“Blame it on Bob and Ralf”

or

Generic Typed Intermediate Languages for the Masses

A partially solved problem

Norman Ramsey

Harvard University & Microsoft Research

Typed compilation can be tedious

Write code again and again:

- Every IL requires its own type checker
- Checker needs support functions for that IL
- Capture-avoiding substitution, which I get wrong

Ralf can make things better

My hopes:

- **Write “standard” substitution once and for all**
- **Write new type checker only if there’s a new idea**

Type checking in a particular setting: 2D

Plan for this talk:

1. **Problem: Compile to 2D**
2. **Problem: Too many ILs**
3. **Problem: Too much boring code**
4. **Solution:[a] Generics!**

---

[a] Partial

The problem

Genesis of a particular multi-IL compiler

How I got into this mess (and why you might too):

1. I liked TIL
2. I liked “generics for the masses”
3. I entered the 2006 ICFP Programming Contest
4. I had to teach 1st-year PhD students

Fused into one idea:

Read cool papers; compile to 2D

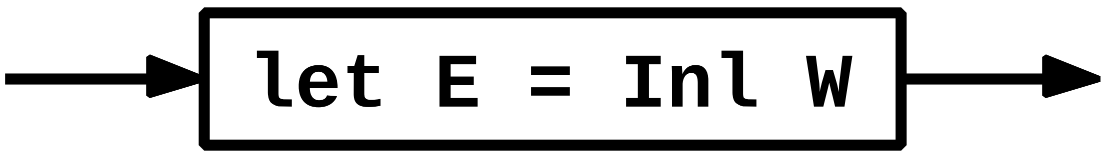
2D = real PL content + cute hackery

An “esoteric” language for the 2006 contest:

- **Classic first-order values:**

  `v ::= () | (v, v) | Inl v | Inr v`

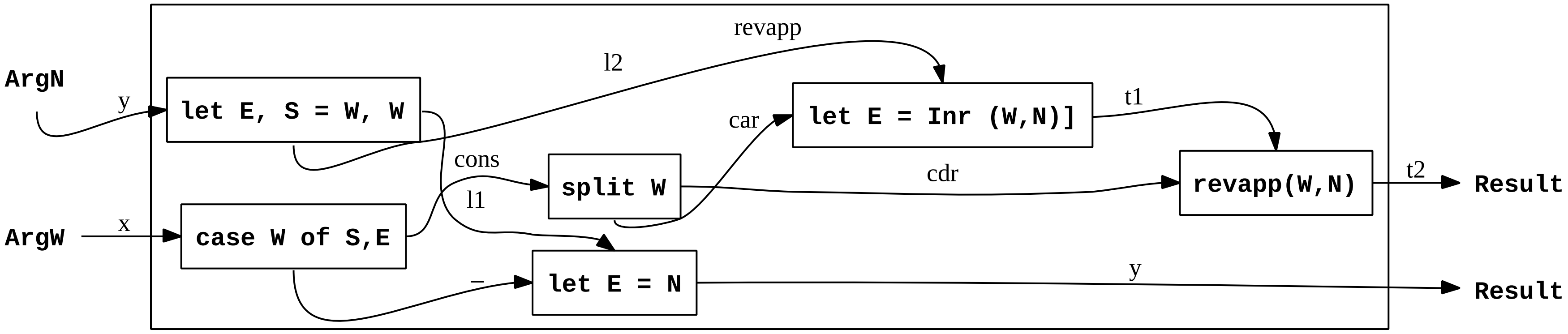
- **Computation by circuits (boxes and arrows):**



- **Named circuits with recursive instantiation (First-order functional language)**
- **Single box: 0 to 2 inedges, may do one of**
  - **Multiple assignment (0 to 2 outedges)**
  - **Function call**
  - **Pair elimination or sum elimination**

2D list reversal w/accumulating parameter

Output from my compiler, run through **`dot`**:



2D: Fundamentals lurk beneath surface

Lots of PL here:

- **Data coded by sums and products**
- **“Syntax” driven by introduction & elimination**
- **“Wires” are linear variables**
- **Control dependence coded via data dependence**
- **A circuit is a (linear!) A-normal form**

