Generics for the Working ML'er

Vesa Karvonen
University of Helsinki
Why Generics?

• An innocent looking example:

```lisp
unitTests
  (title "Reverse")
  (testAll (sq (list int))
    (fn (xs, ys) ⇒
      thatEq (list int)
        {expect = rev (xs @ ys),
         actual = rev xs @ rev ys})))
```

Why Generics?
1. Reverse test
   FAILED:
   with ([521], [7])
   equality test failed:
   expected [7, 521], but got [521, 7].
Hidden Complexity

• Uses quite a few generics:
  – Arbitrary – to generate counterexamples
  – Shrink – to shrink counterexamples
  – Size – to order counterexamples by size ...
  – Ord – ... and an arbitrary linear ordering
  – Eq – to compare for equality
  – Pretty – to pretty print counterexamples
  – Hash – used by several other generics
  – TypeHash – used by Hash (and Pickle)
  – TypeInfo – used by several other generics

• Imagine having to write all those functions by hand to state the property...
Generics?

- A generic can be used at many types:
  \[ \alpha \times \alpha \rightarrow \text{Bool.t} \]
  \[ \alpha \rightarrow \text{String.t} \]

- Values indexed by one or more types

- Question: What is the relation to ad-hoc polymorphism?

- Problem: Types in H-M are implicit
Generics vs Ad-Hoc Poly.

**Generics**
- aka “Polytypic”, “Closed T-I ...”, ...
- Defined once and for all
  - O(1)
- Structural
- Inflexible
- Abstract

**Ad-Hoc Poly.**
- aka “Overloaded”, “Open T-I ...”, ...
- Specialized for each type (con)
  - O(n)
- Nominal
- Flexible
- Concrete
Encoding Types as Values

Value-Dependent

- Witness the value
  \[ \alpha \times \alpha \rightarrow \text{Bool}.t \]
  \[ \alpha \rightarrow \text{String}.t \]
- Hard to compose
- Easy to specialize
- Vanilla H-M

Value-Independent

- Witness the type
  \[ \alpha \leftrightarrow \mathbb{U} \]
- Easy to compose
- Hard to specialize
- GADTs, Existentials, Universal Type

eq : \( \alpha \) Eq.t \( \rightarrow \alpha \times \alpha \rightarrow \text{Bool}.t \)

show : \( \alpha \) Show.t \( \rightarrow \alpha \rightarrow \text{String}.t \)
The Approach in a Nutshell

- Use a value-dependent encoding to allow specialization
- Encode user defined types via sums-of-products and witnessing isomorphisms
- Close relative of Hinze's GM approach
- Encode recursive types using a type-indexed fixed point combinator
- Make type reps open-products to address composability
So, in Practice...

- For each type, the user must provide a type representation constructor (an encoding of the type constructor).
  - This could even be mostly automated.

- As a benefit, the user then gets a bunch of generic utility functions to operate on the type.

- So, instead of $O(mn)$ definitions, only $O(m+n)$ are needed!
Encoding Types

signature CLOSED_REP = sig type \( \alpha \) t and \( \alpha \) s and (\( \alpha \), \( \kappa \)) p end

signature CLOSED_CASES = sig

structure Rep : CLOSED_REP

val iso : \( \beta \) Rep.t \( \rightarrow \) (\( \alpha \), \( \beta \)) Iso.t \( \rightarrow \) \( \alpha \) Rep.t

val \( \otimes \) : (\( \alpha \), \( \kappa \)) Rep.p \( \times \) (\( \beta \), \( \kappa \)) Rep.p \( \rightarrow \) ((\( \alpha \), \( \beta \)) Product.t, \( \kappa \)) Rep.p

val T : \( \alpha \) Rep.t \( \rightarrow \) (\( \alpha \), Generics.Tuple.t) Rep.p

val R : Generics.Label.t \( \rightarrow \) \( \alpha \) Rep.t \( \rightarrow \) (\( \alpha \), Generics.Record.t) Rep.p

val tuple : (\( \alpha \), Generics.Tuple.t) Rep.p \( \rightarrow \) \( \alpha \) Rep.t

val record : (\( \alpha \), Generics.Record.t) Rep.p \( \rightarrow \) \( \alpha \) Rep.t

val \( \oplus \) : \( \alpha \) Rep.s \( \times \) \( \beta \) Rep.s \( \rightarrow \) ((\( \alpha \), \( \beta \)) Sum.t) Rep.s

