# Generic Unification Generalized

or

# Datatypes parameterized over Typerecords

**Magnus Carlsson**

**Oregon Graduate Institute**

# A programming exercise combining:

- Generic Unification via Two-Level Types and Parameterized Modules
- Using Parameterized Signatures to Express Modular Structure
- Monadic Encapsulation of Effects: A Revised Approach
- Encoding Records using Functions
- Cayenne — a language with dependent types

## A programming exercise combining:

- Generic Unification via Two-Level Types and Parameterized Modules (Tim Sheard, functional pearl, ICFP'01)
- Using Parameterized Signatures to Express Modular Structure (Mark Jones, POPL'96)
- Monadic Encapsulation of Effects: A Revised Approach (Eugenio Moggi and Amr Sabry, JFP 2001)
- Encoding Records using Functions (Thorsten Altenkirch, yesterday)
- Cayenne — a language with dependent types (Lennart Augustsson, ICFP'98)

# The Practical Programming side

## Given:

An efficient unification algorithm in Haskell (Tim)

- Unification parameterized over the term datatype.
- Uses the ST-monad for mimicking classical imperative unification.
- Each term datatype comes with a record of suitable operations, such as doing a monadic traversal.
- The term datatypes are not recursively defined, instead knot-tying is done **after** adding structure (variables) to the terms.

# Generic Unification

## A term datatype for commands

```
data C
  = If E C C
  | Skip
```

## Untying the Knot

Parameterize over the recursive occurrences

```
data C x
  = If E x x
  | Skip
```

# Structure and Generic Terms

Add variables and tie the knot

```
data GT s r
  = S (s (GT s r))              -- structure
  | Var (r (Maybe (GT s r)))    -- mutable variables
  | GenVar Int                  -- generic variables
```

- `s` is the structure datatype
- `r` is the type of some ST-monad variable

Example: `GT C r` is the type of command terms with variables.

# Operations over term types

Given a term datatype `s`, we need a record of **structural operations** over `s`:

```
record StructOp s =
  map   :: (x -> y) -> s x -> s y
  acc   :: (x -> y -> y) -> s x -> y -> y
  seq   :: Monad m => s (m x) -> m (s x)
  match :: s x -> s x -> Maybe [(x,x)]
```

(**record** defines a record type by giving the types of its selectors, universally quantified over free variables.)

# The structural operations for `C`

```
structOpC :: StructOp C
structOpC = record
  map :: (x -> y) -> C x -> C y
  map f (If e c1 c2) = If e (f c1) (f c2)
  map f Skip = Skip

  acc :: (x -> y -> y) -> C x -> y -> y
  acc f (If e c1 c2) y = f c1 (f c2 y)
  acc f Skip          y = f y
  ...
```

# The structural operations for `C`

```
...
seq :: Monad m => C (m x) -> m (C x)
seq (If e c1 c2) = return If e `ap` seq c1 `ap` seq c2
seq Skip         = return Skip

match :: C x -> C x -> Maybe [(x,x)]
match f (If e1 c11 c12) (If e2 c21 c22) | e1 == e2
                                        = Just [(c11,c21), (c12,c22)]
match f Skip Skip = Just []
match f _    _    = Nothing
```

# The ST-monad generalized

Define a record type of **monadic reference operations** over a reference type `t` and a monad `m`:

```
record Monad m => RefOp r m =
  writeVar :: r x -> x -> m ()
  readVar  :: r x -> m x
  newVar   :: x -> m (r x)
  sameVar  :: r x -> r x -> Bool
```

# The unification algorithm

```
unify :: ErrorMonad m => RefOp r m -> StructOp s ->
         GT s r -> GT s r -> m ()
unify rop sop (Var r1) (Var r2) | rop.sameVar r1 t2 = return ()
                                | otherwise = sop.write r1 (Just r2)
unify rop sop (VarType r1) t2 = sop.write r1 t2
unify rop sop t1 (VarType r2) = sop.write r2 t1
unify rop sop (GenVar n) (GenVar m) | n == m = return ()
unify rop sop (S s1) (S s2) =
     case sop.match s1 s2 of
          Nothing -> fail "unify"
          Just ps -> mapM_ (uncurry (unify rop sop)) ps
unify rop sop _ _ = fail "unify"

-- Recall mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
```

# Dependent types?

# Mutually defined term types?

```
data C =
   If E C C
 | Skip

data E =
    Do C
 | ...
```

# Mutually defined term types?

```
data C c e =
   If e c c
 | Skip

data E c e =
    Do c
 | ...
```

Breaks our generic unification!

