wxHaskell for the web

Substituting C++ for Haskell and JavaScript

Ruben Alexander de Gooijer

[*Supervisors*] Atze Dijkstra and Doaitse Swierstra
Utrecht University Department of Computing Science
October 4th, 2012

Type-safe web applications

Haskell on the server-side: Snap, Yesod, HappStack.

The Utrecht Haskell Compiler JavaScript back-end gives us Haskell on the client.

Haskell on the server and client equals profit!

- Automatic data type consistency
  no more mapping problems
- Code sharing
  no more duplication of validation, business rules, etc.
- No JavaScript!
  JavaScript as assembly language

O GUI toolkit, Where Art Thou?

For client-side development we need a GUI toolkit.

It makes sense to reuse an existing approach:

- Gtk2Hs (Linux)
- wxHaskell (multi-platform)

Both interface to **foreign** GUI toolkits, but neither run in the web browser.

Research Question

How can wxHaskell be made to run in the web browser?

We claim to have answered our research question through the implementation of a subset of wxHaskell that runs in the web browser.

As a case study we made it possible to run a feature-light version of wxAsteroids on the desktop as well as in the web browser.

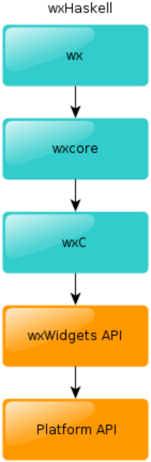
Why wxHaskell?

- Structured into multiple layers
  Will make porting easier
- Abstractions
  More declarative programming-style compared to Gtk2Hs
- Already has multi-platform support
  The web as yet another platform

History in Utrecht

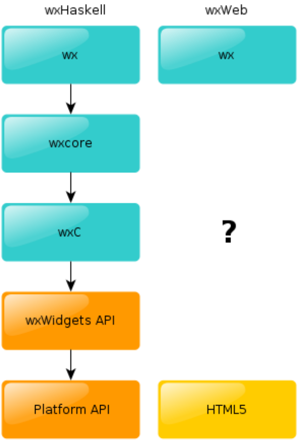
Daan Leijen, the original author of wxHaskell is a former UU PhD

wxHaskell Architecture



- wx
  abstractions for parts of wxCore
- wxCore
  a Haskell interface to wxWidgets using wxC
- wxC
  automatically generated Foreign Function Interface (FFI) bindings to wxWidgets
- wxWidgets
  an API for cross a platform GUI toolkit
- Platform API
  native GUI libraries: GTK, Cocoa, Windows

wxHaskell for the web



What do we want?

- Easily transfer existing wxHaskell programs to the web
- A portable solution
- The benefits of Haskell
  type-safety, compiler optimizations...

We cannot reuse wxWidgets!

- Reimplement it in JavaScript?
- Or in Haskell?

Blue: Haskell, Orange: C++, Yellow: JavaScript

Solution

Implement the wxcore API in Haskell:

Plan of attack:

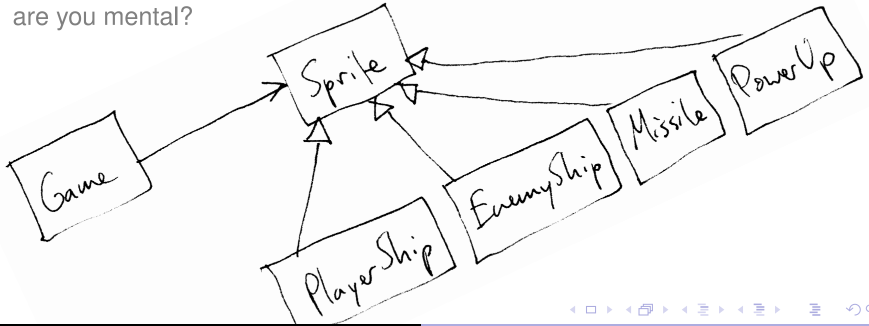
- Create a library for OO programming in Haskell
- Use it to implement the wxcore API (OO design)
- Interface to the web browser using the JavaScript FFI

$$\textit{foreign import js}\ \texttt{"\%1.alert(\%2)"}$$
$$\quad alert :: Window \rightarrow JSString \rightarrow IO\ ()$$

- Compile everything with UHC to JavaScript.