“Hell is other programming languages” —Sartran

2D compiler motivates lots of good papers

Compiling 2D requires reading:

- Parsing combinators (Hutton 1992; Fokker 1995)
- Type inference (Peyton Jones et al. 2007)
- Typed defunctionalization (Pottier and Gauthier 2005)
- A-normalization (Flanagan et al. 1993)
- Linearity (Wadler 1993)

And to help with the grunt work

- Generics for the masses (Hinze 2004)

Typed intermediate languages proliferate

- Abstract syntax (partially typed)
- Target of type inference (mutable ref cells)
- Language supporting defunctionalization (need GADTs)
- First-order target language
- Language of A-normal forms (Also used for linearization)
- Boxes and arrows (`dot`, untyped)

Lots of commonality with System F

Four TILs needn't mean drudgery

Much can be done generically!

Four TILs needn't mean drudgery

Much can be done generically!

In any language, manipulate names:

```
freeVars  :: (Language a) => a -> [Name]
isFreeIn  :: (Language a) => Name -> a -> Bool
substName :: (Language a) => (Name, Name) -> a -> a
```

Four TILs needn't mean drudgery

Much can be done generically!

In any language, manipulate names:

```
freeVars  :: (Language a) => a -> [Name]
isFreeIn  :: (Language a) => Name -> a -> Bool
substName :: (Language a) => (Name, Name) -> a -> a
```

In some languages, map name to term:

```
subst :: (Mapable e a) => (Name, e) -> a -> a
```

Four TILs needn't mean drudgery

Much can be done generically!

In any language, manipulate names:

```
freeVars  :: (Language a) => a -> [Name]
isFreeIn  :: (Language a) => Name -> a -> Bool
substName :: (Language a) => (Name, Name) -> a -> a
```

In some languages, map name to term:

```
subst :: (Mapable e a) => (Name, e) -> a -> a
```

Can even linearize generically:

```
linearize :: (Language e) => [Name] -> e -> e
  -- 1st arg is list of names to be consumed
```

The solution

Ralf's nice idea: Encode by isomorphism

Make algebraic data type $ADT \simeq v$ **where**

$$v ::= () \mid (v, v) \mid \texttt{Inl}\ v \mid \texttt{Inr}\ v \mid (iso, ADT)$$

Where iso **is** $(ADT \rightarrow \tau, \tau \rightarrow ADT)$
(iso **supplied by client)**

Ralf's nice idea: Encode by isomorphism

Make algebraic data type $ADT \simeq v$ where

$$v ::= () \mid (v, v) \mid \texttt{Inl}\ v \mid \texttt{Inr}\ v \mid (iso, ADT)$$

Where iso is $(ADT \to \tau, \tau \to ADT)$
(iso supplied by client)

My gloss: encode a language

Ralf's nice idea: Encode by isomorphism

Make algebraic data type $ADT \simeq v$ where

$$v ::= () \mid (v, v) \mid \texttt{Inl}\ v \mid \texttt{Inr}\ v \mid (iso, ADT) \mid b$$

Where iso is $(ADT \rightarrow \tau, \tau \rightarrow ADT)$
(iso supplied by client)

My gloss: encode a language

Encoding basics: Data

Ralf says:

```
data Unit      = Unit
data Plus a b = Inl a | Inr b
data Pair a b = Pair a b
data Iso a b  = Iso { fromData :: b -> a
                    , toData   :: a -> b }
```

I added

```
data Lit a = Lit a -- no vars or subterms
```

N.B. `Unit` ≃ `Lit ()`

Encoding free and bound variables

My contribution (so far):

```
data Occurrence e = Free Name
                  | Substituted e
data Binder e     = Bind Name e
```

(Linearity too, but not in this talk)

Defining generic functions

Definition by cases over encoding:

```
class Generic g where
  unit        :: g Unit
  plus        :: (Language a, Language b) =>
                              g (Plus a b)
  pair        :: (Language a, Language b) =>
                              g (Pair a b)
  binder      :: (Language a) => g (Binder a)
  occurrence  :: (Language a) => g (Occurrence a)
  datatype    :: (Language a) => Iso a b -> g b
  lit         :: g (Lit a)
```