# Typerecords

Idea: Stuff the recursive types in one single **record** parameter (Mark Jones)

```
data Cmd x =
   If x.expr x.cmd x.cmd
 | Skip

data Expr x =
    Do x.cmd
 | ...
```

Tying the knot:

```
typerecord Syntax =
     cmd  = Cmd Syntax
     expr = Expr Syntax
```

# Encoding typerecords in Cayenne

- Define a datatype encoding the labels in the record

```
data SyntaxTag = cmd | expr
```

- Encode typerecords as functions from the tags to types

```
Ex :: SyntaxTag -> #
Ex (cmd)  = Int
Ex (expr) = Bool
```

# Encoding typerecords in Cayenne

In general:

- Tag types

  ```
  TagT = #
  ```

- Typerecords

  ```
  TRecord :: TagT -> #1
  TRecord T = T -> #
  ```

Our example:

```
Ex :: TRecord SyntaxTag
```

# Syntax using the encoding

```
Cmd :: TRecord SyntaxTag -> #
Cmd x = data If (x expr) (x cmd) (x cmd)
           | Skip

Expr :: TRecord SyntaxTag -> #
Expr x = data Do (x cmd)
            | ...
```

# Syntax using the encoding

Tying the knot:

```
Syntax :: TRecord SyntaxTag
Syntax t = SyntaxF t Syntax

SyntaxF :: SyntaxTag -> TRecord SyntaxTag -> #
SyntaxF (cmd)  = Cmd
SyntaxF (expr) = Expr

ex :: Cmd Syntax
ex = If (Do Skip) Skip Skip
```

# Defining functions over `Syntax`

```
showCmd :: (T :: TRecord SyntaxTag) |->
           (? -> String) ->
            Cmd T -> String
showCmd shf (If e c1 c2) = "If (" ++ shf e ++
                            ") (" ++ shf c1 ++
                            ") (" ++ shf c2 ++ ")"
showCmd shf (Skip) = "Skip"
```

What is the type of `shf`?

# Defining functions over `Syntax`

```
showCmd :: (T :: TRecord SyntaxTag) |->
           (? -> String) ->
            Cmd T -> String
showCmd shf (If e c1 c2) = "If (" ++ shf e ++
                             ") (" ++ shf c1 ++
                             ") (" ++ shf c2 ++ ")"
showCmd shf (Skip) = "Skip"
```

A **record** of functions:

```
shf :: { cmd  = T.cmd  -> String;
         expr = T.expr -> String }
```

A **record** of functions:

```
shf :: { cmd  = T.cmd  -> String;
         expr = T.expr -> String }
```

Using our encoding:

```
shf :: ((t :: SyntaxTag) -> T t -> String)
```

Back to our show-function:

```
showCmd :: (T :: TRecord SyntaxTag) |->
           ((t :: SyntaxTag) -> T t -> String) ->
            Cmd T -> String
showCmd shf (If e c1 c2) = "If (" ++ shf expr e ++
                            ") (" ++ shf cmd c1 ++
                            ") (" ++ shf cmd c2 ++ ")"
showCmd shf (Skip) = "Skip"
```

Using **hidden parameters** for poor man's overloading

```
showCmd :: (T :: TRecord SyntaxTag) |->
           ((t :: SyntaxTag) -> T t -> String) ->
            Cmd T -> String
showCmd shf c =

  let p :: (t :: SyntaxTag) |-> T t -> String
      p |t = shf t

  in case c of
       (If e c1 c2) -> "If (" ++ p e ++
                         ") (" ++ p c1 ++
                         ") (" ++ p c2 ++ ")"
       (Skip) -> "Skip"
```

