Structural Typing for Structured Products

Tim Williams
Peter Marks

8th October 2014

BARCLAYS

Background

The FPF Framework

- A standardized representation for describing payoffs
- A common suite of tools for trades which use this representation
    - UI for providing trade parameters
    - Mathematical document descriptions
    - Pricing and risk management
    - Barrier analysis
    - Payments and other lifecycle events

FPF Lucid

- A DSL for describing exotic payoffs and strategies
- Control constructs based around schedules
- Produces abstract syntax–allowing multiple interpretations
- Damas-Hindley-Milner type inference with constraints and polymorphic extensible row types

Lucid language

Articulation driven design

Lucid language

Articulation driven design

Lucid type system

Structural typing with

Row Polymorphism

A simple numeric expression

$\exp(x)$

A simple numeric expression

$\exp(x)$

Monomorphic

$\exp : \text{Double} \longrightarrow \text{Double}$
$x : \text{Double}$
$\exp(x) : \text{Double}$

A conditional expression

```
if c then x else y
```

A conditional expression

```
if c then x else y
```

Polymorphic

if _ then _ else _ : $(\text{Bool}, a, a) \to a$

c : Bool

x : a

y : a

if c then x else y : a

- Hindley-Milner type system

Overloaded numeric literal

```
x + 42
```

Overloaded numeric literal

x + 42

Subtyping

$(+) : (\text{Num}, \text{Num}) \to \text{Num}$

42 : Integer

x : Num

x + 42 : Num

- Subtyping constraints difficult to solve with full inference
- A complex extension to Hindley-Milner

Overloaded numeric literal

```
x + 42
```

Polymorphic with type variable constraints

$(+) : \text{Num}\ a \Rightarrow (a, a) \rightarrow a$

$42 : \text{Num}\ a \Rightarrow a$

$x : \text{Num}\ a \Rightarrow a$

$x + 42 : \text{Num}\ a \Rightarrow a$

- Any type variable can have a single constraint
- Unifier ensures constraints are met
- Simple extension to Hindley-Milner

A simple Lucid function

```
function capFloor(perf, cap, floor)
  return max(floor, min(cap, perf))
end
```

A simple Lucid function

```
function capFloor(perf, cap, floor)
  return max(floor, min(cap, perf))
end
```

$\text{capFloor} : \text{Num}\ a \Rightarrow (a, a, a) \rightarrow a$

BARCLAYS

A simple Lucid function

```
function capFloor(perf, cap, floor)
  return max(floor, min(cap, perf))
end
```

capFloor : Num $a \Rightarrow (a, a, a) \to a$

```
capFloor(perf, 0, 1)
```

- Not obvious which argument when applying function

Grouping and labelling arguments

```
function capFloor(perf, {cap, floor})
  return max(floor, min(cap, perf))
end
```

Grouping and labelling arguments

```
function capFloor(perf, {cap, floor})
  return max(floor, min(cap, perf))
end
```

Records via Nominal typing

data Num $a \Rightarrow$ CapFloor a = CapFloor
$\{$cap : a, floor : a $\}$

capFloor : Num $a \Rightarrow (a,$ CapFloor $a) \rightarrow a$

Grouping and labelling arguments

```
function capFloor(perf, {cap, floor})
  return max(floor, min(cap, perf))
end
```

Records via Nominal typing

data Num $a \Rightarrow$ CapFloor a = CapFloor
$\{$cap : a, floor : a $\}$

capFloor : Num $a \Rightarrow (a,$ CapFloor $a) \rightarrow a$

- Don't want to force users to define data types

Grouping and labelling arguments

```
function capFloor(perf, {cap, floor})
  return max(floor, min(cap, perf))
end
```

Records via Nominal typing

data Num $a \Rightarrow$ CapFloor a = CapFloor
  $\{$cap : a, floor : a $\}$

capFloor : Num $a \Rightarrow (a,$ CapFloor $a) \rightarrow a$

- Don’t want to force users to define data types
- Don’t want to force users to name a combination of fields

