

Reconciling OO and Haskell-Style Overloading

Mark Shields

Simon Peyton Jones

Microsoft Research Cambridge

`{markshie, simonpj}@microsoft.com`

It's an OO World

The screenshot shows a web browser window titled ".NET Framework SDK Documentation". The navigation bar includes links: Hide, Locate, Back, Forward, Stop, Refresh, Home, Print, and Options. Below the navigation bar are tabs: Contents, Index, Search, and Favorites. The left sidebar displays a tree view of the .NET Framework Class Library, with "System.Drawing.Imaging" selected. The main content area is titled ".NET Framework Class Library" and "System.Drawing.Imaging Namespace". It contains a note about the topic's status, a link to send feedback, and a description of the namespace's functionality. Below this, a table lists classes in the namespace.

.NET Framework SDK Documentation

Hide Locate Back Forward Stop Refresh Home Print Options

Contents Index Search Favorites

.NET Framework Reference
 .NET Framework Class Library
 Microsoft.CSharp
 Microsoft.JScript
 Microsoft.VisualBasic
 Microsoft.Win32
 System
 System.CodeDom
 System.CodeDom.Compiler
 System.Collections
 System.Collections.Specialized
 System.ComponentModel
 System.ComponentModel.Design
 System.ComponentModel.Design.Serialization
 System.Configuration.Assemblies
 System.Configuration.Install
 System.Data
 System.Data.Common
 System.Data.OleDb
 System.Data.SqlClient
 System.Data.SqlTypes
 System.Diagnostics
 System.Diagnostics.SymbolStore
 System.DirectoryServices
 System.Drawing
 System.Drawing.Design
 System.Drawing.Drawing2D
 System.Drawing.Imaging
 System.Drawing.Printing
 System.Drawing.Text
 System.EnterpriseServices

.NET Framework Class Library
System.Drawing.Imaging Namespace

Note: This topic is not complete for RTM, nor has it been handed off to localization. Enter bugs for inaccurate, not incomplete, information using the following RAID [template](#).

This is preliminary documentation and subject to change.
[Send feedback](#) on this topic.

The **System.Drawing.Imaging** namespace provides advanced GDI+ imaging functionality. Basic graphics functionality is provided by the [System.Drawing](#) namespace.

The [Metafile](#) class provides methods for recording and saving metafiles. The [Encoder](#) and Decoder classes enable users to extend GDI+ to support any image format. The [PropertyItem](#) class provides methods for storing and retrieving metadata in image files.

Classes

Class	Description
ApmFileHeader	Defines an Aldus Placetable Metafile.
BitmapData	Specifies the attributes of a bitmap image.
ColorMap	Defines a map for converting colors.
ColorMatrix	Defines a 5 x 5 matrix that contains the homogenous coordinates for the RGBA space.
ColorPalette	Defines an array of colors that make up a color palette.

Basic FFI

C#	Haskell
<pre>class A { static int m(int, char); }</pre>	<pre>m :: Int -> Char -> IO Int</pre>

- Marshal C# base types to/from Haskell types
- Design decisions yet to make:
 - Unboxing? (Probably ignore)
 - Byrefs? (Probably copy in, copy out)
 - Null Pointers? (Probably use `Maybe`)
 - Purity? Non-null? (Need standard C# type annotations)

Novelty 1: Subtyping

C#	Haskell
<pre>class A { int m(char); } class B : A, I { ... }</pre>	<pre>m :: Char -> A -> IO Int</pre>

- `a :: A, b :: B`
- `a # m '1' :: IO Int`
- `b # m '1' :: IO Int`

(#) :: a -> (a -> b) -> b (depending on this talk. In a derived class.)

