Why Haskell Does Not Matter

Stefan Holdermans

Dutch HUG Day 2010
April 24, 2010

Vector Fabrics

Paradijslaan 28
5611 KN Eindhoven
The Netherlands
E-mail: stefan@vectorfabrics.com

About Me

- Started programming in Haskell as a CS undergraduate at Utrecht University (1998).
- MSc Software Technology (2005).
- PhD studies in the field of type-based program analysis (2005–2010).

- Software Development Engineer at Vector Fabrics (since March 1, 2010).

Outline

1. About Vector Fabrics
2. Some of the Challenges We Face
3. Some of Our Experiences with Functional Programming
4. Conclusion

Outline

1 About Vector Fabrics

2 Some of the Challenges We Face

3 Some of Our Experiences with Functional Programming

4 Conclusion

Vector Fabrics

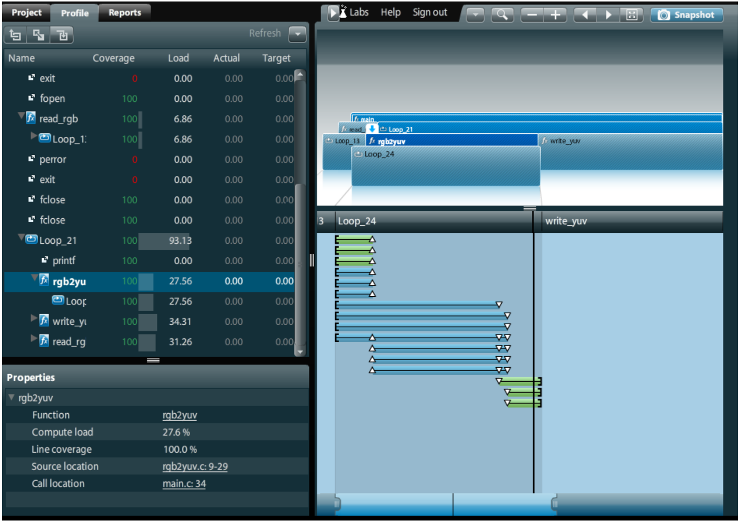
- High-tech start-up, founded in February 2007.
- Unique technology for designing and implementing multicore/multithreaded systems.
- 15 employees.
- Funded with venture capital from the US and the Netherlands.

- vfAnalyst:
  - First released product (March 1st, 2010).
  - On-line source-code analysis.
- Future products focus on program transformation and mapping to hardware.

vfAnalyst
Overview

- Customers
  - log in at `www.vectorfabrics.com`;
  - upload C source files and an accompanying test set through a web interface.
- vfAnalyst
  - runs and analyses the code;
  - visualises properties of interest;
  - pinpoints performance bottlenecks;
  - uncovers opportunities for parallelisation;
  - generates reports.

vfAnalyst
Web Interface



vfAnalyst
Under the Hood

- Combination of static and dynamic analysis.
- C compiler that targets a propriety assembly format.
- Small virtual machine executes assembly and obtains precise information about
  - how different parts of the program communicate through different memory locations;
  - execution times of different functions;
  - code coverage.

Practices

- Short release cycles split up in short (2-to-4-week) development iterations.
- Daily stand-up meetings.
- Collective code ownership.
- Firm coding standards.
- Distributed SCM.
- Pair programming.
- Frequent code reviews.
- Continuous integration.
- Fully automatic unit tests.
- Fully automatic acceptance/regression tests.

Nonpractices

We don’t use Haskell.

Caml

- Choice for Caml was made already before the company was officially founded.
- Established research contact between Utrecht University and Philips Research sparked an interest in functional programming amongst the founders.
- Of the languages considered (Caml and Haskell), Haskell was recognised as the more mature and the elegant.
- Still, eager evaluation tipped the balance to Caml.
  - In some experiments performance of Haskell programs was very hard to predict.
  - In these experiments Caml performed much more consistently.

- Concern for a small company: how do we recruit Caml programmers?

Outline

Many Different Programming Languages

- GUI: Flex/ActionScript.
- Database management: Caml.
- Static analyses: Caml.
- Dynamic analyses: C.
- C compiler: Caml, C++ (LLVM).
- Virtual machine: C.
- Content-management system: PHP (Joomla).

- A fair dose of Python and shell scripts to glue everything together.

- Each of these components communicates data, often to different machines.

Interoperability

- Caml has a good foreign-function interface to C. . . but not much else.
    - To enable communication between components written in Caml and the Flex GUI, Caml functions need to be exposed through a C wrapper.
    - From the C wrapper, a tool called SWIG (Simplified Wrapper and Interface Generator) these C wrappers can be interfaced with in Python.
    - The Python wrappers, finally, can be immediately called from Flex GUI.

Data Marshalling

- With all this indirection, we really need to keep the amount of effort spent in marshalling data to a minimum.
- Caml functions that are exposed to the GUI only take simple types, such as integers and strings, as input.
- They produce either simple types or XML.
- The produced XML is either processed by the server-side Python scripts or forwarded to the client-side Flex GUI (resulting in quite a lot of bandwidth usage).