Grouping and labelling arguments

```
function capFloor(perf, {cap, floor})
  return max(floor, min(cap, perf))
end
```

Records via Nominal typing

data Num $a \Rightarrow$ CapFloor a = CapFloor
  $\{$cap : a, floor : a $\}$

capFloor : Num $a \Rightarrow (a,$ CapFloor $a) \rightarrow a$

- Don't want to force users to define data types
- Don't want to force users to name a combination of fields
- Want to use the same fields in different data types

Grouping and labelling arguments

```
function capFloor(perf, {cap, floor})
  return max(floor, min(cap, perf))
end
```

Structural record types

$\text{capFloor} : \text{Num}\ a \Rightarrow$
$(a, \{\text{cap} : a, \text{floor} : a\}) \rightarrow a$

Grouping and labelling arguments

```
function capFloor(perf, {cap, floor})
  return max(floor, min(cap, perf))
end
```

```
capFloor(perf, {cap=0, floor=1})
capFloor(perf, {floor=1, cap=0})
```

Structural record types

$\text{capFloor} : \text{Num}\ a \Rightarrow$
$(a, \{\text{cap} : a, \text{floor} : a\}) \to a$

- Unifier is agnostic to field order

Grouping and labelling arguments

```
function capFloor(perf, {cap, floor})
  return max(floor, min(cap, perf))
end
```

```
capFloor(perf, {cap=0, floor=1})
capFloor(perf, {floor=1, cap=0})
```

Structural record types

$$\text{capFloor} : \text{Num}\ a \Rightarrow (a, \{\text{cap} : a, \text{floor} : a\}) \rightarrow a$$

- Unifier is agnostic to field order
- Note the above is still not quite what Lucid infers

BARCLAYS

Grouping and labelling arguments

```
function capFloor(perf, r)
  return max(floor, min(r.cap, r.perf))
end
```

```
capFloor(perf, {cap=0, floor=1})
capFloor(perf, {floor=1, cap=0})
```

Structural record types

$\text{capFloor} : \text{Num}\ a \Rightarrow$
$(a, \{\text{cap} : a, \text{floor} : a\}) \to a$

- Unifier is agnostic to field order
- Note the above is still not quite what Lucid infers
- Pattern matching is just syntactic sugar for field selection

Ignoring additional fields

```
function kgcf(perf, r)
  return capFloor( r.part * (perf - r.strike)
                 , {cap = r.cap, floor = r.floor})
end

kgcf(perf, {part=1, strike=0.9, cap=0, floor=1.2})
```


BARCLAYS

Ignoring additional fields

```
function kgcf(perf, r)
  return capFloor(r.part * (perf - r.strike), r)
end


kgcf(perf, {part=1, strike=0.9, cap=0, floor=1.2})
```

Ignoring additional fields

```
function kgcf(perf, r)
  return capFloor(r.part * (perf - r.strike), r)
end
```

```
kgcf(perf, {part=1, strike=0.9, cap=0, floor=1.2})
```

Structural Record types

$$\text{capFloor} : \text{Num } a \Rightarrow (a, \{\text{cap} : a, \text{floor} : a\}) \rightarrow a$$
$$\text{kgcf} : \text{Num } a \Rightarrow (a, \{\text{part} : a, \text{strike} : a, \text{cap} : a, \text{floor} : a\}) \rightarrow a$$

- How do we allow a superset of fields to be passed to CapFloor?

Ignoring additional fields

```
function kgcf(perf, r)
  return capFloor(r.part * (perf - r.strike), r)
end
```

```
kgcf(perf, {part=1, strike=0.9, cap=0, floor=1.2})
```

Structural Record types

$$\text{capFloor} : \text{Num } a \Rightarrow (a, \{\text{cap} : a, \text{floor} : a\}) \to a$$
$$\text{kgcf} : \text{Num } a \Rightarrow (a, \{\text{part} : a, \text{strike} : a, \text{cap} : a, \text{floor} : a\}) \to a$$

- How do we allow a superset of fields to be passed to CapFloor?
- Subtyping would require a new type system and inference algorithm