Novelty 2: Overloading

C#	Haskell
<pre>class A { int m(char); int m(int); }</pre>	<pre>m :: Char -> A -> IO Int m :: Int -> A -> IO Int</pre>

- `a :: A`
- `a # m '1' :: IO Int` `-- call first m`
- `a # m 1 :: IO Int` `-- call second m`
- (This doesn't seem so bad, give Haskell already has type-based overloading)

Novelty 3: "Overlapping" Methods

C#	Haskell
<pre>class C { ... } class D : C { ... } class A { int m(C); int m(D); }</pre>	<pre>m :: C -> A -> IO Int m :: D -> A -> IO Int</pre>

`a :: A, c :: C, d :: D`

`a # m c :: IO Int -- call first m`

`a # m d :: IO Int -- call second m`

- (Since both `m`'s match second call, this seems well outside of Haskell's system)

Objectives

- Account for these novelties in a way which:
 - is as natural as *C#*: no name mangling or explicit coercions
 - blends with existing constrained polymorphism and class constraints framework
- Non objectives:
 - Extend Haskell to be a "calculus of objects"
 - in particular, internalizing virtual methods
 - be as natural as *C#* for constructing new classes
 - (The MLj approach would probably be the best were these to be the objectives)

Constrained Polymorphism in 1 Slide

- **Classes:** `class D a => C a where a -> a`
- **Constraints:** `C τ`
true if type τ is a member of type class `C`
- **Witnesses:** `instance C Int = t`
`C Int` is true, and witnessed by the pair
 (W, t)
where W is a witness for `D Int`
and $t :: \text{Int} \rightarrow \text{Int}$
- Type inference propagates constraints at compile-time
- Dictionary transformation causes witnesses to be propagated at run-time

Obvious Non-starter

- Can't just encode C# class as Haskell class:
 - Will reject C# class with overloaded methods
(Haskell method names must be unique within class)
 - Will reject C# classes sharing the same method name
(Haskell method names must be unique across all classes)
 - Too much witness passing:

```
f :: A a => a -> IO Int
```

Each call to f passes bogus dictionary of methods for
a

Subtyping: Attempt

- Simulate subtyping constraints using classes:

C#	Haskell
<pre>class A { int m(char); } class B : A, I { ... }</pre>	<pre>newtype A = ObjRef newtype B = ObjRef class BelowA a instance BelowA A instance BelowA B class (BelowA a, BelowI a) => BelowB a instance BelowB B m :: BelowA a => Char -> a -> IO Int</pre>

Subtyping: Attempt

- ☺ No witness passing required
- ☺ Works for multiple inheritance
- ☺ AboveA may be defined analogously
- ☺ Captures transitivity

```
m :: BelowA a => Char -> a -> IO Int
n :: BelowB a => Char -> a -> IO Int
f :: BelowB a => a -> IO Int
f x = do i <- x # m '1'
        j <- x # n '2'
        return i + j
```

Subtyping: Attempt

☹ Quadratic growth in instance declarations

☹ Almost no subtype simplifications possible

`(AboveA a, BelowA a) => a -> a`

instead of `A -> A`

`(AboveA a, BelowB a) => a -> a`

instead of type error

Subtyping: Solution

- Add a new subtype constraint

$t \leq u$ "type t is a subtype of type u "

- Subtype constraints only appear on

- imported methods

- `coerce :: a ≤ b => a -> b`

- (possibly) functions using the above

- Subtypes only introduced by `newtype` declarations

`newtype A ≤ B1, ..., Bn = ObjRef`

Subtyping: Solution

C#	Haskell
<pre>class A { int m(char); } class B : A, I { ... }</pre>	<pre>newtype A = ObjRef newtype B <: A, I = ObjRef m :: a <: A => Char -> a -> IO Int</pre>

- `a :: A, b :: B`
- `a # m '1' :: IO Int`
- `b # m '1' :: IO Int`
- `f :: a <: A => a -> IO Int`
`f x = x # m '1'`

Subtyping: Quirks

- For convenience, rules for structural subtyping builtin
- Subtyping is **not** implicit on every application
- Hence, maximization & minimization simplification rules not applicable
$$(A \text{ :<: } a, b \text{ :<: } B) \Rightarrow b \text{ -> } a$$
cannot be reduced to $B \text{ -> } A$
- We only assume newtypes form a poset, **not** a lattice or upwards-closed semi-lattice
- Hence, no \perp or \top types

Subtyping: Quirks

- The subtype poset is downward extensible
- Hence glb rules are **not** applicable

newtype A

newtype B

newtype C :<: A, B

$(a :<: A, a :<: B) \Rightarrow a \rightarrow a$

cannot be reduced to

$a :<: C \Rightarrow a \rightarrow a$

since another module may add

newtype D :<: A, B

- lub rules ok

Overloading: Attempt

- Encode OO overloading as degenerate Haskell type-class overloading

C#	Haskell
<pre>class A { int m(char); int m(int); }</pre>	<pre>class Has_m a where m :: a instance a <: A => Has_m (Char -> a -> IO Int) where m = {- call first m -} instance a <: A => Has_m (Int -> a -> IO Int) where m = {- call second m -}</pre>