(Cf. overloading in Mark Shields' system of type-indexed rows)

# The show-function for `Syntax`

```
showCmd ::  (T :: TRecord SyntaxTag) ->
            ((t :: SyntaxTag) -> T t -> String) ->
            Cmd T -> String
showExpr :: (T :: TRecord SyntaxTag) ->
            ((t :: SyntaxTag) -> T t -> String) ->
            Expr T -> String
```

## The show-function for `Syntax`

```
showCmd ::  (T :: TRecord SyntaxTag) ->
            ((t :: SyntaxTag) -> T t -> String) ->
            Cmd T -> String
showExpr :: (T :: TRecord SyntaxTag) ->
            ((t :: SyntaxTag) -> T t -> String) ->
            Expr T -> String

showSyntax :: (T :: TRecord SyntaxTag) ->
               ((t :: SyntaxTag) -> T t -> String) ->
               (t :: SyntaxTag) -> SyntaxF t T ->
               String
```

Recall

```
SyntaxF :: SyntaxTag -> TRecord SyntaxTag -> #
SyntaxF (cmd) = Cmd
SyntaxF (expr) = Expr
```

```
showSyntax :: (T :: TRecord SyntaxTag) ->
              ((t :: SyntaxTag) -> T t -> String) ->
              (t :: SyntaxTag) -> SyntaxF t T ->
              String

showSyntax T shf (cmd) = showCmd T shf
showSyntax T shf (expr) = showExpr T shf
```

# Generalizing the generic unification

Recall the StructOp type:

```
record StructOp s =
  map   :: (x -> y) -> s x -> s y
  acc   :: (x -> y -> y) -> s x -> y -> y
  seq   :: Monad m => s (m x) -> m (s x)
  match :: s x -> s x -> Maybe [(x,x)]
```

We need to generalize this to typerecords.

# StructOp in Cayenne

```
StructOp :: (T :: TagT) -> (S :: TRecord T -> #) -> #1.0
StructOp T S = sig
  map ::   (A :: TRecord T) |-> (B :: TRecord T) |->
           ((t::T) -> (A t -> B t)) -> S A -> S B

  acc ::   (A :: TRecord T) |-> (B :: #) ->
           ((t::T) -> A t -> B -> B) -> S A -> B -> B
  ...
```

# StructOp in Cayenne, cont'd

```
StructOp :: (T :: TagT) -> (S :: TRecord T -> #) -> #1.0
StructOp T S = sig
  ...
  seq ::    (M :: # -> #) -> (m :: Monad M) ->
            (A :: TRecord T) |->
            S (M · A) -> M (S A)
```

Note: `M · A = \(t::T) -> M (A t)`

```
  match :: (A :: TRecord T) |->
            S A -> S A ->
            Maybe ((t :: T) -> [(A t, A t)])
```

## `StructOp` in Cayenne

```
seq ::   (M :: # -> #) -> (m :: Monad M) ->
         (A :: TRecord T) |->
         S (M · A) -> M (S A)
```

## StructOp for Cmd

```
structOpCmd :: StructOp SyntaxTag Cmd
structOpCmd = struct
  map f' c = let f = hide f'
             in case c of
                  (If i t e)   -> If b (f i) (f t) (f e)
                  (Skip)       -> Skip b

  acc y f' c v = let f = hide f'
                 in case c of
                      (If i t e)   -> f i (f t (f e v))
                      (Skip)       -> v
  ...
```

# StructOp for Cmd, cont'd

```
structOpCmd :: StructOp SyntaxTag Cmd
structOpCmd = struct
  ...
  seq M m c =
     let (<<) = ap M m --  Monadic application
     in case c of
           (If i t e)  -> (m.return If) << i << t << e
           (Skip)      -> m.return Skip
  ...
```

# `StructOp` for Cmd, cont'd

```
structOpCmd :: StructOp SyntaxTag Cmd
structOpCmd = struct
  ...
  match (If i1 t1 e1) (If i2 t2 e2) = Just (mkr [(t1,t2),(e1,e2)]
                                                [(i1,i2)])
  match (Skip) (Skip)               = Just (mkr [] [])
  match _ _                         = Nothing


  mkr l1 l2 (cmd)  = l1
  mkr l1 l2 (expr) = l2
```

# Structure and Generic Terms

Recall

```
data GT s r
  = S (s (GT s r))                 -- structure
  | Var (r (Maybe (GT s r)))       -- mutable variables
  | GenVar Int                     -- generic variables
```

How do we adapt it to `TRecords`?

