Generics for the Working ML'er

Vesa Karvonen
University of Helsinki

Why Generics?

- An innocent looking example:

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

Test Output

```
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:

  $\text{eq}_{\alpha} : \alpha \times \alpha \to \text{Bool.t}$

  $\text{show}_{\alpha} : \alpha \to \text{String.t}$

- 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

eq : α Eq.t $\to \alpha \times \alpha \to$ Bool.t

show : α Show.t $\to \alpha \to$ String.t

Value-Dependent

- Witness the value

  $\alpha \times \alpha \to$ Bool.t

  $\alpha \to$ String.t
- Hard to compose
- Easy to specialize
- Vanilla H-M

Value-Independent

- Witness the type

  $\alpha \leftrightarrow$ u
- Easy to compose
- Hard to specialize
- GADTs, Existentials, Universal Type

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

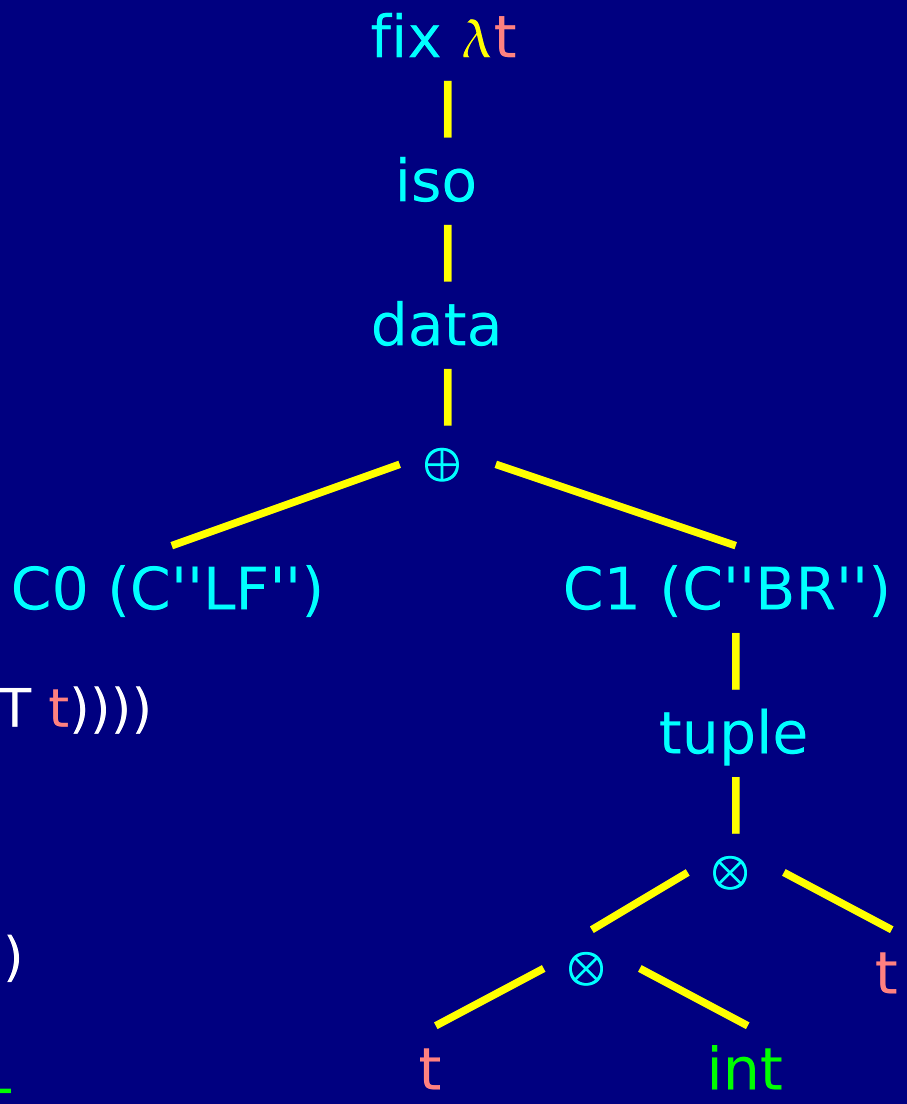
```
signature CLOSED_REP = sig type α t and α s and (α, κ) p end
signature CLOSED_CASES = sig
  structure Rep : CLOSED_REP
  val iso : β Rep.t → (α, β) Iso.t → α Rep.t
  val ⊗ : (α, κ) Rep.p × (β, κ) Rep.p → ((α, β) Product.t, κ) Rep.p
  val T : α Rep.t → (α, Generics.Tuple.t) Rep.p
  val R : Generics.Label.t → α Rep.t → (α, Generics.Record.t) Rep.p
  val tuple : (α, Generics.Tuple.t) Rep.p → α Rep.t
  val record : (α, Generics.Record.t) Rep.p → α Rep.t
  val ⊕ : α Rep.s × β Rep.s → ((α, β) Sum.t) Rep.s
  val C0 : Generics.Con.t → Unit.t Rep.s
  val C1 : Generics.Con.t → α Rep.t → α Rep.s
  val data : α Rep.s → α Rep.t
  val Y : α Rep.t Tie.t
  val → : α Rep.t × β Rep.t → (α → β) Rep.t
  val refc : α Rep.t → α Ref.t Rep.t
  (* ... *)
```

Binary Tree