Ignoring additional fields

```
function kgcf(perf, r)
  return capFloor(r.part * (perf - r.strike), r)
end
```

```
kgcf(perf, {part=1, strike=0.9, cap=0, floor=1.2})
```

Polymorphic extensible Records

$$\text{capFloor} : \text{Num } a \Rightarrow (a, \{\text{cap} : a, \text{floor} : a \,|r\}) \rightarrow a$$
$$\text{kgcf} : \text{Num } a \Rightarrow (\text{perf}, \{\text{part} : a, \text{strike} : a, \text{cap} : a, \text{floor} : a \,|s\}) \rightarrow a$$

- How do we allow a superset of fields to be passed to CapFloor?
- Subtyping would require a new type system and inference algorithm
- Can use parametric polymorphism by using a type variable to represent the remaining fields

Extending Records

```
function gcfBasket(perf, weights, r)
 return kgcf( sumProduct(perfs, weights)
             , {strike=1, part=r.part, cap=r.cap, floor=r.floor})
end
```

Extending Records

```
function gcfBasket(perf, weights, r)
  return kgcf(sumProduct(perfs, weights), {strike=1 |r})
end
```

Extending Records

```
function gcfBasket(perf, weights, r)
  return kgcf(sumProduct(perfs, weights), {strike=1 |r})
end
```

Polymorphic extensible Records

$$\text{kgcf} : (\text{Num } a, s/\text{part}/\text{strike}/\text{cap}/\text{floor}) \Rightarrow$$
$$(a, \{\text{part} : a, \text{strike} : a, \text{cap} : a, \text{floor} : a \mid s\}) \to a$$

$$\text{gcfBasket} : (\text{Num } a, r/\text{part}/\text{strike}/\text{cap}/\text{floor}) \Rightarrow$$
$$(a, \{\text{part} : a, \text{cap} : a, \text{floor} : a \mid r\}) \to a$$

Extending Records

```
function gcfBasket(perf, weights, r)
  return kgcf(sumProduct(perfs, weights), {strike=1 |r})
end
```

Polymorphic extensible Records

$$\text{kgcf} : (\text{Num } a, s/\text{part}/\text{strike}/\text{cap}/\text{floor}) \Rightarrow$$
$$(a, \{\text{part} : a, \text{strike} : a, \text{cap} : a, \text{floor} : a \mid s\}) \to a$$

$$\text{gcfBasket} : (\text{Num } a, r/\text{part}/\text{strike}/\text{cap}/\text{floor}) \Rightarrow$$
$$(a, \{\text{part} : a, \text{cap} : a, \text{floor} : a \mid r\}) \to a$$

- Type inference introduces ”lacks” constraints on row variables

Row Polymorphism

The idea of row (parametric) polymorphism is to use a type variable to represent any additional unknown fields:[1]

$$\texttt{gcfBasket : } (\texttt{Num } a\texttt{, } r\texttt{/part/strike/cap/floor}) \Rightarrow$$
$$(a\texttt{, } \{\texttt{part : } a\texttt{, cap : } a\texttt{, floor : } a \ | r\}) \rightarrow a$$

[1] Row polymorphism can be implemented with or without the lacks predicate, depending on whether repeated (scoped) labels are desired.

Type constructors: Row kinds

- The empty row

$$(\!|\,|\!) : \mathrm{ROW}$$

- Extend a row type (one constructor per label):

$$(\!| \ell : \_ \mid \_ |\!) : \star \to \mathrm{ROW} \to \mathrm{ROW}$$

Type constructors: Records

- Construct a Record from a row type (gives product types, structurally):

$$\{\_\} : \mathrm{ROW} \rightarrow \star$$

Primitive operations on Records

- Selection

$$(\_.\ell) \; : \; \forall a r. \; (r/\ell) \Rightarrow \{\ell \; : \; a \; | r\} \rightarrow a$$

- Restriction

$$(\_ \; / \; \ell) \; : \; \forall a r. \; (r/\ell) \Rightarrow \{\ell \; : \; a \; | r\} \rightarrow \{r\}$$

- Extension[2]

$$\{\ell = \_ | \_\} \; : \; \forall a r. \; (r/\ell) \Rightarrow (a, \; \{r\}) \rightarrow \{\ell \; : \; a \; | r\}$$

[2] Note that Record literals are desugared to record extension.

