Implementing Global Variables in Functional Programming

Härmel Nestra

Institute of Computer Science

University of Tartu

e-mail: harmel.nestra@ut.ee

Outline

- Introduction of the problem.
- Five approaches to solve the problem.
  - Code show.
  - The series of the first 4 approaches follows the paper

    John Hughes, *Global Variables in Haskell.* JFP **14**(5) (2004), pp 489–502.
  - The last approach to present, as well as the way of implementing variables via type families (and in fact, all code examples), is my contribution.

The Problem

Preliminaries from Functional Programming

Referential transparency

One of the main features of functional programming is **referential transparency** that means that

- any subexpression affects the value of the whole expression only via its value;
- i.e., any subexpression can be replaced with another with the same value;
- i.e., the evaluation of any expression has no side-effects.

  Side-effects are changes in computation state that could have impact to the values of expressions.

Consequences

- There can be no side-effects at all (as the whole program is also an expression that must be evaluated).
- The value of a variable does not change during reading its scope (otherwise, it could be used for creating side-effects).

  In mathematics analogously, correct interpretation of formula $x^2 = x \cdot x$ assumes that the value of x is the same over the formula, although during different readings, the value of x can be different.

Input from the environment

In practice, it is inevitable to have programs that take information from the environment into account:

- in interactive processes;
- in handling asynchronous exceptions;
- in using memory common to many threads;
- etc..

A former problem, related to that of this talk

How to use the information taken from the environment while preventing it from becoming a side-effect?

- Solution (idea): assure that the values being influenced by the environment are always immediately assigned to variables at their initialization.

Input-output action types

In Haskell, there are special types for computations that introduce values influenced by the environment.

Namely, type

$$\mathrm{IO}\ A$$

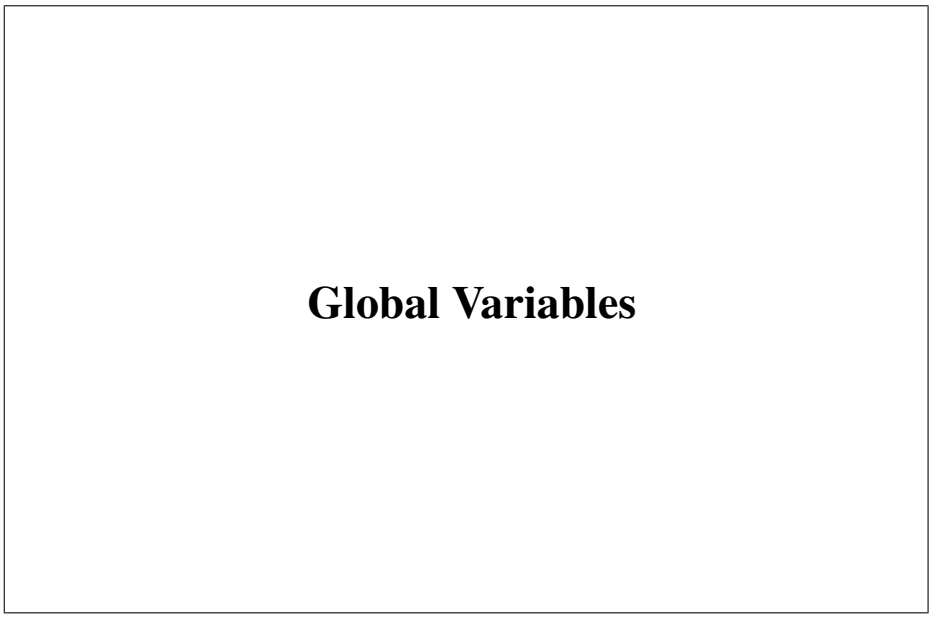
contains **actions** that give rise to a value of type A potentially influenced by the environment.

- This value is called the **return value** of the action.

Operating with memory in functional programming

In functional languages, computing with memory locations with changing content is possible but tricky.

- The address of the location, the reference, must differ from the address of any variable.
- In order to avoid side-effects arising from changing the contents, the contents are treated as potentially influenced by the environment.


Global Variables

Global variable

In imperative languages, global variables justify the words:

- their scope is the whole module;
- their R-value can change during the computation.

Global variables are typically used for holding data structures that are needed during the whole computation.

The big problem

The values of variables with global scope cannot change.

A freely changing entity can be assigned only to local variables at their initialization.

What is the problem?