val C0 : Generics.Con.t \( \rightarrow \) Unit.t Rep.s

val C1 : Generics.Con.t \( \rightarrow \) \( \alpha \) Rep.t \( \rightarrow \) \( \alpha \) Rep.s

val data : \( \alpha \) Rep.s \( \rightarrow \) \( \alpha \) Rep.t

val Y : \( \alpha \) Rep.t Tie.t

val \( \rightarrow \) : \( \alpha \) Rep.t \( \times \) \( \beta \) Rep.t \( \rightarrow \) (\( \alpha \) \( \rightarrow \) \( \beta \)) Rep.t

val refc : \( \alpha \) Rep.t \( \rightarrow \) \( \alpha \) Ref.t Rep.t

(* ... *)
**Binary Tree**

```ocaml
datatype α bt =  
  LF  
  | BR of α bt × α × α bt
val bt : α Rep.t → α t Rep.t =  
  fn a ⇒  
    fix Y (fn t ⇒  
      iso (data (C0 (C"LF") ⊕  
                  C1 (C"BR")  
                    (tuple (T t ⊗ T a ⊗ T t))))))  
  (fn LF ⇒ INL ()  
   | BR (a,b,c) ⇒ INR (a&b&c),  
   fn INL () ⇒ LF  
   | INR (a&b&c) ⇒ BR (a,b,c)))
val intBt : Int.t bt Rep.t = bt int
```
The Catch

- Recall that a value-dependent encoding makes it harder to combine generics
  - The type rep needs to be a product of all the generic values that you want [Yang]

- So, we use an open product for the type rep [Berthomieu] and use open structural cases

- A generic is implemented as a functor for extending a given (existing) combination

- But you still need to explicitly define the combination that you want and close it (non-destructively) for use
signature EQ = sig
  structure EqRep : OPEN_REP
  val eq : (α, χ) EqRep.t → α BinPr.t
  val notEq : (α, χ) EqRep.t → α BinPr.t
  val withEq : α BinPr.t → (α, χ) EqRep.t UnOp.t
end
signature EQ_CASES = sig
  include CASES EQ
  sharing Open.Rep = EqRep
end
signature WITH_EQ_DOM = CASES
functor WithEq (Arg : WITH_EQ_DOM) : EQ_CASES
signature HASH = sig
  structure HashRep : OPEN_REP
  val hashParam : (α, χ) HashRep.t → {totWidth : Int.t,
                                   maxDepth : Int.t} → α → Word.t

  val hash : (α, χ) HashRep.t → α → Word.t
end
signature HASH_CASES = sig
  include CASES HASH
  sharing Open.Rep = HashRep
end
signature WITH_HASH_DOM = sig
  include CASES TYPE_HASH TYPE_INFO
  sharing Open.Rep = TypeHashRep = TypeInfoRep
end
functor WithHash (Arg : WITH_HASH_DOM) : HASH_CASES
Extending a Composition

- **Root generic** ($(G)/with/generic.sml)
  
  ```sml
  structure Generic = struct
  structure Open = RootGeneric
  end
  ```

- **Equality** ($(G)/with/eq.sml)
  
  ```sml
  structure Generic = struct
  structure Open = WithEq (Generic)
  open Generic Open
  end
  ```

- **Hash** ($(G)/with/hash.sml)
  
  ```sml
  structure Generic = struct
  structure Open = WithHash
  (open Generic
   structure TypeHashRep = Open.Rep and TypeInfoRep = Open.Rep)
  open Generic Open
  end
  ```
Defining a Composition

• With the ML Basis System:

```ml
local
  $(G)/lib.mlb
  $(G)/with/generic.sml
  $(G)/with/eq.sml
  $(G)/with/type-hash.sml
  $(G)/with/type-info.sml
  $(G)/with/hash.sml
  $(G)/with/ord.sml
  $(G)/with/pretty.sml
  $(G)/with/close.pretty-with-extra.sml
in
  my-program.sml
end
```
Algorithmic Details Matter

• Generic algorithms:
  – must terminate on recursive types
  – must terminate on cyclic data structures
  – must respect identities of mutable objects
  – should avoid unnecessary computation
  – should be competitive with handcrafted algorithms

• The Eq generic (example in the paper) is easy only because SML's equality already does the right thing!
One of the simplest generics

But, there is a catch

At a sum, which direction do you choose, left or right?