Defining generic functions

Definition by cases over encoding:

```
class Generic g where
  unit        :: g Unit
  plus        :: (Language a, Language b) =>
                                 g (Plus a b)
  pair        :: (Language a, Language b) =>
                                 g (Pair a b)
  binder      :: (Language a) => g (Binder a)
  occurrence  :: (Language a) => g (Occurrence a)
  datatype    :: (Language a) => Iso a b -> g b
  lit         :: g (Lit a)
```

Example instance (free vars):

$$g\ a \simeq \text{Name} \to a \to \text{Bool}$$

A generic function applies to any language

`Language a` holds if `generic` can be inferred

```
class Language a where
  generic :: (Generic g) => g a
```

Infer by grabbing the right method of class `Generic`

```
instance Language Unit
  where generic = unit

instance (Language a, Language b) =>
         Language (Plus a b)
  where generic = plus

instance (Language a, Language b) =>
         Language (Pair a b)
  where generic = pair
  ...
```

Define by instantiating `Generic`

Example: variable free in term (removed type tags)

```
isFreeIn :: (Language a) => Name -> a -> Bool
isFreeIn = generic

instance Language a => Generic (Name -> a -> Bool)
where
 unit = \x Unit -> False
 plus = \x e -> case e of Inl l -> isFreeIn x l
                          Inr r -> isFreeIn x r
 pair = \x (Pair e1 e2) ->
                     isFreeIn x e1 || isFreeIn x e2
 binder = \x (Bind x’ e) -> x /= x’ && isFreeIn x e
 occurrence = \x t -> case t of Free x’ -> x == x’
                                Substituted t’ -> ...
 datatype iso = \x e -> isFreeIn x (fromData iso e)
 lit = \ _ _ -> False
```

The miracle of mutual recursion!

Define by instantiating `Generic`

Example: variable free in term (removed type tags)

```
isFreeIn :: (Language a) => Name -> a -> Bool
isFreeIn = generic

instance Language a => Generic (Name -> a -> Bool)
where
 unit = \x Unit -> False
 plus = \x e -> case e of Inl l -> isFreeIn x l
                          Inr r -> isFreeIn x r
 pair = \x (Pair e1 e2) ->
                     isFreeIn x e1 || isFreeIn x e2
 binder = \x (Bind x' e) -> x /= x' && isFreeIn x e
 occurrence = \x t -> case t of Free x' -> x == x'
                                Substituted t' -> ...
 datatype iso = \x e -> isFreeIn x (fromData iso e)
 lit = \ _ _ -> False
```

The miracle of mutual recursion!

Define by instantiating `Generic`

Example: variable free in term (removed type tags)

```
isFreeIn :: (Language a) => Name -> a -> Bool
isFreeIn = generic

instance Language a => Generic (Name -> a -> Bool)
where
 unit = \x Unit -> False
 plus = \x e -> case e of Inl l -> isFreeIn x l
                          Inr r -> isFreeIn x r
 pair = \x (Pair e1 e2) ->
                     isFreeIn x e1 || isFreeIn x e2
 binder = \x (Bind x’ e) -> x /= x’ && isFreeIn x e
 occurrence = \x t -> case t of Free x’ -> x == x’
                                Substituted t’ -> ...
 datatype iso = \x e -> isFreeIn x (fromData iso e)
 lit = \ _ _ -> False
```

The miracle of mutual recursion!

Define by instantiating `Generic`

Example: variable free in term (removed type tags)

```
isFreeIn :: (Language a) => Name -> a -> Bool
isFreeIn = generic

instance Language a => Generic (Name -> a -> Bool)
where
 unit = \x Unit -> False
 plus = \x e -> case e of Inl l -> isFreeIn x l
                          Inr r -> isFreeIn x r
 pair = \x (Pair e1 e2) ->
                     isFreeIn x e1 || isFreeIn x e2
 binder = \x (Bind x' e) -> x /= x' && isFreeIn x e
 occurrence = \x t -> case t of Free x' -> x == x'
                                Substituted t' -> ...
 datatype iso = \x e -> isFreeIn x (fromData iso e)
 lit = \ _ _ -> False
```

The miracle of mutual recursion!