Overloading: Attempt

☹ Type inference won't commit to return type:

```
a :: A
```

```
a # m '1' :: Has_m (Char -> A -> b) => b
```

☹ Type inference won't report error eagerly:

```
a # m True :: Has_m (Bool -> A -> b) => b
```

☹ Type inference won't commit to a method:

```
instance a :<: A =>
```

```
  Has_m ((Int, Int) -> a -> IO Int)
```

```
a # m (x,y) :: Has_m ((b,c) -> A -> d) => d
```

Overloading: Attempt

- These problems are a consequence of the **openness** of instance declarations:
 - simplifier must assume `Has_m` could be extended in another module
 - hence `Has_m` constraints will propagate within type schemes until become ground
- But, we want most/all method overloading to be resolved before generalization based on which methods are visible **now**

Overloading: Solution

- Introduce notion of `closed` class
- If simplifier sees a class constraint for a closed class, it may assume all instance declarations are in scope
- This allows more aggressive simplification: improvement and early failure

- `Has_m` classes are declared `closed`
- Constraint syntactic sugar

`m :: t`

for

`Has_m t`

Overloading: Solution

C#	Haskell
<pre>class A { int m(char); int m(int); }</pre>	<pre>class closed m :: a where m :: a instance a <: A => m :: Char -> a -> IO Int where m = {- call first m -} instance a <: A => m :: Int -> a -> IO Int where m = {- call second m -}</pre>

Overloading: Solution

☺ Only one method on `A` with `Char` argument possible, so commit:

```
a :: A
```

```
a # m '1' :: IO Int
```

☺ No methods on `A` with `Bool` argument possible, so reject:

```
a # m True
```

```
error: cannot satisfy m :: Bool -> A -> c
```

☺ Only one method on `A` with pair argument, so commit:

```
instance a :<: A =>
```

```
  m :: (Int, Int) -> a -> IO Int
```

```
a # m (x, y) :: IO Int
```

```
(and x :: Int, y :: Int)
```

Overloading: Solution

☺ A little more subtle: still don't know which method, but know result must be `IO Int`

`a :: A`

`f :: (m :: a -> A -> IO Int) =>
a -> IO Int`

`f x = a # m x`

- (Deferring method constraints seems reminiscent of virtual methods. We shall return to this)

Simplification for Closed Class Constraints

Given constraint $C \tau$ for closed class C :

- Step 1: Collect all "candidate" instances

$$\forall \alpha . \Phi \Rightarrow C \upsilon$$

such that there exists a θ such that

$$\theta [\alpha \mapsto \beta] (C \upsilon) = \theta (C \tau)$$

and $\theta [\alpha \mapsto \beta] \Phi$ (may be) satisfiable

- Step 2: Calculate least-common generalization, τ' , of all $\theta [\alpha \mapsto \beta] \upsilon$
- Step 3: Unify τ with τ'
- Step 4: If only one candidate, use it to construct witness for $C \tau'$

Overlapping Overloading: Attempt

- Overlapping methods become overlapping instances

C#	Haskell
<pre>class C { ... } class D : C { ... } class A { int m(C); int m(D); }</pre>	<pre>class closed m :: a where m :: a instance (a <:: A, b <:: C) => m :: b -> a -> IO Int instance (a <:: A, b <:: D) => m :: b -> a -> IO Int</pre>

Overlapping Overloading: Attempt

☹ Overlapping instances are illegal in standard Haskell

☹ GHC/Hugs support them, but (as usual for Haskell...) without any formal description, and inconsistently (even between versions)

☺ Could be made to work if only one candidate:

```
a :: A, c :: C
```

```
a # m c    -- only first m possible
```

☹ But still fails with multiple candidates:

```
a :: A, d :: D
```

```
a # m d :: (m :: D -> A -> IO Int) => IO Int
           -- should choose second m
```