One solution is to analyze the type...

```haskell
fun a ⊕ b = case hasBaseCase a & hasBaseCase b
  of true & false ⇒ INL o getS a
  | false & true ⇒ INR o getS b
  | _ ⇒ ...
```
Does it Have a Base Case?

id \ T = \top

fix \ \lambda t

iso

data

\top \ C0 (C''LF''') \ T \lor \bot = \top

C1 (C''BR''')

tuple

id \bot = \bot

\bot \land \bot = \bot

\bot \land \top = \bot

int \ T

\top \lor \bot = \top

\bot \land \top = \bot

t \bot
val pretty : (α, χ) PrettyRep.t → α → Prettier.t

- Features:
  - Uses Wadler's combinators
  - Output mostly in SML syntax
  - Doesn't produce unnecessary parentheses
  - Formatting options (ints, words, reals)
  - Optionally shows only partial value
  - Shows sharing of mutable objects
  - Handles cyclic data structures
  - Supports infix constructors
  - Supports customization
The Library

- Provides the framework (signatures, layering functors) and
- several generics (17+) from which to choose
- Most of the generics have been implemented quite carefully
- Available from MLton's repository
- MLton license (a BSD-style license)
In the Paper

- Implementation techniques
  - Sum-of-Products encoding
  - Type-indexed fixpoint combinator
  - Layering functors

- Discussion about the design

- NOTE: Some of the signatures have changed (for the better) after writing the paper, but the basic techniques are essentially same
Conclusion

• Works in plain SML'97

• Allows you to define generics both independently and incrementally and combine later for convenient use

• And I dare say the technique is reasonably convenient to use – definitely preferable to writing all those utilities by hand
Shopping List

- Definitely:
  - First-class polymorphism
  - Existentials
  - In the core language!

- Maybe:
  - Deriving
  - Type classes – well, something much better

- Wishful:
  - Lightweight syntax
    - `let open DSL in ... end vs (open DSL ; ...)`
Pickle

\[
\text{val pickle} : (\alpha, \chi) \text{PickleRep.t} \rightarrow \alpha \rightarrow \text{String.t}
\]
\[
\text{val unpickle} : (\alpha, \chi) \text{PickleRep.t} \rightarrow \text{String.t} \rightarrow \alpha
\]

- **Highlights:**
  - Platform independent and compact pickles
    - Tag size depends on type
    - Introduces sharing automatically
  - Handles cyclic data structures
  - Actually uses 6 other generics
    - Some & DataRecInfo
    - Eq & Hash
    - TypeHash
    - TypeInfo
List of Generics

- Arbitrary
- DataRecInfo
- [Debug]
- Dynamic
- Eq
- Hash
- Ord
- Pickle
- Pretty
- Reduce
- Seq

- Shrink
- Size
- Some
- Transform
- TypeExp
- TypeHash
- TypeInfo
Example: Generic Equality

• Desired:
  ```
  val eq : α Eq.t → α × α → Bool.t
  - Where Eq.t is the type representation type constructor
  ```

• Just define:
  ```
  structure Eq = (type α t = α × α → Bool.t)
  val eq : α Eq.t → α × α → Bool.t = id
  ```

• How to build type representations?
Equality types are trivial:

\[
\begin{align*}
&\text{val unit : Unit.t Eq.t = op =} \\
&\text{val int : Int.t Eq.t = op =} \\
&\text{val string : String.t Eq.t = op =}
\end{align*}
\]

So are some non-equality types:

\[
\begin{align*}
&\text{val real : Real.t Eq.t = fn (l, r) ⇒} \\
&\quad \text{PackRealBig.toBytes l = PackRealBig.toBytes r} \\
&\quad \text{Makes sense: reflexive, symmetric, antisymmetric, and transitive} \\
&\quad \text{Application: unpickle \ (pickle x) = x}
\end{align*}
\]

What about user-defined types?
UDTs via Sums-of-Products 1/2

• First define sum and product datatypes:

\[
\text{datatype } (\alpha, \beta) \text{ sum } = \text{INL of } \alpha \mid \text{INR of } \beta \\
\text{datatype } (\alpha, \beta) \text{ product } = \& \text{ of } \alpha \times \beta \\
\text{infix } \& \oplus \otimes
\]