Capture-avoiding substitution, generically

```
class Mapable e a where
  subst :: (Name :|-->: e) -> a -> a
```

Capture-avoiding substitution, generically

```
class Mapable e a where
  subst :: (Name :|-->: e) -> a -> a
```

Interesting case is the binder

```
instance (Language a, Language e, Mapable e a) =>
         Mapable e (Binder a)
where
  subst s@(x :|-->: y) (Bind x’ e)
    | x == x’ || not (x ‘isFreeIn‘ e) = Bind x’ e
    | x’ ‘isFreeIn‘ y =
         subst s $ rename (freeVars y) (Bind x’ e)
    | otherwise = Bind x’ (subst s e)
```

Capture-avoiding substitution, generically

```
class Mapable e a where
  subst :: (Name :|-->: e) -> a -> a
```

Interesting case is the binder

```
instance (Language a, Language e, Mapable e a) =>
         Mapable e (Binder a)
where
  subst s@(x :|-->: y) (Bind x' e)
    | x == x' || not (x `isFreeIn` e) = Bind x' e
    | x' `isFreeIn` y =
         subst s $ rename (freeVars y) (Bind x' e)
    | otherwise = Bind x' (subst s e)
```

Other cases trivial, e.g.:

```
(Mapable e a, Mapable e b) => Mapable e (Pair a b) where
  subst s (Pair a b) = Pair (subst s a) (subst s b)
```

Works for lots of languages

Will show you Type language in a few minutes

Works for lots of languages

Will show you `Type` language in a few minutes

Also works for compilation to 2D:

```
data M' x v m g = Return x
                | Consume x m
                | Let   Name v m g
                | Let2  Name v Name v m g
                | Split Name Name x m g
                | Call1 Name Name x m g
                | Call2 Name Name x x m
                | Case x (Name, m) (Name, m) g
```

Works for lots of languages

Will show you `Type` language in a few minutes

Also works for compilation to 2D:

```
data M’ x v m g = Return x
                | Consume x m
                | Let   Name v m g
                | Let2  Name v Name v m g
                | Split Name Name x m g
                | Call1 Name Name x m g
                | Call2 Name Name x x m
                | Case x (Name, m) (Name, m) g
```

Recursive knot tied two ways:

1. Full first-order function body
2. A-normal form

Review: Have achieved something

Claims:

1. Free variables and capture-avoiding substitution can be tedious (even wrong!)
2. I have written them generically
3. All a user need do is write isomorphisms

Review: Have achieved something

Claims:

1. Free variables and capture-avoiding substitution can be tedious (even wrong!)
2. I have written them generically
3. All a user need do is write isomorphisms

But writing isomorphisms is not so fun :-(

The solution, continued

Writing isomorphisms can be tedious

From Ralf's paper:

```
data Tree a = Leaf a | Fork (Tree a) (Tree a)

fromTree :: Tree a -> Plus a (Pair (Tree a) (Tree a))
fromTree (Leaf x)          =  Inl x
fromTree (Fork l r)        =  Inr (Pair l r)

toTree   :: Plus a (Pair (Tree a) (Tree a)) -> Tree a
toTree (Inl x)             =  Leaf x
toTree (Inr (Pair l r))    =  Fork l r
```

This style does not scale

Real languages have more constructors

Types (Peyton Jones et al. 2007, with variation):

```
data Type = ForAll [TyVar] Type
          | Fun    Type Type
          | TyApp  TyCon [Type]
          | TyVar  TyVar
```

(Real version has two more value constructors)

(Showed 2D target with eight value constructors)

Encoding uses sum-injection helpers

From `Type` to "`Language`":

```
fromTy (ForAll [] t)     = L.c1 t
fromTy (ForAll (a:tvs) t) =
  L.c2 (L.Bind (tyName a) (ForAll tvs t))
fromTy (Fun t1 t2)       = L.c3 (t1 ‘L.Pair‘ t2)
fromTy (TyApp c [])      = L.c4 (L.Lit c)
fromTy (TyApp c (t:ts)) = L.c5 (t ‘L.Pair‘ TyApp c ts)
fromTy (TyVar a)         = L.c6last (L.Free (tyName a))
```

Helpers:

```
L.c1     = Inl
L.c2     = Inr . Inl
L.c3     = Inr . Inr . Inl
L.c4     = Inr . Inr . Inr . Inl
L.c5     = Inr . Inr . Inr . Inr . Inl
L.c6last = Inr . Inr . Inr . Inr . Inr
```