Object-Oriented programming in Haskell

are you mental?



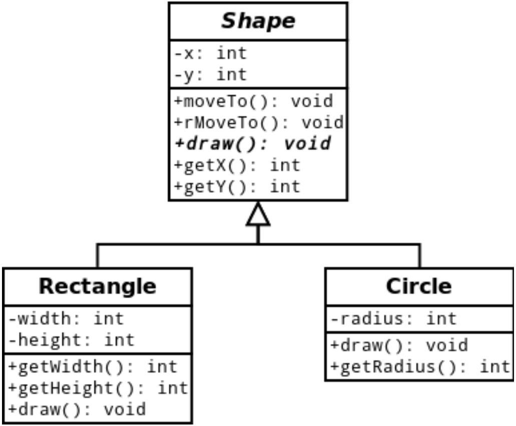
Introduction

The library is based on the ”Mutable Objects, with tail-polymorphism” approach described in the OOHaskell paper.

We have extended their approach with:

- A proper implementation of inheritance
- Generic functions for casting
- Parameterized classes
- Macros for deriving parts of the boilerplate

Introduction



The Shapes Benchmark

A benchmark for testing a language's ability to express:

- Data encapsulation
- Inheritance
- Subtype polymorphism
- Abstract methods

Introduction

Object encoding:

- Objects as (plain) records of closures
  - record selectors → method implementations
  - closure → data encapsulation

Combining records:

- Type extension through tail-polymorphism
  - poorman's approach to extensible records
  - type parameter represents a record extension (the tail)
  - special selector for manipulating the tail

Object Types

```
data IShape α = IShape {
    _getX     :: IO Int
  , _getY     :: IO Int
  , _setX     :: Int → IO ()
  , _setY     :: Int → IO ()
  , _moveTo   :: Int → Int → IO ()
  , _draw     :: IO ()
  , _shapeTail :: α
}
```

```
data IRectangle α = IRectangle {
    _getWidth      :: IO Int
  , _getHeight     :: IO Int
  , _setWidth      :: Int → IO ()
  , _setHeight     :: Int → IO ()
  , _rectangleTail :: α
}
```

```
data ICircle α = ...
```

Record Combination

Creating a rectangle out of *IShape* and *IRectangle*.

$$
\begin{aligned}
\mathit{IShape}\ \{\ & \_\mathit{setX} = ... \\
,\ & \_\mathit{setY} = ... \\
,\ & \_\mathit{moveTo} = ... \\
,\ & \_\mathit{draw} = ... \\
,\ & \_\mathit{shapeTail} = ? \\
\}\ &
\end{aligned}
$$

Record Combination

$$
\begin{array}{l}
IShape\ \{\ \_setX \quad = \ldots \\
\qquad ,\ \_setY \quad = \ldots \\
\qquad ,\ \_moveTo \quad = \ldots \\
\qquad ,\ \_draw \quad = \ldots \\
\qquad ,\ \_shapeTail = IRectangle\ \{\ \_getWidth \quad = \ldots \\
\qquad\qquad\qquad\qquad ,\ \_getHeight \quad = \ldots \\
\qquad\qquad\qquad\qquad ,\ \_setWidth \quad = \ldots \\
\qquad\qquad\qquad\qquad ,\ \_setHeight \quad = \ldots \\
\qquad\qquad\qquad\qquad ,\ \_rectangleTail = () \\
\qquad\qquad\qquad\qquad \} \\
\qquad \} :: IShape\ (IRectangle\ ())
\end{array}
$$

Subtype Polymorphism

Subtype polymorphism by quantifying over the tail:

$\textit{IShape}\ \alpha$

$\textit{IShape}\ ()$

$\textit{IShape}\ (\textit{IRectangle}\ ())$ $\quad$ $\textit{IShape}\ (\textit{ICircle}\ ())$

Foo takes at least a Shape:

$$\textit{foo} :: \textit{IShape}\ \alpha \rightarrow ...$$

Restriction: polymorphic object types can only be used at the contravariant (consuming) position.