• And equality on sums and products:

\[
\text{val op } \oplus : \alpha \text{ Eq.t } \times \beta \text{ Eq.t } \rightarrow (\alpha, \beta) \text{ Sum.t Eq.t } = \\
\text{ fn (eA, eB) } \Rightarrow \text{ fn (INL l, INL r) } \Rightarrow \text{ eA (l, r) } \\
\mid (\text{INR l, INR r) } \Rightarrow \text{ eB (l, r) } | \_ \Rightarrow \text{false}
\]

\[
\text{val op } \otimes : \alpha \text{ Eq.t } \times \beta \text{ Eq.t } \rightarrow (\alpha, \beta) \text{ Product.t Eq.t } = \\
\text{ fn (eA, eB) } \Rightarrow \text{ fn (IA & IB, rA & rB) } \Rightarrow \\
\text{ eA (IA, rA) andalso eB (rA & rB)}
\]
Then define isomorphism witness type:

\[
\text{type } (\alpha, \beta) \text{ iso } = (\alpha \to \beta) \times (\beta \to \alpha)
\]

- Note: Should be total!

And equality given a witness:

\[
\text{val iso : } \beta \text{ Eq.t } \to (\alpha, \beta) \text{ Iso.t } \to \alpha \text{ Eq.t } = \text{fn eB } \Rightarrow
\text{fn (a2b, b2a) } \Rightarrow \text{fn (lA, rA) } \Rightarrow \text{eB (a2b lA, a2b rA)}
\]

Example:

\[
\text{val option : } \alpha \text{ Eq.t } \to \alpha \text{ Option.t Eq.t } = \text{fn a } \Rightarrow
\text{iso (unit } \oplus \text{ a)}
\]

\[
(\text{fn NONE } \Rightarrow \text{INL () } | \text{ SOME a } \Rightarrow \text{INR a},
\text{fn INL () } \Rightarrow \text{NONE } | \text{ INR a } \Rightarrow \text{SOME a})
\]
Value Recursion Challenge

• What about recursive datatypes:
  ```
  val rec list : α Eq.t → α List.t Eq.t = fn a ⇒
  iso (unit ⊕ (a ⊗ list a))
  (fn [] ⇒ INL () | x::xs ⇒ INR (x & xs),
    fn INL () ⇒ [] | INR (x & xs) ⇒ x::xs)
  - Type checks, but diverges!
  ```

• \(\eta\)-expansion not a solution
  - Doesn't work for pairs of functions

• We must use a fixpoint combinator
  - But how do you compute fixpoints over arbitrary products of multiple abstract types?
Type-Indexed Fix 1/3

• Signature for a type-indexed fix:

```plaintext
signature TIE = sig
  type α dom and α cod type α t = α dom → α cod
  val fix : α t → (α → α) → α
  val pure : (Unit.t → (α × (α → α))) → α t
  val ⊗ : α t × β t → (α, β) Product.t t
  val iso : β t → (α, β) Iso.t → α t
end
```
Type-Indexed Fix 2/3

• An implementation of type-indexed fix:

```plaintext
structure Tie :> TIE = struct
  type α dom = Unit.t and α cod = Unit.t → α × (α → α)
  type α t = α dom → α cod
  fun fix aW f = let val (a, tA) = aW () () in tA (f a) end
  val pure = const
  fun iso bW (a2b, b2a) () () =
    let val (b, tB) = bW () () in (b2a b, b2a o tB o a2b) end
  fun op ⊗ (aW, bW) () () =
    let val (a, tA) = aW () () val (b, tB) = bW () ()
    in (a & b, fn a & b ⇒ tA a & tB b) end
end
```
Type-Indexed Fix 3/3

• An ad-hoc witness for functions:

```plaintext
structure Tie = struct open Tie
    val function : (α → β) t = fn ? ⇒
        pure (fn () ⇒ let
            val r = ref (fn _ ⇒ raise Fix)
            in
                (fn x ⇒ !r x,
                 fn f ⇒ (r := f ; f))
            end) ?
        end
```

• Back to the Eq generic...
Tying the Knot

• First we define a fixpoint witness for the `Eq` type representation
  
  ```haskell
  val Y : α Eq.t Tie.t = Tie.function
  ```

• Example:
  
  ```haskell
  val list : α Eq.t → α List.t Eq.t = fn a ⇒
  Tie.fix Y (fn aList ⇒
    iso (unit ⊕ (a ⊗ aList))
      (fn [] ⇒ INL () | x::xs ⇒ INR (x & xs),
       fn INL () ⇒ [] | INR (x & xs) ⇒ x::xs))
  ```