Enums and switching behaviour

```
function calcOffset(ccy)
  return
    if ccy == USD then 3
    else if ccy == JPY then 2
    else 0
end
```

Enums and switching behaviour

```
function calcOffset(ccy)
  return
    if ccy == USD then 3
    else if ccy == JPY then 2
    else 0
end
```

- No way to limit the set of atoms that can be used

Enums and switching behaviour

```
function calcOffset(ccy)
  return
    if ccy == USD then 3
    else if ccy == JPY then 2
    else 0
end
```

- No way to limit the set of atoms that can be used
- Forced to provide a default value in the else clause

BARCLAYS

Enums and switching behaviour

```
function calcOffset(ccy)
  return
    case ccy of
      USD → 3,
      JPY → 2
end
```

Enums and switching behaviour

```
function calcOffset(ccy)
  return
    case ccy of
      USD → 3,
      JPY → 2
end
```

Row polymorphism

$$\text{calcOffset} : \text{Num}\ a \Rightarrow \langle \text{USD}, \text{JPY} \rangle \rightarrow a$$

Enums and switching behaviour

```
function calcOffset(ccy)
  return
    case ccy of
      USD → 3,
      JPY → 2
end
```

Row polymorphism

$$\text{calcOffset} : \text{Num } a \Rightarrow \langle \text{USD}, \text{JPY} \rangle \rightarrow a$$

- Enums can be implemented using row types with unit fields

Enums and switching behaviour

```
function calcOffset(ccy)
  return
    case ccy of
      USD → 3,
      JPY → 2
end
```

Row polymorphism

$\text{calcOffset} : \text{Num } a \Rightarrow \langle \text{USD}, \text{JPY} \rangle \rightarrow a$

- Enums can be implemented using row types with unit fields
- Note the top-level type is closed

Extending enums and reusing behaviour

```
function calcOffsetExt(ccy)
  return
    case ccy of
      GBP → 3,
      otherwise c → calcOffset(c)
end
```

Extending enums and reusing behaviour

```
function calcOffsetExt(ccy)
  return
    case ccy of
      GBP → 3,
      otherwise c → calcOffset(c)
end
```

Polymorphic extensible cases

$$\text{calcOffset} : \text{Num } a \Rightarrow \langle \text{USD, JPY} \rangle \rightarrow a$$

$$\text{calcOffsetExt} : \text{Num } a \Rightarrow \langle \text{GBP, USD, JPY} \rangle \rightarrow a$$

$$\text{c} : \langle \text{USD, JPY} \rangle$$

Extending enums and reusing behaviour

```
function calcOffsetExt(ccy)
  return
    case ccy of
      GBP → 3,
      otherwise c → calcOffset(c)
end
```

Polymorphic extensible cases

$$\text{calcOffset} : \text{Num}\ a \Rightarrow \langle\text{USD, JPY}\rangle \rightarrow a$$

$$\text{calcOffsetExt} : \text{Num}\ a \Rightarrow \langle\text{GBP, USD, JPY}\rangle \rightarrow a$$

$$\text{c} : \langle\text{USD, JPY}\rangle$$

- Composition of cases using delegation

Extending enums and reusing behaviour

```
function calcOffsetExt(ccy)
  return
    case ccy of
      GBP → 3,
      otherwise c → calcOffset(c)
end
```

Polymorphic extensible cases

calcOffset : Num $a \Rightarrow$
$\langle$USD, JPY$\rangle \rightarrow a$

calcOffsetExt : Num $a \Rightarrow$
$\langle$GBP, USD, JPY$\rangle \rightarrow a$

c : $\langle$USD, JPY$\rangle$

- Composition of cases using delegation
- Creates a new type containing a superset of fields