Decoding uses just one helper

```
toTy =
 (\t -> ForAll [] t)                                          |+|
 (\(L.Bind x (ForAll xs t)) -> ForAll (tyvar x:xs) t)
                                                              |+|
 (\(L.Pair t1 t2) -> Fun t1 t2)                               |+|
 (\(L.Lit c)      -> TyApp c [])                              |+|
 (\(t ‘L.Pair‘ TyApp c ts) -> TyApp c (t:ts))                 |+|
 (\x -> case x of L.Free a -> TyVar (tyvar a)
                  L.Substituted t -> t)
```

Decoding uses just one helper

```
toTy =
 (\t -> ForAll [] t)                                          |+|
 (\(L.Bind x (ForAll xs t)) -> ForAll (tyvar x:xs) t)
                                                              |+|
 (\(L.Pair t1 t2) -> Fun t1 t2)                               |+|
 (\(L.Lit c)      -> TyApp c [])                              |+|
 (\(t ‘L.Pair‘ TyApp c ts) -> TyApp c (t:ts))                 |+|
 (\x -> case x of L.Free a -> TyVar (tyvar a)
                  L.Substituted t -> t)
```

Decoding uses just one helper

```
toTy =
 (\t -> ForAll [] t)                                    |+|
 (\(L.Bind x (ForAll xs t)) -> ForAll (tyvar x:xs) t)
                                                        |+|
 (\(L.Pair t1 t2) -> Fun t1 t2)                         |+|
 (\(L.Lit c)       -> TyApp c [])                       |+|
 (\(t ‘L.Pair‘ TyApp c ts) -> TyApp c (t:ts))           |+|
 (\x -> case x of L.Free a -> TyVar (tyvar a)
                  L.Substituted t -> t)
```

Helper |+| is either:

```
  (|+|) :: (a -> c) -> (b -> c) -> (L.Plus a b) -> c
  (|+|) f1 f2 (Inl x) = f1 x
  (|+|) f1 f2 (Inr x) = f2 x
```

Types are less scary in infix

```
type (:+:) = L.Plus
type (:*:) = L.Pair

fromType :: Type -> Type                :+:
                    L.Binder Type       :+:
                    (Type :*: Type)     :+:
                    L.Lit TyCon         :+:
                    (Type :*: Type)     :+:
                    L.Occurrence Type

toType :: Type                :+:
          L.Binder Type       :+:
          (Type :*: Type)     :+:
          L.Lit TyCon         :+:
          (Type :*: Type)     :+:
          L.Occurrence Type
          -> Type
```

With isomorphisms in hand, all is easy

Using the isomorphisms:

```
instance L.Language Type where
  generic = L.datatype (L.Iso fromType toType)

instance L.Mapable Type Type where
  subst s = toType . L.subst s . fromType
```

Voilà! Free variables, capture-avoiding substitution

The next problem

To solve (this week?)

Perhaps something similar for a type checker. . .

To solve (this week?)

Perhaps something similar for a type checker. . .

Unlike name binding, no near-universal type system

To solve (this week?)

Perhaps something similar for a type checker. . .

Unlike name binding, no near-universal type system

But one is promising for many applications:

F_ω + fixed point + GADTs

First steps

New type classes and instances:

- `LanguageOver e x`

  — Type e is a language with names x

- Useful instances:

```
instance LanguageOver Type TyVar -- as above
instance LanguageOver Term Var   -- by analogy
```

First steps

New type classes and instances:

- `LanguageOver e x`

  — Type e is a language with names x

- Useful instances:

```
instance LanguageOver Type TyVar -- as above
instance LanguageOver Term Var   -- by analogy
instance LanguageOver Term TyVar -- don't overlook
```

The object of the exercise

Signature of a generic type checker

```
data Error a = ... -- error monad

class FwaTypeable e t x a where
  typeOf :: ( LanguageOver e x
            , LanguageOver t a
            , LanguageOver e a
            ) =>
            Map a Kind -> Map x t -> e -> Error t
```

Do join me

I've been having fun

- Results are entertaining
- Might be useful