• Thanks to Tie.⊗, mutually recursive datatypes are not a problem.
Composability 1/2

• To address composability, the type representation is made to carry extra data $\chi$:

```plaintext
signature OPEN_REP = sig
  type ($\alpha$, $\chi$) t and ($\alpha$, $\chi$) s and ($\alpha$, $\kappa$, $\chi$) p
  val getT : ($\alpha$, $\chi$) t $\rightarrow$ $\chi$
  val mapT : ($\chi$ $\rightarrow$ $\chi$) $\rightarrow$ (($\alpha$, $\chi$) t $\rightarrow$ ($\alpha$, $\chi$) t)
  val getS : ($\alpha$, $\chi$) s $\rightarrow$ $\chi$
  val mapS : ($\chi$ $\rightarrow$ $\chi$) $\rightarrow$ (($\alpha$, $\chi$) s $\rightarrow$ ($\alpha$, $\chi$) s)
  val getP : ($\alpha$, $\kappa$, $\chi$) p $\rightarrow$ $\chi$
  val mapP : ($\chi$ $\rightarrow$ $\chi$) $\rightarrow$ (($\alpha$, $\kappa$, $\chi$) p $\rightarrow$ ($\alpha$, $\kappa$, $\chi$) p)
end
```
Composability 2/2

- And structural cases made to build the extra data:

signature OPEN_CASES = sig

structure Rep : OPEN_REP

val iso : (δ → (α, β) Iso.t → γ) →

   (β, δ) Rep.t → (α, β) Iso.t → (α, γ) Rep.t

val ⊗ : (γ × δ → ε) →

   (α, κ, γ) Rep.p × (β, κ, δ) Rep.p →

   ((α, β) Product.t, κ, ε) Rep.p

val Y : x Tie.t → (α, x) Rep.t Tie.t

val list : (γ → δ) → (α, γ) Rep.t → (α List.t, δ) Rep.t

val int : γ → (Int.t, γ) Rep.t

(* ... *)
Layering Generics

• The open rep and cases allow one to extend a generic. We do so by means of layering functors:
  - LayerRep (OPEN_REP, CLOSED_REP) :> LAYERED_REP
  - LayerCases (OPEN_CASES, LAYERED_REP, CLOSED_CASES) :> OPEN_CASES
  - LayerDepCases (OPEN_CASES, LAYERED_REP, DEP_CASES) :> OPEN_CASES
The Benefit

● Having the binary tree type rep means that we can
  - pretty print binary trees,
  - pickle and unpickle them,
  - compare them for equality,
  - hash them
  - reduce and transform them,
  - ...

● Let's try...
Goals and Requirements

- Available yesterday (SML'97)
- Reasonably expressive (eq, ord, show, read, pickle-unpickle, hash, arbitrary, ...)
- Support all types (mutually rec., mutable)
- Specialization required by applications
- Composability for convenient use
- Not a toy – Algs must do The Right Thing
- Reasonably efficient
In Summary

• First you select which generics you want,
  – add the generics one-by-one to a composition, and
  – close it for use

• Then you define type rep constructors for your types

• And you then get to use those generic utility functions with your types
Three type cons for type reps?

- SML's datatypes are not binary sums and tuples & records are not binary products!

- So, we generalize:
  
  \[
  \text{signature CLOSED_REP} = (\text{type } \alpha \ t \ \text{and} \ \alpha \ s \ \text{and} \ (\alpha, \ k) \ p) 
  \]

  - Distinguishes between complete and incomplete types as well as tuples and records
  
  - The extra tycons are useful; sometimes you really want different representations for sums and products (e.g. pickle/unpickle, read)
Order

datatype order = LESS | EQUAL | GREATER

val order : Order.t Rep.t =
  iso (data (C0 (C"LESS") ⊕ C0 (C"EQUAL") ⊕ C0 (C"GREATER")))
  (fn LESS ⇒ INL (INL ()) | EQUAL ⇒ INL (INR ()) | GREATER ⇒ INR (),
   fn INL (INL ()) ⇒ LESS | INL (INR ()) ⇒ EQUAL | INR () ⇒ GREATER)

iso
  /
data
  /
⊕

⊕
C0 (C"GREATER")

⊕
C0 (C"LESS")

⊕
C0 (C"EQUAL")