`, then the program will output: [*''Hello World*! *What is your name?''*, *''Hello*, *corny*!*''*].

**theorem** (**in** *io-stdio*)
**assumes** *stdout*: *stdout-of world* = []
**and** *stdin*: *stdin-of world* = [*''corny''*]
**shows** *stdout-of* (*exec main world*) =
[*''Hello World*! *What is your name?''*,
*''Hello*, *corny*!*''*]
**proof** −
**let** *?world1*=*exec* (*println* (*STR ''Hello World*! *What is your name?''*)) *world*
**have** *stdout-world2*:
*literal.explode STR ''Hello World*! *What is your name?''* =
*''Hello World*! *What is your name?''*
**by** *code-simp*
**from** *stdin-of-getLine*[**where** *stdin*=[], *OF stdin*] **have** *stdin-world2*:
*eval getLine ?world1* = *String.implode ''corny''*
**by** (*simp add*: *stdin-of-getLine stdin*)
**show** *?thesis*
**unfolding** *main-def*
**apply**(*simp add*: *exec-bind*)
**apply**(*simp add*: *stdout*)
**apply**(*simp add*: *stdout-world2 stdin-world2*)
**apply**(*simp add*: *plus-literal.rep-eq*)
**apply** *code-simp*
**done**
**qed**

**end**