Functions to Junctions

Ultra Low Power Chip Design with Some Help from Haskell

Gregory Wright

Antiope Associates
Fair Haven, New Jersey, USA
gwright@antiope.com

October 6, 2008

antiope

Outline

antiope

A bit about us

Antiope Associates is a small company specializing in radio system design.

The product

The product is a wireless tag for displaying prices in stores. It had to:

- be cheap
- have high communication throughput (update an entire store in one shift)
- be reliable (no more than one undetected error in a million messages
- run on little power (minimum five year life using a small coin cell battery)

antiope

The product

The product is a wireless tag for displaying prices in stores. It had to:

- be cheap
- have high communication throughput (update an entire store in one shift)
- be reliable (no more than one undetected error in a million messages
- run on little power (minimum five year life using a small coin cell battery)

antiope

The product

The product is a wireless tag for displaying prices in stores. It had to:

- be cheap
- have high communication throughput (update an entire store in one shift)
- be reliable (no more than one undetected error in a million messages
- run on little power (minimum five year life using a small coin cell battery)

antiope

The product

The product is a wireless tag for displaying prices in stores. It had to:

- be cheap
- have high communication throughput (update an entire store in one shift)
- be reliable (no more than one undetected error in a million messages
- run on little power (minimum five year life using a small coin cell battery)

antiope

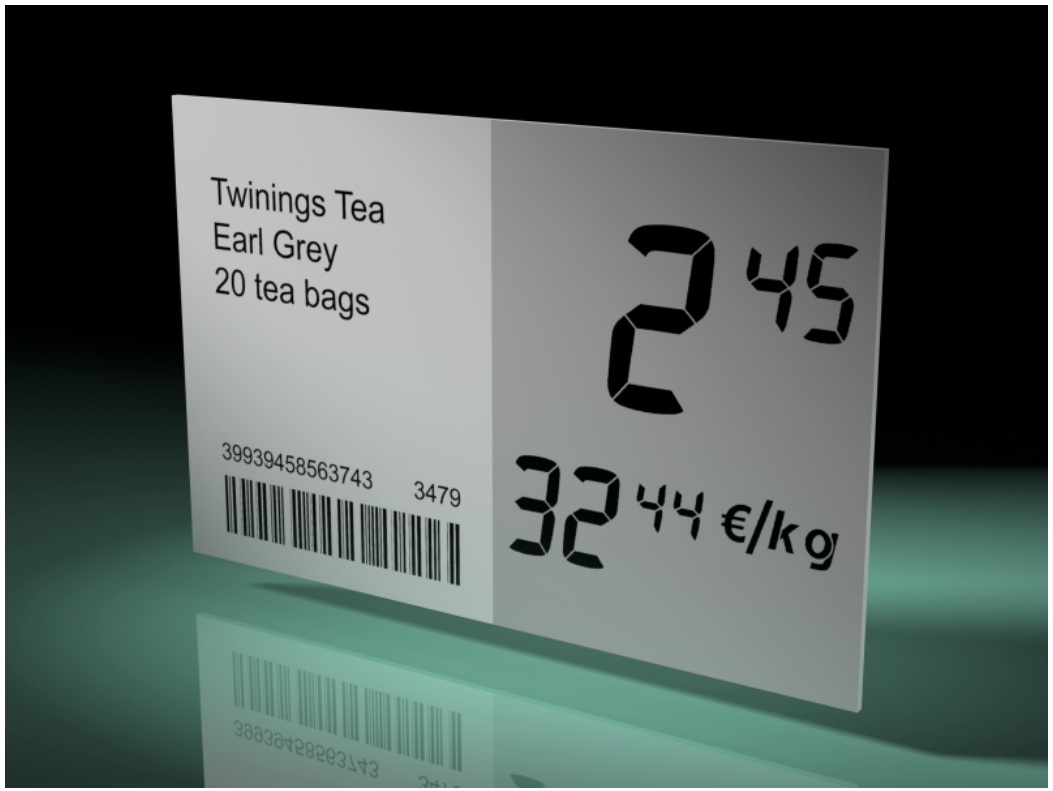
The product

The product is a wireless tag for displaying prices in stores. It had to:

- be cheap
- have high communication throughput (update an entire store in one shift)
- be reliable (no more than one undetected error in a million messages
- run on little power (minimum five year life using a small coin cell battery)

antiope

The product



antiope

The product

Antiope Associates contributed the system architecture, protocol design and hardware design for the link layer protocol engine.