Extending enums and reusing behaviour

```
function calcOffsetExt(ccy)
  return
    case ccy of
      GBP → 3,
      otherwise c → calcOffset(c)
end
```

Polymorphic extensible cases

calcOffset : Num $a \Rightarrow$
$\langle$USD, JPY$\rangle \rightarrow a$

calcOffsetExt : Num $a \Rightarrow$
$\langle$GBP, USD, JPY$\rangle \rightarrow a$

c : $\langle$USD, JPY$\rangle$

- Composition of cases using delegation
- Creates a new type containing a superset of fields
- Flexibility similar to OOP subclassing, without giving up extensibility of functions

Limiting the behaviour of existing code

```
function calcOffset2(ccy)
  return calcOffsetExt( ⟨JPY |ccy⟩ )
end
```

Limiting the behaviour of existing code

```
function calcOffset2(ccy)
  return calcOffsetExt( ⟨JPY |ccy⟩ )
end
```

Embedding

calcOffsetExt : Num $a \Rightarrow$
$\langle$GBP, USD, JPY$\rangle \rightarrow a$

calcOffset2 : Num $a \Rightarrow$
$\langle$GBP, USD$\rangle \rightarrow a$

Limiting the behaviour of existing code

```
function calcOffset2(ccy)
  return calcOffsetExt( ⟨JPY |ccy⟩ )
end
```

Embedding

calcOffsetExt : Num $a \Rightarrow$
$\langle$GBP, USD, JPY$\rangle \rightarrow a$

calcOffset2 : Num $a \Rightarrow$
$\langle$GBP, USD$\rangle \rightarrow a$

- Embedding adds JPY to the type and restricts its values as possible input

Overriding existing behaviour

```
function calcOffsetExt2(ccy)
  return
    case ccy of
      override USD → 4,
      otherwise c → calcOffsetExt(c)
end
```

Overriding existing behaviour

```
function calcOffsetExt2(ccy)
  return
    case ccy of
      override USD → 4,
      otherwise c → calcOffsetExt(c)
end
```

Embedding

$$\text{calcOffsetExt} : \text{Num } a \Rightarrow \langle \text{GBP, USD, JPY} \rangle \to a$$

$$\text{calcOffsetExt2} : \text{Num } a \Rightarrow \langle \text{GBP, USD, JPY} \rangle \to a$$

$$\text{c} : \langle \text{GBP, USD, JPY} \rangle$$

- Override is syntactic sugar for embedding in the otherwise clause

Optional arguments

```
function calcBasket(perfs, aggregation, weights)
  return case aggregation of
    Worst → minArray(perfs),
    Best → maxArray(perfs),
    Weighted
      → sumProduct(perfs, weights)
end
```

Optional arguments

```
function calcBasket(perfs, aggregation, weights)
  return case aggregation of
    Worst → minArray(perfs),
    Best → maxArray(perfs),
    Weighted
      → sumProduct(perfs, weights)
end
```

The weights argument is only used in the Weighted case

Optional arguments

```
function calcBasket(perfs, aggregation)
  return case aggregation of
    Worst → minArray(perfs),
    Best → maxArray(perfs),
    Weighted(weights)
      → sumProduct(perfs, weights)
end
```

The weights argument is only used in the Weighted case

Optional arguments

```
function calcBasket(perfs, aggregation)
  return case aggregation of
    Worst → minArray(perfs),
    Best → maxArray(perfs),
    Weighted(weights)
      → sumProduct(perfs, weights)
end
```

The weights argument is only used in the Weighted case

Variants

$$\text{calcBasket} : r/\text{Worst}/\text{Best}/\text{Weighted} \Rightarrow \big(\text{double}[n] \; , \langle \text{Worst}, \text{Best}, \text{Weighted} : \text{double}[n] \; | r \rangle \big) \rightarrow \text{double}$$

Note that array sizes are also represented with a type variable

Type constructors: Records and Variants

- Construct a record from a row type (gives product types, structurally):