# Typerecord families

Recall (again)

```
SyntaxF :: SyntaxTag -> TRecord SyntaxTag -> #
SyntaxF (cmd)  = Cmd
SyntaxF (expr) = Expr
```

- Aside: A typerecord parameterised over a typerecord

  ```
  flip SyntaxF :: TRecord SyntaxTag -> SyntaxTag -> #
               :: TRecord SyntaxTag -> TRecord SyntaxTag
  ```

Abstracting from the tag type, we get the pattern

```
(T -> TRecord T -> #)
```

```
RFamily :: TagT -> #1
RFamily T = T -> TRecord T -> #
```

# Typerecord families

```
RFamily :: TagT -> #1
RFamily T = T -> TRecord T -> #

SyntaxF :: SyntaxTag -> TRecord SyntaxTag -> #
SyntaxF :: RFamily SyntaxTag
SyntaxF (cmd)   = Cmd
SyntaxF (expr) = Expr
```

# The `GT` type using typerecord families

Haskell:

```
data GT s r
  = S (s (GT s r))                -- structure
  | Var (r (Maybe (GT s r)))      -- mutable variables
  | GenVar Int                    -- generic variables
```

Cayenne:

```
GT :: (T :: TagT) -> (RFamily T) -> RefT -> TRecord T
GT T F R t = data S (F t (GT T F R))
                | Var (R (Maybe (GT T F R t)))
                | GenVar Int
```

where

```
  RefT = (# -> #)
```

# Example

```
SyntaxGT :: RefT -> TRecord SyntaxTag
SyntaxGT = GT SyntaxTag SyntaxF

ex :: (R :: RefT) -> SyntaxGT R cmd
ex R = S (If (S (I 1)) (S Skip)
                       (S Skip))
```

# The ST monad revised

Encapsulating state-manipulating computations

```
type ST s a;
type Var s a;
newVar    :: a -> ST s (Var s a)
readVar   :: Var s a -> ST s a
writeVar  :: Var s a -> a -> ST s ()
sameVar   :: Var s a -> Var s b -> Bool

runST     :: (∀s.ST s a) -> a
```

`s` is the state region parameter: it isolates distinct computations.

(John Launchbury and Simon Peyton Jones)

# Factor out the state region parameter

(Eugenio and Amr)

```
ST :: #1.0
ST = sig
    type Var _
    type M _
    m        :: System$Monad M
    newVar   :: (A :: #) |-> A -> M (Var A)
    readVar  :: (A :: #) |-> Var A -> M A
    writeVar :: (A :: #) |-> Var A -> A -> M Unit
    sameVar  :: (A :: #) |-> (B :: #) |-> Var A -> Var B -> Bool
```

# Factor out the state region parameter

(Eugenio and Amr)

```
ST :: #1.0
ST = sig
    type Var _
    type M _
    m        :: System$Monad M
    newVar   :: (A :: #) |-> A -> M (Var A)
    readVar  :: (A :: #) |-> Var A -> M A
    writeVar :: (A :: #) |-> Var A -> A -> M Unit
    sameVar  :: (A :: #) |-> (B :: #) |-> Var A -> Var B -> Bool
```

```
runST :: (A :: #) |-> ((s :: ST) -> s.M A) -> A
```

In `runST`, the computation must accept **any** implementation of the `ST` record.

# Final remarks

- Practical programming with dependent types is a blast. Some favorite features of Cayenne (in contrast to Haskell):
  - Simple but powerful module system (records).
  - Full abstraction power over types.
  - No need to fight against a complicated class system.
- The type checker in Cayenne (in particular, the elaboration step for finding hidden arguments) would benefit from improvements.