Overlapping Overloading: Solution

- Classes declared as `closed` (or `overlap`) may have arbitrarily overlapping instances
- Solve class constraints as before, but if multiple candidates, may choose **least** under (an approximation of) the instantiation ordering \leq
- Eg:

$$\forall a\ b . (a :<: A, b :<: D) \Rightarrow m :: b \rightarrow a \rightarrow IO\ Int$$
$$\leq$$
$$\forall a\ b . (a :<: A, b :<: C) \Rightarrow m :: b \rightarrow a \rightarrow IO\ Int$$

since from $a :<: A, b :<: D$

we may derive $a :<: A, b :<: C$

Overlapping Overloading: Solution

- More formally, \leq is defined by entailment:

$$\frac{\forall \beta . \Xi \Rightarrow C \tau, [\alpha \rightarrow \alpha'] \Phi \vdash^e [\alpha \rightarrow \alpha'] C \upsilon}{\forall \alpha . \Phi \Rightarrow C \upsilon \leq \forall \beta . \Xi \Rightarrow C \tau}$$

- However, it is implemented by simplification:

$$\frac{\begin{array}{l} < \forall \beta . \Xi \Rightarrow C \tau, [\alpha \rightarrow \alpha'] \Phi, C \upsilon > \\ \rightarrow^* < \forall \beta . \Xi \Rightarrow C \tau, [\alpha \rightarrow \alpha'] \Phi > \end{array}}{\forall \alpha . \Phi \Rightarrow C \upsilon \leq \forall \beta . \Xi \Rightarrow C \tau}$$

Overlapping Overloading: Incompleteness

- Constraint simplifier is deliberately incomplete w.r.t. constraint entailment for overlapping instances:

- Only consider **one** class constraint at a time:

```
instance m :: Int -> Int
instance m :: Bool -> Int
instance n :: Int -> Int
instance n :: Char -> Int
f x = m x + n x
```

Valid typing is $f :: \text{Int} \rightarrow \text{Int}$

But inferred type is

```
f :: (m :: a -> Int, n :: a -> Int) =>
      a -> Int
```

Overlapping Overloading: Incompleteness

- Don't fully exploit unsatisfiability

```
class closed C a where ...
```

```
class closed D a where ...
```

```
instance C Int
```

```
instance C a => m :: a -> IO Int
```

```
instance D a => m :: a -> IO Int
```

```
f x = m x
```

Valid typing is `f :: Int -> IO Int`

But inferred type is

```
f :: (m :: a -> IO Int) => a -> IO Int
```

Virtual Methods vs Member Constraints

C#

```
class A {  
    virtual int m();  
}  
class B : A {  
    virtual int m();  
}  
int f(A, A);  
f (x, y) {  
    int i, j;  
  
}
```

Call m passed as
implicit argument

Haskell

```
newtype A  
newtype B <:: A  
instance m :: A -> IO Int  
instance m :: B -> IO Int
```

Dispatch based on
dynamic type of x

```
x y = do i <- x # m  
         j <- y # m  
         return i + j
```

=>

Int

Virtual Methods vs Member Constraints

- Can we simulate OO references using existentials? Not quite:

```
data Sub a = forall b . b ::<: a => Sub b
```

```
newtype A
```

```
newtype B ::<: A
```

```
instance m :: A -> IO Int
```

```
instance m :: B -> IO Int
```

```
f = do a :: A <- new
```

```
      b :: B <- new
```

```
      list :: Sub A = [Sub a, Sub b]
```

```
      return (map (\Sub a -> m a) list)
```

error: constraint `m :: a -> b` escapes scope
of existentially quantified variable `a`

But, as stands, system
cannot determine `m`
from subtype
constraint passed
captured in `Sub`

Virtual Methods vs Member Constraints

- Upshot: it rarely makes sense for a method constraint to escape into a type scheme
- If class `C` has `global` modifier, it is an error for `C τ` to appear in a scheme

```
class global closed m :: a
```

```
instance m :: A -> IO Int
```