  $\{\_\} : \mathsf{ROW} \to \star$

- Construct a variant from a row type (gives sum types, structurally):

  $\langle\_\rangle : \mathsf{ROW} \to \star$

Primitive operations on Variants

- Injection (dual of selection)

$$\langle \ell = \_ \rangle \;:\; \forall a r.\ (r/\ell) \Rightarrow a \rightarrow \langle \ell : a \mid r \rangle$$

- Embedding (dual of restriction)

$$\langle \ell \mid \_ \rangle \;:\; \forall a r.\ (r/\ell) \Rightarrow \langle r \rangle \rightarrow \langle \ell : a \mid r \rangle$$

- Decomposition (dual of extension)

$$\langle \ell \in \_ ? \_ : \_ \rangle \;:\; \forall a b r.\ (r/\ell) \Rightarrow \langle \ell : a \mid r \rangle \rightarrow a \oplus \langle r \rangle$$

Primitive operations on Variants

- Injection (dual of selection)

$$\langle \ell\text{=\_} \rangle \ : \ \forall a r.\ (r/\ell) \Rightarrow a \rightarrow \langle \ell \ : \ a \ | r \rangle$$

$$(\_.\ell) \ : \ \forall a r.\ (r/\ell) \Rightarrow \{\ell \ : \ a \ | r\} \rightarrow a$$

- Embedding (dual of restriction)

$$\langle \ell | \_ \rangle \ : \ \forall a r.\ (r/\ell) \Rightarrow \langle r \rangle \rightarrow \langle \ell \ : \ a \ | r \rangle$$

$$(\_ \ / \ \ell) \ : \ \forall a r.\ (r/\ell) \Rightarrow \{\ell \ : \ a \ | r\} \rightarrow \{r\}$$

- Decomposition (dual of extension)

$$\langle \ell \in \_?\_:\_ \rangle \ : \ \forall a b r.\ (r/\ell) \Rightarrow \langle \ell \ : \ a \ | r \rangle \rightarrow a \oplus \langle r \rangle$$

$$\{\ell\text{=\_}|\_\} \ : \ \forall a r.\ (r/\ell) \Rightarrow (a,\ \{r\}) \rightarrow \{\ell \ : \ a \ | r\}$$

- Decomposition (fused with a fold on the coproduct)[3]

$$\texttt{case \_ of } \ell \rightarrow \texttt{\_, otherwise} \rightarrow \texttt{\_ :}$$
$$\forall abr.\ (r/\ell) \Rightarrow$$
$$(\langle \ell : a \mid r \rangle,\ a \rightarrow b,\ \langle r \rangle \rightarrow b) \rightarrow b$$

- The empty alternative is used to close variants:

$$\texttt{emptyAlt : } \langle\rangle \rightarrow b$$

[3] Note that the case construct provides a notion of type refinement.

Tracking Effects

```
function paySomething(amt, sched, settl)
  on sched pay Coupon amt with settl end
end
```

Tracking Effects

```
function paySomething(amt, sched, settl)
  on sched pay Coupon amt with settl end
end
```

Row-polymorphic effect types

$$\text{paySomething} : (\text{double}, \text{schedule}, \text{settlement}) \to \{\text{payments}\}\ ()$$

- Use row types to add an effect parameter to every function

$$\forall abe.\, a \to \{e\}\, b$$

- Only consider effects that are intrinsic to the language.
- Assume strict semantics, a function call inherits the effects from evaluation of its arguments.

“Lacks” constraint to restrict effects

We want to prevent users from making payments in case alternatives, as conditional payments must be handled via other primitives:

$$
\begin{array}{l}
\texttt{case \_ of } \ell \to \texttt{\_, otherwise} \to \texttt{\_ :} \\
\quad \forall abre.\ (r/\ell,\ e/\texttt{payments}) \Rightarrow \\
\qquad (\langle \ell : a \mid r \rangle,\ a \to \{e\}\ b,\ \langle r \rangle \to b) \to b
\end{array}
$$

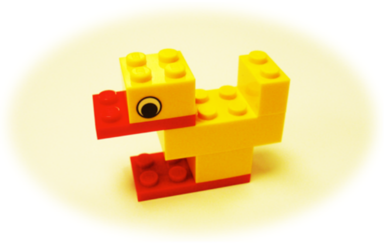
Note that we omit all unconstrained effect row variables.