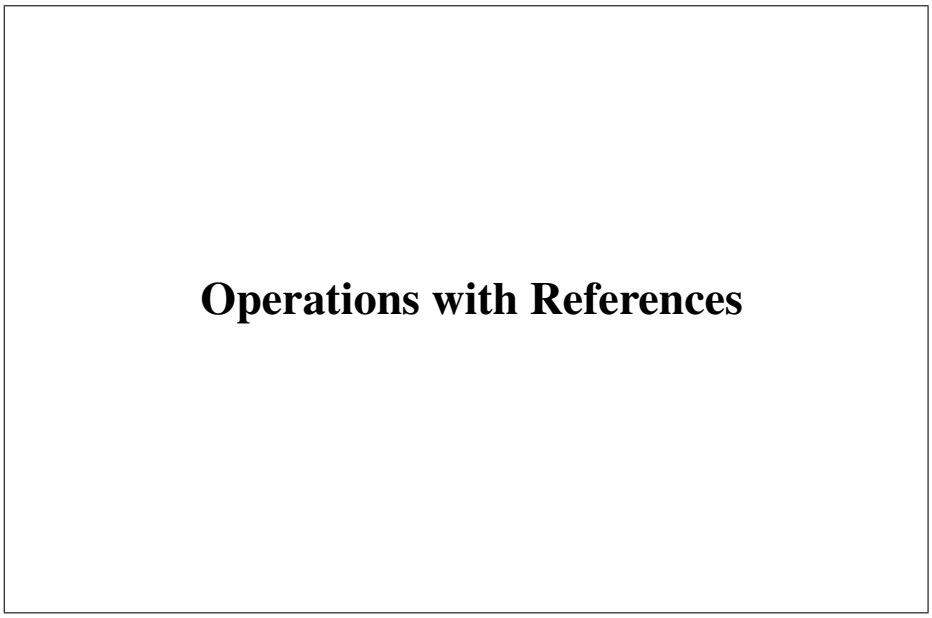
But wait: having a globally visible reference to a memory location is sufficient.

- Because then any part of code can make use of the location via dereferencing.

This means that only the variable that holds the reference has to be assured to be constant.

But the reference corresponds to the L-value of the variable of imperative languages which never changes during the reading of its scope!

What is then the problem?


Operations with References

Input-output references

In the case of input-output references, the changing contents of memory locations are handled by the same mechanism as the influence of environment.

Input-output reference types and operations with them are exported by the non-standard module Data.IORef.

Type

Type

IORef A

contains references to entities of type A.

Reading and writing

References are used by the following operators:

reading (dereferencing):

```
readIORef ::  IORef a -> IO a,
```

writing (updating):

```
writeIORef ::  IORef a -> a -> IO ().
```

Creation

Creating a new reference together with initialization:

```
newIORef ::  a -> IO (IORef a).
```

Note that creating a reference gives an action rather than the reference itself.

Hence, the reference is only accessible locally…

The root of the initial problem

A new reference (the address of a fresh memory location) is a bit of information from the environment.

- If the value of expressions of the form `newIORef a` were references, the value of such an expression should not depend on the value of the argument `a`.
  - In other words, referential transparency would demand that equal initial values implied equal memory locations.

  Of course, this is the last to be desired.

State thread references

There is another kind of references that rely on state thread actions rather than input-output actions.

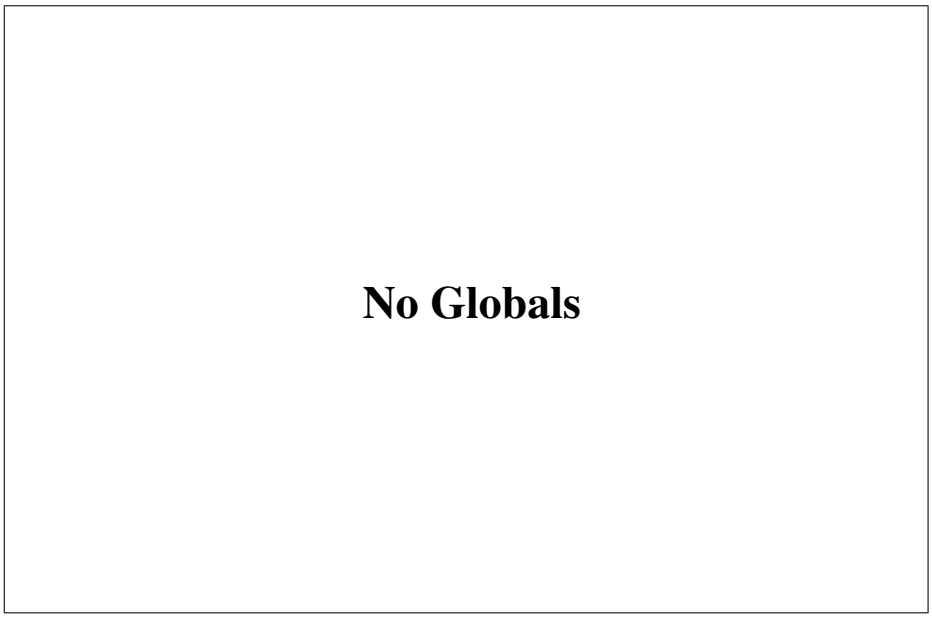
- State thread actions enable to make difference between computations with result really independent from the environment and others.