A chip design company with expertise in mixed signal design did the analog portions and the embedded microcontroller.

The problem

Three of the requirements — speed, reliability and low power — are intimately connected with the design of the communication protocol.

So the task was to design a protocol met these goals and was

- easily implemented in hardware, and
- had reasonable assurance of correctness.

The problem

Three of the requirements — speed, reliability and low power — are intimately connected with the design of the communication protocol.

So the task was to design a protocol met these goals and was

- easily implemented in hardware, and
- had reasonable assurance of correctness.

The problem

Three of the requirements — speed, reliability and low power — are intimately connected with the design of the communication protocol.

So the task was to design a protocol met these goals and was

- easily implemented in hardware, and
- had reasonable assurance of correctness.

Why we used Haskell

- During the 1999 and 2000 academic years I was a visitor at the Berkeley Wireless Research Center (part of the UC Berkeley EECS department). Part of our research looked at functional languages for executable specification of communication systems.
- For another company I designed a lisp-like language for representing simulations.
- Antiope Associates first used Haskell on a small project for a government customer 4 years ago.

antiope

Why we used Haskell

- During the 1999 and 2000 academic years I was a visitor at the Berkeley Wireless Research Center (part of the UC Berkeley EECS department). Part of our research looked at functional languages for executable specification of communication systems.
- For another company I designed a lisp-like language for representing simulations.
- Antiope Associates first used Haskell on a small project for a government customer 4 years ago.

antiope

Why we used Haskell

- During the 1999 and 2000 academic years I was a visitor at the Berkeley Wireless Research Center (part of the UC Berkeley EECS department). Part of our research looked at functional languages for executable specification of communication systems.
- For another company I designed a lisp-like language for representing simulations.
- Antiope Associates first used Haskell on a small project for a government customer 4 years ago.

antiope

Why we used Haskell

- During the 1999 and 2000 academic years I was a visitor at the Berkeley Wireless Research Center (part of the UC Berkeley EECS department). Part of our research looked at functional languages for executable specification of communication systems.
- For another company I designed a lisp-like language for representing simulations.
- Antiope Associates first used Haskell on a small project for a government customer 4 years ago.

How we used Haskell

We used Haskell in two ways in the design process:

- It was the principal language used for our system simulator.
- We used it for numerous utilities that tied together the design verification and testing processes.

How we used Haskell

We used Haskell in two ways in the design process:

- It was the principal language used for our system simulator.
- We used it for numerous utilities that tied together the design verification and testing processes.

How we used Haskell

We used Haskell in two ways in the design process:

- It was the principal language used for our system simulator.
- We used it for numerous utilities that tied together the design verification and testing processes.

Our approach

Our general approach to the problem was:

- Write a program to simulate the entire system — multiple base stations and tags — in compiled Haskell. This would let us run many simulations in a reasonable time.
- Put all of the code which described what the hardware was supposed to do in one module. Functions that represented hardware blocks would be single entities in the final VHDL.
- Run and debug. Repeat until done (or out of time).
- Manually translate the code that described the hardware into VHDL.

Our approach

Our general approach to the problem was:

- Write a program to simulate the entire system — multiple base stations and tags — in compiled Haskell. This would let us run many simulations in a reasonable time.
- Put all of the code which described what the hardware was supposed to do in one module. Functions that represented hardware blocks would be single entities in the final VHDL.
- Run and debug. Repeat until done (or out of time).
- Manually translate the code that described the hardware into VHDL.

antiope

Our approach

Our general approach to the problem was:

- Write a program to simulate the entire system — multiple base stations and tags — in compiled Haskell. This would let us run many simulations in a reasonable time.
- Put all of the code which described what the hardware was supposed to do in one module. Functions that represented hardware blocks would be single entities in the final VHDL.
- Run and debug. Repeat until done (or out of time).
- Manually translate the code that described the hardware into VHDL.

antiope

Our approach

Our general approach to the problem was:

- Write a program to simulate the entire system — multiple base stations and tags — in compiled Haskell. This would let us run many simulations in a reasonable time.
- Put all of the code which described what the hardware was supposed to do in one module. Functions that represented hardware blocks would be single entities in the final VHDL.
- Run and debug. Repeat until done (or out of time).
- Manually translate the code that described the hardware into VHDL.

Our approach

Our general approach to the problem was:

- Write a program to simulate the entire system — multiple base stations and tags — in compiled Haskell. This would let us run many simulations in a reasonable time.
- Put all of the code which described what the hardware was supposed to do in one module. Functions that represented hardware blocks would be single entities in the final VHDL.
- Run and debug. Repeat until done (or out of time).
- Manually translate the code that described the hardware into VHDL.