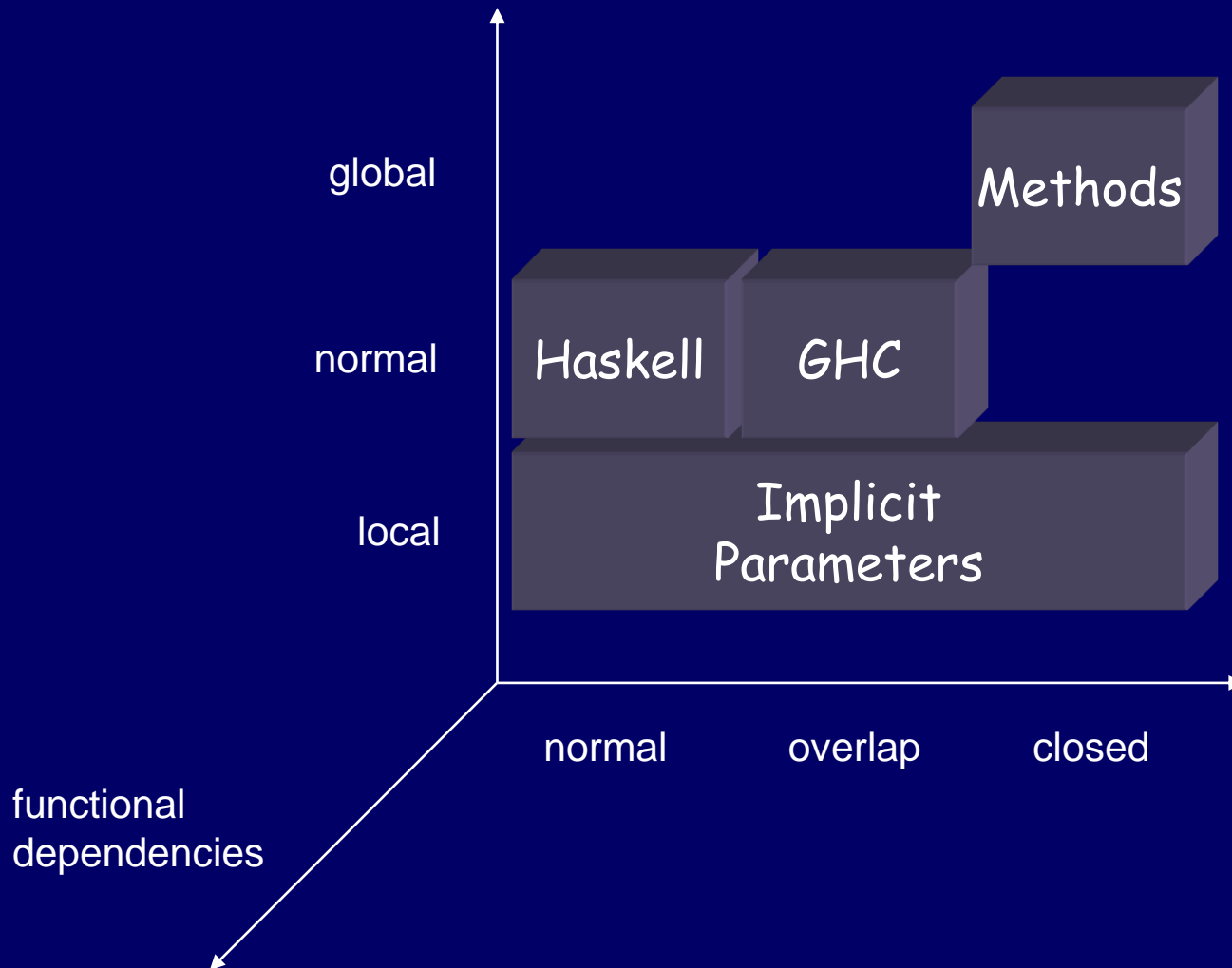
```
instance m :: B -> IO Int
```

```
f x = x # m
```

```
error: constraint m :: a -> b cannot  
       be generalized
```

Type Class Space

- Modifiers aren't as ad-hoc as may first appear:



Future Work

- About (finally...) to submit to Haskell or Babel Workshop
- Ulterior motive is to establish standard "calculus for type systems" including all the bells and whistles used in practice
- Simplifier defined quite abstractly...
 - ... still work to be done in refining it to a good algorithm
 - ... subtyping responsible for over half of complexity of simplifier
- Many small-scale FFI decisions to be made
- Simon & co about to start on implementation
- No formal properties shown as yet
- What about exporting Haskell as a .NET class?
- Can we further exploit nominal subtyping within Haskell?