An example Lucid function type

```
autocallable : { asset : asset
               , fixedCouponAmt : double
               , asianInSchedule : schedule
               , asianOutSchedule : schedule
               , couponSchedule : schedule
               , autocallSchedule : schedule
               , digitalCouponParams : { direction : <Up, Down, StrictlyUp, StrictlyDown>
                                       , level : double
                                       , amount : double
                                       }
               , perfCouponOption :    { type : <Call, Put, Forward, Straddle, Const>
                                       , strike : double
                                       , part : double
                                       }
               , autocallParams :      { direction : <Up, Down, StrictlyUp, StrictlyDown>
                                       , level : double
                                       , amount : double
                                       }
               , maturityDate : schedule
               , finalRedemptionAmt : double
               , kiSchedule : schedule
               , kiBarrierParams :     { direction : <Up, Down, StrictlyUp, StrictlyDown>
                                       , level : double
                                       }
               , kiOption :            { type : <Call, Put, Forward, Straddle, Const>
                                       , strike : double
                                       , part : double
                                       }
               } -> {payments, exit} ()
```

Structural Typing

- Type equivalence determined by the type's actual structure, not by e.g. a name as in nominal typing.
- Research literature is almost completely concerned with structural type systems (TAPL 2002)



Benefits

- Permits sharing of constructors/labels amongst different types

Benefits

- Permits sharing of constructors/labels amongst different types
- Allows reuse of code across different (extended) types

Benefits

- Permits sharing of constructors/labels amongst different types
- Allows reuse of code across different (extended) types
- No requirement or dependency on type definitions or declarations, types can be fully inferred from their usage and are self-describing

Benefits

- Permits sharing of constructors/labels amongst different types
- Allows reuse of code across different (extended) types
- No requirement or dependency on type definitions or declarations, types can be fully inferred from their usage and are self-describing
- Creation of a Nominal type from a Structural type, just requires “newtype” or similar. The inverse is not so easy.

Benefits

- Permits sharing of constructors/labels amongst different types
- Allows reuse of code across different (extended) types
- No requirement or dependency on type definitions or declarations, types can be fully inferred from their usage and are self-describing
- Creation of a Nominal type from a Structural type, just requires “newtype” or similar. The inverse is not so easy.
- Combines the flexibility of unityping, with the safety of nominal typing

Benefits

- Permits sharing of constructors/labels amongst different types
- Allows reuse of code across different (extended) types
- No requirement or dependency on type definitions or declarations, types can be fully inferred from their usage and are self-describing
- Creation of a Nominal type from a Structural type, just requires "newtype" or similar. The inverse is not so easy.
- Combines the flexibility of unityping, with the safety of nominal typing
- Achieves the composition of data-types that we see in frameworks like Data types à la carte

Structural Typing in Haskell

- Haskell vanilla ADTs and Records are nominally typed
- We regularly see the tension of e.g. Tuples/HList versus Records.
- But Haskell essentially uses structural typing exclusively for functions:

Haskell	Java (Nominal)
() -> IO ()	class Runnable { void run() }
a -> a -> Ordering	class Comparator { int compare(T, T) }
a -> b	class Function<? super T,? extends R> { R apply(T) }

An example type checker

https://github.com/willtim/row-polymorphism

References

[1] B. R. Gaster, M. P. Jones, “A Polymorphic Type System for Extensible Records and Variants”, 1996

[2] J. Garrigue, “Programming with Polymorphic Variants”, 1998

[3] J. Garrigue, “Code reuse through polymorphic variants”, 2000

[4] D. Leijen, “Extensible records with scoped labels”, 2005

[5] D. Leijen, “Koka : Programming with Row-polymorphic Effect Types”, 2013