Hardware design in Haskell done backwards

The style we used was a bit different than the one usually associated with functional languages.

- The simulator itself was written in a functional style, making good use of higher order functions.
- The simulated hardware was written in a purely imperative style. Every operation on a mutable variable corresponded to an operation on a register in the real hardware.

At the end of all of this we have not only a detailed description of the protocol engine (at the “register transfer level”) but the simulation serves as an executable specification of other parts of the system.

antiope

Hardware design in Haskell done backwards

The style we used was a bit different than the one usually associated with functional languages.

- The simulator itself was written in a functional style, making good use of higher order functions.
- The simulated hardware was written in a purely imperative style. Every operation on a mutable variable corresponded to an operation on a register in the real hardware.

At the end of all of this we have not only a detailed description of the protocol engine (at the “register transfer level”) but the simulation serves as an executable specification of other parts of the system.

antiope

Hardware design in Haskell done backwards

The style we used was a bit different than the one usually associated with functional languages.

- The simulator itself was written in a functional style, making good use of higher order functions.
- The simulated hardware was written in a purely imperative style. Every operation on a mutable variable corresponded to an operation on a register in the real hardware.

At the end of all of this we have not only a detailed description of the protocol engine (at the “register transfer level”) but the simulation serves as an executable specification of other parts of the system.

antiope

Hardware design in Haskell done backwards

The style we used was a bit different than the one usually associated with functional languages.

- The simulator itself was written in a functional style, making good use of higher order functions.
- The simulated hardware was written in a purely imperative style. Every operation on a mutable variable corresponded to an operation on a register in the real hardware.

At the end of all of this we have not only a detailed description of the protocol engine (at the “register transfer level”) but the simulation serves as an executable specification of other parts of the system.

antiope

A simulator

```
stepStore :: Store
          -> LPH.Heap Event
          -> IO (Store, LPH.Heap Event)
stepStore s q = do
        let
                Event (time, act) = LPH.minElem q
                bsc  = storeBSC  s
                bs   = storeBS   s
                tags = storeTags s
                updateQueue :: [Event] -> IO (Store, LPH.Heap Event)
                updateQueue evList = do
                    if null evList
                       then return (s, LPH.deleteMin q)
                       else if timestamp (head evList) < endTime
                                then return (s, (LPH.insertSeq evList)
                                                (LPH.deleteMin q))
                                else return (s, LPH.deleteMin q)
        case act of
                Wakeup tagNum -> do
                        tag   <- readArray tags tagNum
                        nbits <- bitWindow tag
                        buf   <- getAir bsc bs bitErrorRate nbits
                                        (time `div` (fromIntegral clocksPerBit))
                        ev    <- runTag tag time buf
                        updateQueue ev
                Acknowledge tagNum -> do
                        putStrLn ("tag " ++ show tagNum ++ " acknowledges!")
                        frameAck bs tagNum
                        updateQueue []
                TxFrame baseNum -> do
                        ev    <- runBaseStation bsc bs time
                        updateQueue ev
```

antiope

Why this works

If the simulator fits on one page, you can spend your time on modeling power consumption and the radio channel. In other words, solving the real problem.

Not only that, but it's surprisingly efficient:

- For tens of the thousands of tags, the simulation runs a few times slower than real time on a uniprocessor.
- Although we didn't need it, the simulation could probably have been run on multiple processors. We probably would have used an Erlang style message passing scheme instead of the finer grained parallelism popular for Haskell these days.

antiope

Why this works

If the simulator fits on one page, you can spend your time on modeling power consumption and the radio channel. In other words, solving the real problem.

Not only that, but it's surprisingly efficient:

- For tens of the thousands of tags, the simulation runs a few times slower than real time on a uniprocessor.
- Although we didn't need it, the simulation could probably have been run on multiple processors. We probably would have used an Erlang style message passing scheme instead of the finer grained parallelism popular for Haskell these days.

antiope

Why this works

If the simulator fits on one page, you can spend your time on modeling power consumption and the radio channel. In other words, solving the real problem.

Not only that, but it's surprisingly efficient:

- For tens of the thousands of tags, the simulation runs a few times slower than real time on a uniprocessor.
- Although we didn't need it, the simulation could probably have been run on multiple processors. We probably would have used an Erlang style message passing scheme instead of the finer grained parallelism popular for Haskell these days.

antiope

Did the Plan Work?