- Maintaining, extending, or modifying the communication between components is difficult and hard to debug.

Database Access

- Similiar issues arise in communication with our database.
- Both the Caml and Python components interact with a single MySql database.
- However, the bindings for these two languages make different assumptions about the behaviour of the database.

Outline

A Embedded-system Designer's Perspective

- Unfamiliar to functional programming before joining Vector Fabrics.
- Reasonable easy to learn Caml.
- Biggest hurdle: adapt to the "functional mindset":
  - use recursion rather than for-loops;
  - use higher-order functions;
  - exploit polymorphism.
- Biggest grievance: very poor type-error messages.
- Main advantages: productivity, easier-to-maintain code.
- Main drawback: the price paid for high-level abstractions.

A Haskell Programmer's Perspective

- We adapted to Caml very quickly.
- We really miss having type signatures in the main source files.
- We really miss freely (re)ordering function definitions and type declarations.
- We make some use of the camlp4 preprocessor to extend Caml's syntax.
- We make good use of the module system to organise software both in the small and the large:
  - compared to Haskell type classes, the explicitness of modules is sometimes a gain, sometimes a loss;
  - the structuring of modules sometimes get quite complex.
- We deliberately limit the use of mutable references.
- We think of eagerness as a mixed blessing. ▸ Some code

Outline

1 About Vector Fabrics

2 Some of the Challenges We Face

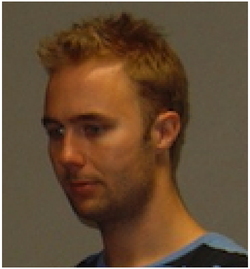
3 Some of Our Experiences with Functional Programming

4 Conclusion

Summary

- Trying to build a successful company that uses functional programming for its core development tasks is lots of fun.
- We need a couple of success stories in order to convince the next high-tech start-up to adapt Haskell.

Acknowledgements

- Alexey Rodriguez Yakushev
- Paul Stravers
- Wouter Swierstra

Evaluation Order

```ocaml
let f x y = () (* TODO: ... *)
let prerr_usage = prerr_string "Usage: foo arg1 arg2\n"
let main =
  if Array.length Sys.argv <= 2 then      (* check #args *)
  begin prerr_usage; exit 1 end else      (* too few: print error *)
  let x = Sys.argv.(1) in                 (* enough: do some work *)
  let y = Sys.argv.(2) in
  f x y
```

```
$ ./foo 2 3 5 7 11
Usage: foo arg1 arg2

$
```

Searching a List

A Haskeller's Take

```
(* hd : α list → α *)
let hd = function x :: _ → x

(* filter : (α → bool) → α list → α list *)
let rec filter p = function
    | [] → []
    | x :: xs → if p x then x :: filter p xs else filter p xs
```

```
(* find : (α → bool) → α list → α *)
let find p xs = hd (filter p xs)
```

Searching a List
Using Exceptions

```
exception Not_found

(* iter : (α → unit) → α list → unit *)
let rec iter f = function
    | [] → ()
    | x :: xs → f x; iter f xs
```

```
exception Found of my_type

(* find : (my_type → bool) → my_type list → my_type *)
let find p xs =
   try
      iter (fun x → if p x then raise (Found x) else ()) xs;
      raise Not_found
   with Found x → x
```

Searching a List

Using a Dedicated Function

```
(* find : (α → bool) → α list → α *)
let rec find p = function
    | [] → raise Not_found
    | x :: xs → if p x then x else find p xs
```

What About Searching a Tree?
Using Exceptions

type α tree $=$ Leaf **of** α | Node **of** α tree $\times$ α tree

```
(* find : (α → bool) → α tree → α *)
let rec find p = function
  Leaf x → if p x then x else raise Not_found
  Node (l, r) → try find p l with Not_found → find p r
```

What About Searching a Tree?

Using Optional Values

```
type α tree = Leaf of α | Node of α tree × α tree
type α option = None | Some of α
```

```
(* find : (α → bool) → α tree → α *)
let find p t =
  let rec find_aux = function
      | Leaf x → if p x then Some x else None
      | Node (l, r) → match find_aux p l with
        | None → find_aux p r
        | x_opt → x_opt
  in
  match find_aux p t with
      | None → raise Not_found
      | Some x → x
```

What About Searching a Tree?

Using Continuations

$$\textbf{type}\ \alpha\ \text{tree} = \text{Leaf}\ \textbf{of}\ \alpha \mid \text{Node}\ \textbf{of}\ \alpha\ \text{tree} \times \alpha\ \text{tree}$$

```
(* find : (α → bool) → α tree → α *)
let find p =
  let rec find_aux k = function
      | Leaf x → if p x then x else k ()
      | Node (l, r) → find_aux (fun _ → find_aux k r) l
  in
  find_aux (fun _ → raise Not_found)
```

Return