Method Lookup

Method implementation:

$$\mathit{\_getWidth} :: \mathit{IRectangle}\ \alpha \rightarrow \mathit{IO}\ \mathit{Int}$$

Method lookup:

$$\begin{aligned}&\mathit{getWidth} :: \mathit{IShape}\ (\mathit{IRectangle}\ \alpha) \rightarrow \mathit{IO}\ \mathit{Int}\\&\mathit{getWidth} = \mathit{\_getWidth} \circ \mathit{\_shapeTail}\end{aligned}$$

Implementation

- class implementation → function
- mutable variables → *IORef*

```
shape newx newy concreteDraw = do
  x ← newIORef newx
  y ← newIORef newy
  return IShape {
        _getX    = readIORef x
      , _getY    = readIORef y
      , _setX    = writeIORef x
      , _setY    = writeIORef y
      , _moveTo = λnewx newy → do
           ??
      ...
  }
```

Self-reference

- invoke methods of the same object

```
shape newx newy concreteDraw self = do
  ...
  return IShape {
    ...
    , _moveTo = λnewx newy → do
        setX self newx
        setY self newy
    , _draw = concreteDraw self
    ...
  }
```

Record Extension

tail is a computation that results in the extension of the record.

$$
\begin{array}{l}
\mathit{shape}\ \mathit{newx}\ \mathit{newy}\ \mathit{concreteDraw}\ \mathit{tail}\ \mathit{self} = \mathbf{do} \\
\quad \ldots \\
\quad t \leftarrow \mathit{tail} \\
\quad \mathit{return}\ \mathit{IShape}\ \{ \\
\qquad \ldots \\
\qquad , \_\mathit{shapeTail} = t \\
\quad \}
\end{array}
$$

Let's see the type

Definition

A class is a function that provides an implementation for an object.

$$\textit{type Class tail self obj} = \textit{IO tail} \rightarrow \textit{self} \rightarrow \textit{IO obj}$$

In the case of *shape*, *self* and *obj* are specialized to at least a Shape:

$$\textit{shape}\ 2\ 3\ (\lambda\_ \rightarrow \textit{print}\ 42) :: \textit{Class tail}\ (\textit{IShape}\ \alpha)\ (\textit{IShape tail})$$

Instantiation

We instantiate a class by:

- closing record extension
- and providing a self-reference

Closing record extension with the empty record:

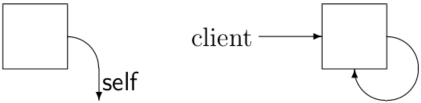
$$\mathit{emptyRecord} :: \mathit{IO}\ ()$$
$$\mathit{emptyRecord} = \mathit{return}\ ()$$

Apply *shape* to *emptyRecord*:

$$\mathit{shape}\ 2\ 3\ \mathit{drawImpl}\ \mathit{emptyRecord} :: \mathit{IShape}\ \alpha \rightarrow \mathit{IO}\ (\mathit{IShape}\ ())$$

Fixing the self-reference

Before instantiation *self* is unbound:



We connect *self* back to the class by taking its fixed point:

$$fixIO :: (o \rightarrow IO\ o) \rightarrow IO\ o$$

Apply fixIO to *shape*:

$$fixIO\ \$\ shape\ 2\ 3\ drawImpl\ emptyRecord :: IO\ (IShape\ ())$$

The new combinator

The *new* combinator takes a class and instantiates it.

$$new\ c = fixIO\ \$\ c\ emptyRecord$$

E.g.

$$\begin{aligned}&do\ let\ drawImpl\ \_ = print\ 42\\&\quad s \leftarrow new\ \$\ shape\ 2\ 3\ drawImpl\\&\quad draw\ s\end{aligned}$$

Inheritance

What is inheritance?

A technique for sharing behavior between objects.

How?