```
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

Interface of a Generic

```
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
```

And another...

```
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)

```
structure Generic = struct structure Open = RootGeneric end
```

- Equality ($(G)/with/eq.sml)

```
structure Generic = struct
  structure Open = WithEq (Generic)
  open Generic Open
end
```

- Hash ($(G)/with/hash.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:

```
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!

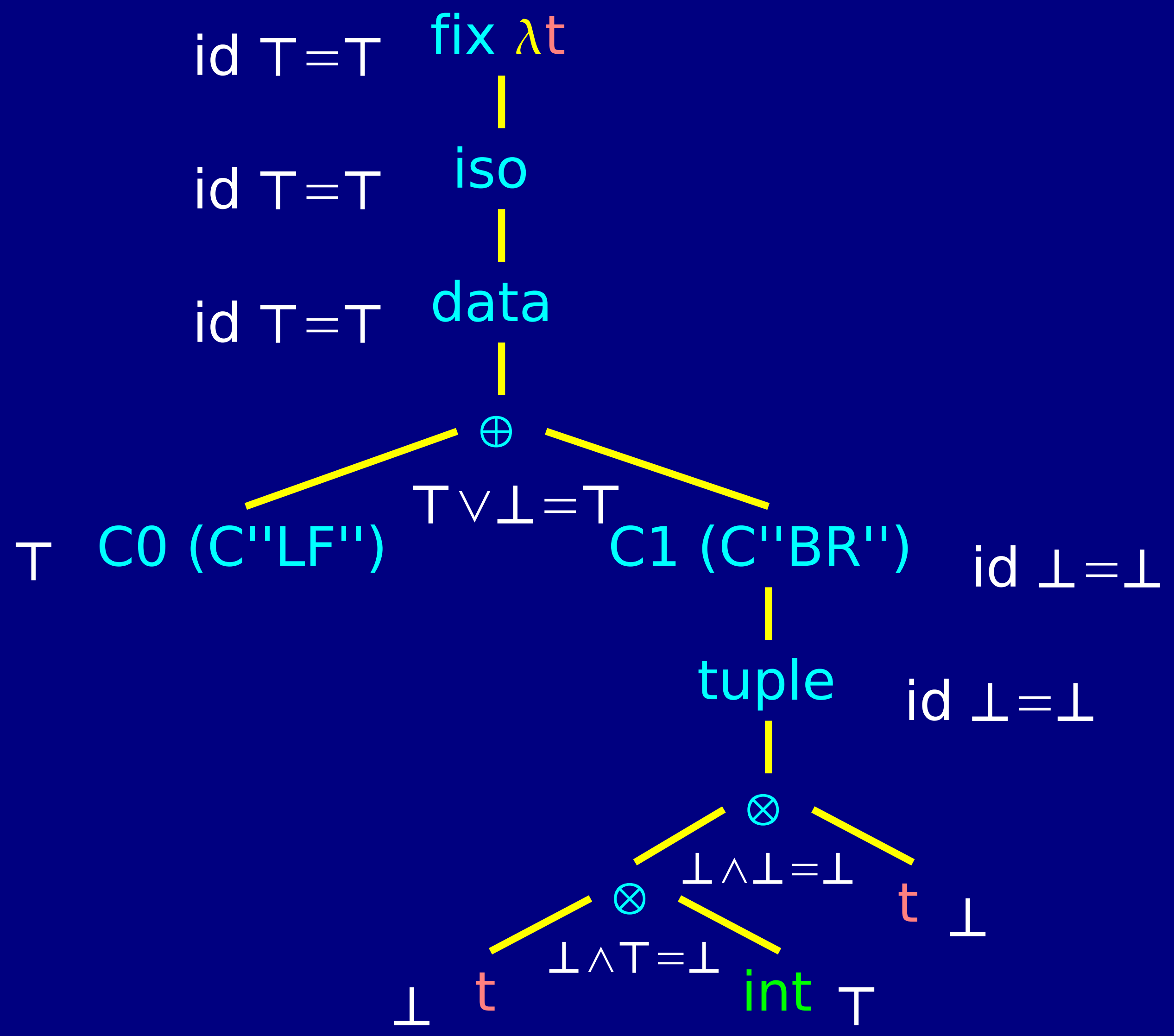
Some

val some : (α, χ) SomeRep.t → α

- 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…

```
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?



Pretty

val pretty : (α, χ) PrettyRep.t $\rightarrow$ α $\rightarrow$ 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

val pickle : (α, χ) PickleRep.t $\rightarrow \alpha \rightarrow$ String.t
val unpickle : (α, χ) PickleRep.t $\rightarrow$ 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 $\to \alpha \times \alpha \to$ Bool.t

  – Where Eq.t is the type representation type constructor

- Just define:

  structure Eq = (type α t = $\alpha \times \alpha \to$ Bool.t)

  val eq : α Eq.t $\to \alpha \times \alpha \to$ Bool.t = id