  From the formers, the result can be used without restrictions (unlike in the case of input-output actions).

- They make use of higher-order polymorphism.

Using these references ends up with the same problem as input-output references.

Solutions


No Globals

No globals

This means that all entities that must be used everywhere over the code must be passed around as parameters.

Advantages

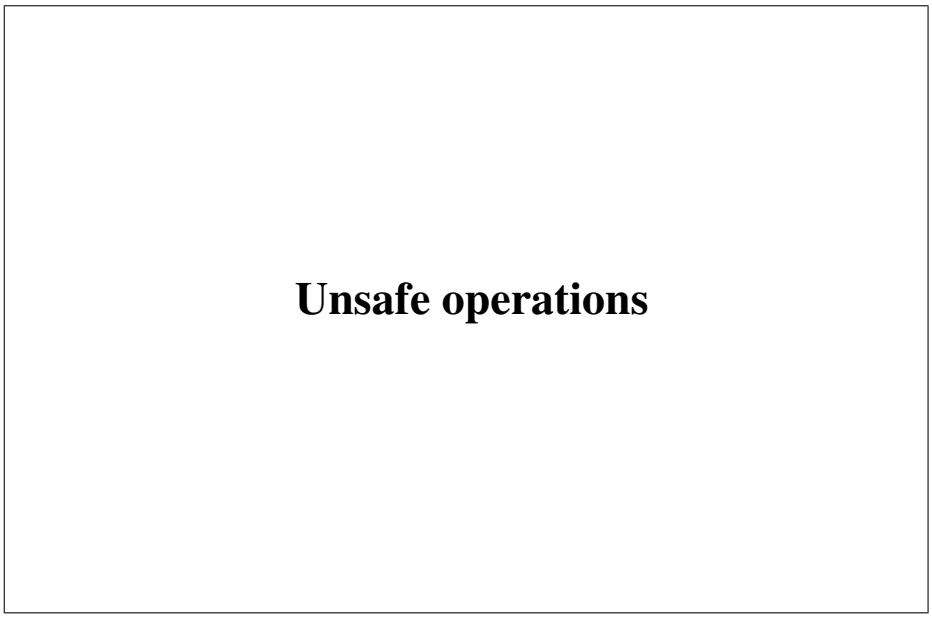
- Perhaps improves flexibility: explicit parameters can overuse same names etc.

Drawbacks

- More work for the programmer.
- Clunky code if compared to imperative languages.
- Implementations of algorithms differ from their classical versions considerably.

Corollary

This approach is a non-solution.


Unsafe operations

Operator `unsafePerformIO`

In GHC and Hugs, there is a non-standard module System.IO.Unsafe that provides

```
unsafePerformIO ::  IO a -> a.
```

With this, one can simply take out the return value from an action.

This way, one could create references for global data and make them globally visible.

Advantages

- Easy to use.
- The references created are truly global.