For the most part, the original idea worked. We were able to centralize the description of the protocol into one module, simulate and debug it, then translate (by hand) the functions used to define the protocol into VHDL.

The protocol engine worked and in several months of testing, we have uncovered only one minor bug.

Business issues: how did FP help us?

Haskell helped get the job done faster.

- We were able to make good use of higher order functions and the other abstraction mechanisms in Haskell.
- An elegant syntax helps.

Business issues: how did FP help us?

Haskell helped get the job done faster.

- We were able to make good use of higher order functions and the other abstraction mechanisms in Haskell.
- An elegant syntax helps.

Business issues: how did FP help us?

Haskell helped get the job done faster.

- We were able to make good use of higher order functions and the other abstraction mechanisms in Haskell.
- An elegant syntax helps.

Business issues: how did FP help us?

In retrospect, we got a lot of benefit from using the already known strengths of Haskell.

- Our big simulation gave us a specification for the link layer hardware in the tag, and a specification for the link and network layers in base station as well.
- We wrote a large number of programs that parsed and unparsed simulation data to generate many tests.

antiope

Business issues: how did FP help us?

In retrospect, we got a lot of benefit from using the already known strengths of Haskell.

- Our big simulation gave us a specification for the link layer hardware in the tag, and a specification for the link and network layers in base station as well.
- We wrote a large number of programs that parsed and unparsed simulation data to generate many tests.

antiope

Business issues: how did FP help us?

In retrospect, we got a lot of benefit from using the already known strengths of Haskell.

- Our big simulation gave us a specification for the link layer hardware in the tag, and a specification for the link and network layers in base station as well.
- We wrote a large number of programs that parsed and unparsed simulation data to generate many tests.

antiope

Business issues: how much did FP help us?

A fair question is how much of our methodology bound up with our choice of tools.

- At the very least, Haskell (and FP in general) share a number of common notions with circuit design, even if terminologies differ.

Business issues: how much did FP help us?

A fair question is how much of our methodology bound up with our choice of tools.

- At the very least, Haskell (and FP in general) share a number of common notions with circuit design, even if terminologies differ.

Business issues: what problems can't we solve?

We've found that people who can handle functional programming idioms are also usually good at breaking a task down into tractable pieces.

- In short, they are good at design.
- But should (or can) every programmer be a designer?

Business issues: what problems can't we solve?

We've found that people who can handle functional programming idioms are also usually good at breaking a task down into tractable pieces.

- In short, they are good at design.
- But should (or can) every programmer be a designer?

antiope

Business issues: what problems can't we solve?

We've found that people who can handle functional programming idioms are also usually good at breaking a task down into tractable pieces.

- In short, they are good at design.
- But should (or can) every programmer be a designer?

Reactions to Haskell

How did we manage our partner's concerns about using a functional language in this project?

- We didn't tell them.
- Well, we did tell them later on. The chip designers we worked with used VHDL, so typeful programming was not a mystery.
- We never had to deliver Haskell code to our partners or customer...
- ...but we could have.

antiope

Reactions to Haskell

How did we manage our partner's concerns about using a functional language in this project?

- We didn't tell them.
- Well, we did tell them later on. The chip designers we worked with used VHDL, so typeful programming was not a mystery.
- We never had to deliver Haskell code to our partners or customer...
- ...but we could have.

antiope

Reactions to Haskell

How did we manage our partner's concerns about using a functional language in this project?

- We didn't tell them.
- Well, we did tell them later on. The chip designers we worked with used VHDL, so typeful programming was not a mystery.
- We never had to deliver Haskell code to our partners or customer...
- ...but we could have.

antiope

Reactions to Haskell

How did we manage our partner's concerns about using a functional language in this project?

- We didn't tell them.
- Well, we did tell them later on. The chip designers we worked with used VHDL, so typeful programming was not a mystery.
- We never had to deliver Haskell code to our partners or customer...
- ...but we could have.

antiope

Reactions to Haskell

How did we manage our partner's concerns about using a functional language in this project?

- We didn't tell them.
- Well, we did tell them later on. The chip designers we worked with used VHDL, so typeful programming was not a mystery.
- We never had to deliver Haskell code to our partners or customer...
- ...but we could have.

antiope

Summary

Expected benefit:

- Our Haskell simulation program helped get this job done faster. We were able to write a simulation that went from register level in the tag to physical effects in the radio channel quickly.

Unexpected benefit:

- Haskell was a good “glue” to connect the various pieces of the design task together.

antiope

Summary

Haskell: the preferred language of highly evolved predators.