- How to build type representations?

Nullary TyCons

- Equality types are trivial:

```
val unit : Unit.t Eq.t = op =
val int : Int.t Eq.t = op =
val string : String.t Eq.t = op =
```

- So are some non-equality types:

```
val real : Real.t Eq.t = fn (l, r) ⇒
  PackRealBig.toBytes l = PackRealBig.toBytes r
```

  - Makes sense: reflexive, symmetric, antisymmetric, and transitive
  - Application: unpickle (pickle x) = x
- What about user-defined types?

UDTs via Sums-of-Products 1/2

- First define sum and product datatypes:

```
datatype (α, β) sum = INL of α | INR of β
datatype (α, β) product = & of α × β
infix & ⊕ ⊗
```

- And equality on sums and products:

```
val op ⊕ : α Eq.t × β Eq.t → (α, β) Sum.t Eq.t =
  fn (eA, eB) ⇒ fn (INL l, INL r) ⇒ eA (l, r)
                 | (INR l, INR r) ⇒ eB (l, r) | _ ⇒ false
val op ⊗ : α Eq.t × β Eq.t → (α, β) Product.t Eq.t =
  fn (eA, eB) ⇒ fn (lA & lB, rA & rB) ⇒
     eA (lA, rA) andalso eB (rA & rB)
```

UDTs via Sums-of-Products 2/2

- Then define isomorphism witness type:

```
type (α, β) iso = (α → β) × (β → α)
```

  - Note: Should be <u>total</u>!

- And equality given a witness:

```
val iso : β Eq.t → (α, β) Iso.t → α Eq.t = fn eB ⇒
  fn (a2b, b2a) ⇒ fn (lA, rA) ⇒ eB (a2b lA, a2b rA)
```

- Example:

```
val option : α Eq.t → α Option.t Eq.t = fn a ⇒
  iso (unit ⊕ a)
       (fn NONE ⇒ INL () | SOME a ⇒ INR a,
        fn INL () ⇒ NONE | INR a ⇒ 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!
- η-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:

```
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:

```
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:

```
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

```
val Y : α Eq.t Tie.t = Tie.function
```

- Example:

```
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 χ:

```
signature OPEN_REP = sig
  type (α, χ) t and (α, χ) s and (α, κ, χ) p
  val getT : (α, χ) t → χ
  val mapT : (χ → χ) → ((α, χ) t → (α, χ) t)
  val getS : (α, χ) s → χ
  val mapS : (χ → χ) → ((α, χ) s → (α, χ) s)
  val getP : (α, κ, χ) p → χ
  val mapP : (χ → χ) → ((α, κ, χ) p → (α, κ, χ) p)
end
```

Composability 2/2

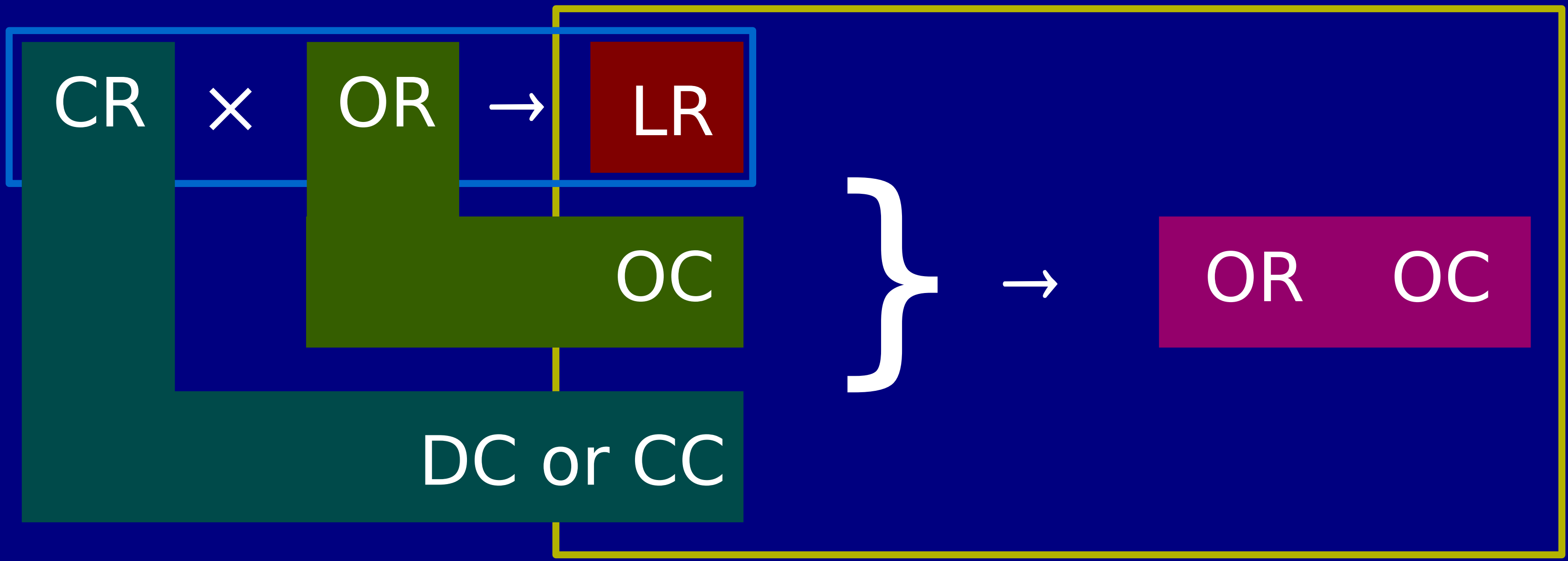
- And structural cases made to build the extra data:

```
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 : χ Tie.t → (α, χ) 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

Layering Scheme



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:

  signature CLOSED_REP = (type α t and α s and (α, κ) 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)
```