By subclassing, i.e. the incremental extension of classes.

Requirements:

- The self-reference should be late-bound
  method invocations made by a superclass can be intercepted by a subclass.
- The addition of methods and data to classes possibly overriding existing methods
- A subclass can invoke methods on the superclass

Cooking up an inheritance combinator

William R. Cook, a denotational semantics of inheritance:

- A combinator for inheritance using records in the untyped lambda calculus

$$W \boxed{\triangleright} G = \lambda \mathit{self} \rightarrow W\ (G\ \mathit{self})\ \mathit{self}\ \oplus\ (G\ \mathit{self})$$

Cooking up an inheritance combinator

William R. Cook, a denotational semantics of inheritance:

- A combinator for inheritance using records in the untyped lambda calculus

$$W \boxed{\triangleright} G = \quad \lambda self \to \underbrace{W\,(\overbrace{G\ self}^{\text{SUPER}})\ self}_{\text{SUB}} \oplus (\overline{G\ self})$$

combine

Implementing Rectangle

```
rectangle x y width height =
  (rectImpl `extends` shape x y draw) noOverride set_Shape_Tail
  where
  rectImpl tail super self = do
    ...
    return IRectangle {
      ...
    }
```

A small OOP example

$$
\begin{aligned}
&myOOP = \mathbf{do} \\
&\quad s_1 \leftarrow new\ \$\ rectangle\ 10\ 20\ 5\ 6 \\
&\quad s_2 \leftarrow new\ \$\ circle\ 15\ 25\ 8 \\
&\quad \mathbf{let}\ scribble :: [?] \\
&\quad\quad\quad scribble = ? \\
&\quad mapM\_\ draw\ scribble
\end{aligned}
$$

A small OOP example

What do we put at the question mark?

A small OOP example

What do we put at the question mark?

$[s_1, s_2]$ -- *Type error*

A small OOP example

What do we put at the question mark?

$[s_1, s_2]$ -- *Type error*

Explicitely tag the values?

$$scribble :: [\mathit{Either}\ (\mathit{IShape}\ (\mathit{IRectangle}\ ()))\ (\mathit{IShape}\ (\mathit{ICircle}\ ()))]$$
$$scribble = [\mathit{Left}\ s_1, \mathit{Right}\ s_2]$$

A small OOP example

Or existentially quantify over the tail?

$$scribble :: [\exists a.IShape\ a]$$
$$scribble = [s_1, s_2]$$

But... we cannot recover from the lost type information.

A small OOP example

Instead we implement two generic casting functions:

$$\begin{aligned} \textit{upcast} &:: \alpha \rightarrow \beta \\ \textit{downcast} &:: \beta \rightarrow \textit{Maybe}\ \alpha \end{aligned}$$

Provided with a source and target, related by subtyping, they generate a proof that the two types can in principle be converted to each other.

Finally...

$$
\begin{aligned}
scribble &:: [\,IShape\ ()\,] \\
scribble &= [\,upcast\ s_1, upcast\ s_2\,]
\end{aligned}
$$

A glimpse at the internals

- *upcast* and *downcast* implemented using a type class
- Instances provide a syntax directed encoding of refl. and trans. of the subtyping relation by pattern matching on the type structure
- Changing the type of a record's tail to:

$$\textit{type Record } \alpha = \textit{Either } \alpha \textit{ Dynamic}$$

- Interplay between *upcast* and *downcast* using dynamic typing to leave a trace to the original type.