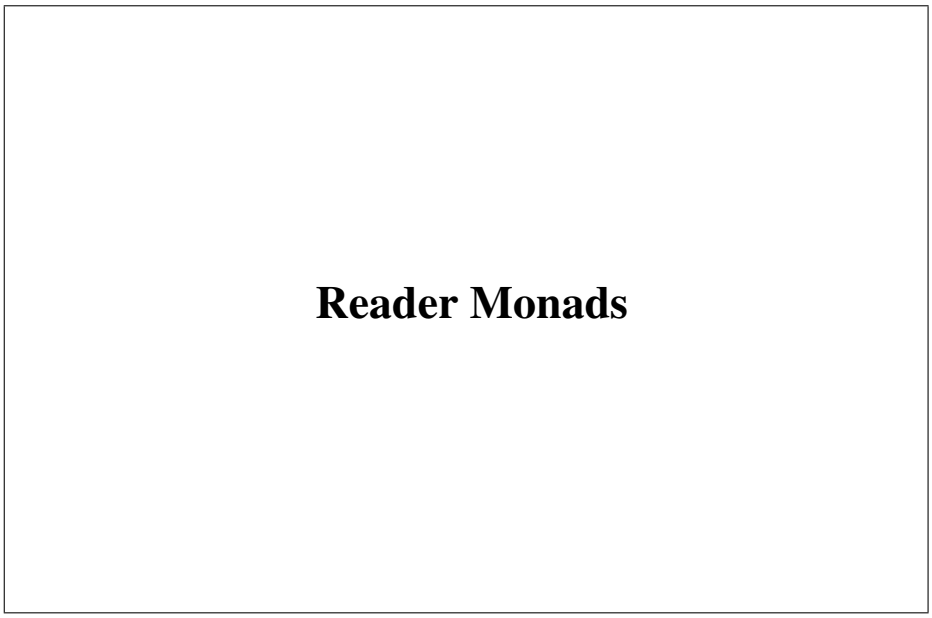
Drawbacks

- This is not functional programming.
  - Violates referential transparency.

  An operator like `unsafePerformIO` should not exist!
- `unsafePerformIO` is unsafe, using it is undesirable for an average programmer.
  - Using it, one can easily create runtime type errors etc., that are supposed to never arise in functional programming.

Corollary

This approach is not a solution.


Reader Monads

Reader monads

If M is a monad and R is an arbitrary type then the type function mapping any type A to type $R \to M\,A$ is also a monad, so-called reader monad.

The idea:

- there is one copy of $M\,A$ for each value from type R
- and the usual programming in monad M goes on, in all copies simultaneously.

Point-free style

In principle, this requires programming in point-free style.

- All definitions are specified at the function level.
  Arguments are not explicitly given.

Advantages

- It does not go beyond the functional paradigm.
- It enables to hide the parameter that is passed around.

Drawbacks

- Hard to understand by average programmers.
- The references are not really global.
- Instead of repeating the reference parameter, other clunky repeating things are needed in order to keep type correctness.

Corollary

This approach addresses a wrong problem.

- The initial problem was not in code repetition but in accessibility.

Implicit Parameters

Implicit parameters

Implicit parameters are implicit in the code function arguments that are nevetheless consumed.

- Parametric polymorphism.
- GHC enables also implicit data parameters.

The principles

- Implicit paramters are shown by the type context.
- The names of implicit parameters are visible in the definition. (They start with question mark.)
- At the call of an operator having implicit parameters from a context having the same implicit parameters, the values of these parameters are silently passed unchanged, if the code does not say otherwise.
- Implicit data parameters can be bound also explicitly (e.g., in a code fragment whose context does not contain this parameter).

Implicit data parameters can be (and, in GHC, actually are) implemented via dynamic scoping.

Advantages

- It does not go beyond the functional paradigm.
- The parameter is passed silently where necessary.
- Easy to learn.
- Little trouble in using.

Drawbacks

- Implicit parameters are not truly global.
- Implicit parameters still appear in type contexts.

Corollary

Implicit parameters are surprisingly useful but the solution to our problem is supposititious. (Again, the wrong problem is addressed.)

My 2 cents

Variable type family

A central module `Var` could declare a data family

```
Var ::  * -> *.
```

For each type A, it gives a type of "variables of type A".

- User modules can provide the real variable types as data instances.
- This way:
  - variables get names
  - whereby, technically, these names are data constructors.

Explicit variable environment

A variable environment is a function from variables of some type to references to data of that type:

```
type Env a
  = Var a -> IORef a
```

- For efficiency, this function should be implemented as a lookup from an array.
  - Note that the array would be read-only.

Global environment

Another central module `Global` could then implement global environment:

- a type class `AbstrRef` of type pairs,

  relating, by intention, variable types with corresponding reference types;

- a class method `globalEnv :: Env a`

  that is implemented as unsafe creation of a new environment.

  * This use of `unsafePerformIO` is safe as `globalEnv` is evaluated only once.

Advantages

- Users do not have to go beyond functional programming.
  - Unsafe operator occurs only in the central module.
  - Alternatively, the explicit global environment could be implemented directly into the language by its designers.
    - Referential transparency is not violated as the references are depending on variables rather than their initial values.
- References are truly global.
- Easy to use.
- Code becomes close to mainstream imperative programming.

Drawbacks

- All globals of the same type that are used simultaneously must be introduced together, in one declaration.
  - Haskell does not have open data declarations.
  - But this drawback for programmers may be considered as an advantage for readers.