A glimpse at the internals

$$\textbf{class}\ a \prec b\ \textbf{where}\quad upcast :: a \rightarrow b$$

$$\textbf{class}\ a \succ b\ \textbf{where}\quad downcast :: a \rightarrow Maybe\ b$$

$$\text{-- Reflexivity}$$

$$\textbf{instance}\ a\ () \prec a\ ()\ \textbf{where}\quad upcast = id$$

$$\textbf{instance}\ a\ () \succ a\ ()\ \textbf{where}\quad downcast = Just$$

$$\text{-- Transitivity}$$

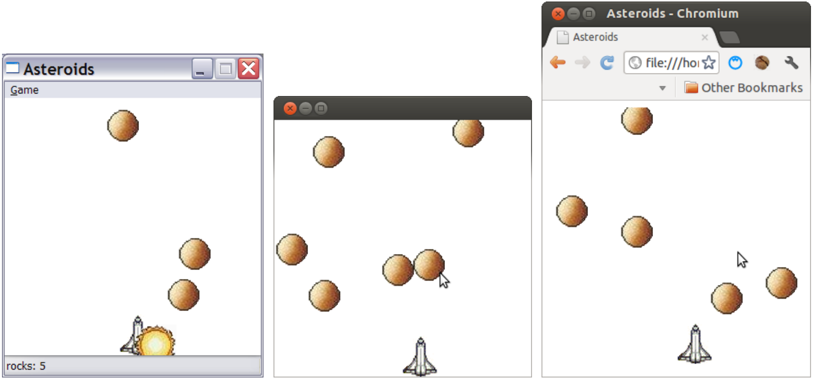
$$\textbf{instance}\ (a\ () \prec x, Narrow\ (a\ (b\ ()))\ (a\ ())) \Rightarrow a\ (b\ ()) \prec x$$

$$\textbf{instance}\ (a\ (b\ ()) \succ a\ (b\ c), Widen\ (a\ ())\ (a\ (b\ ()))) \Rightarrow a\ () \succ a\ (b\ c)$$

Porting wxHaskell to the web browser

a case study wxAsteroids

After some hard work...



Original ⟶ Simplified ⟶ In the browser.

Implementation details

- wxHaskell uses phantom types to model a type-safe interface to C++ objects.
- Our object encoding follows the exact same type structure, hence we simply replace the pointer structure with objects implemented in Haskell.
- At some places we needed to make the wxcore interface less polymorph such that we could still use casting.

Implementation details

- Cyclic type dependencies break organizational scheme: a module per interface and class.
- We choose a straight-forward mapping from wxWidgets abstractions to HTML5.
    - wxWindow → HTML DIV
    - Drawing Context → HTML Canvas
    - Events → native events wrapped in wxWidgets event objects
    - Timer → setInterval

Conclusion

wxHaskell can be made to run in the web browser.

However,

- Different forms of subtyping and the use of plain records makes the library not particularly easy to use.
- The lack of recursive modules will make any attempt at OO programming in Haskell feel clumsy.
- It remains unclear if wxWidgets provides the right abstractions for the web platform.

Questions?

OO library:
`https://github.com/rubendg/lightoo`

JS prelude:
`https://github.com/rubendg/uhc-js`

wxAsteroids:
`https://github.com/rubendg/wxasteroids`

Related work

- wxFlashkell: Building Flash based GUI's in Haskell
- GHCJS, Haste, YHC, Fay

Future Work

- A translation from Featherweight Java to our OO library?
- Automated type checking based on FFI types
- Reuse *Data*.*Dynamic* for type checking? Building a JavaScript *TypeRep*?
- A more granular model for JavaScript types in Haskell, structural types?

A combinator for inheritance

$$
\begin{array}{l}
\mathit{extends}\ w\ g\ \mathit{override}\ \oplus = \mathit{clazz}\ \$\ \lambda \mathit{tail}\ \mathit{self} \rightarrow \mathbf{do} \\
\quad \text{-- instantiate super} \\
\quad \mathit{super} \ \ \leftarrow g\ \mathit{emptyRecord}\ \mathit{self} \\
\quad \text{-- provide the subclass with super and self} \\
\quad \mathit{sub} \quad\ \ \leftarrow w\ \mathit{tail}\ \mathit{super}\ \mathit{self} \\
\quad \text{-- possibly override methods} \\
\quad \mathit{super}' \leftarrow \mathit{override}\ \mathit{super}\ \mathit{self} \\
\quad \text{-- combine the modified super and subclass records} \\
\quad \mathit{return}\ \$\ \mathit{super}' \oplus \mathit{sub}
\end{array}
$$