# The Implementation of Idris 2

## Part 4: Conversion and Unification

Edwin Brady (ecb10@st-andrews.ac.uk)
University of St Andrews, Scotland
@edwinbrady

SPLV, 20th August 2020

# Today's Lecture

- Completing Elaboration
  - Conversion and Unification
- References, related work

# Recap: Elaborating TTImp

**Checking a RawImp (`TTImp.Elab.Term`)**

```
checkTerm : {vars : _} ->
            {auto c : Ref Ctxt Defs} ->
            Env Term vars ->          -- the environment
            RawImp ->                 -- term to check
            Maybe (Glued vars) ->     -- expected type
            Core (Term vars, Glued vars)
```

A `Glued` is a (lazily calculated) pair of a `Term` and a value, `NF`

# Defining Values

## Values (Core.Value)

```
data NF : List Name -> Type where
     NDCon : Name -> (tag : Int) -> (arity : Nat) ->
             List (Closure vars) -> NF vars
     NTCon : Name -> (tag : Int) -> (arity : Nat) ->
             List (Closure vars) -> NF vars
     NBind : (x : Name) -> Binder (NF vars) ->
             (Defs -> Closure vars -> Core (NF vars)) ->
             NF vars
     NApp  : NHead vars -> List (Closure vars) -> NF vars
     NType : NF vars
     NErased : NF vars
```

# Defining Values

## "Stuck" applications (Core.Value)

```
data NHead : List Name -> Type where
     NLocal : (idx : Nat) -> (0 p : IsVar name idx vars) ->
              NHead vars
     NRef   : NameType -> Name -> NHead vars
     NMeta  : Name -> List (Closure vars) -> NHead vars
```

## Unevaluated arguments (Core.Value)

```
data Closure : List Name -> Type where
     MkClosure : LocalEnv free vars ->
                 Env Term free ->
                 Term (vars ++ free) ->
                 Closure free
```

# Conversion

## Checking conversion (Core.Normalise)

```
interface Convert (tm : List Name -> Type) where
  convert : Defs -> Env Term vars ->
            tm vars -> tm vars -> Core Bool
  convGen : Ref QVar Int ->
            Defs -> Env Term vars ->
            tm vars -> tm vars -> Core Bool

  convert defs env tm tm'
      = do q <- newRef QVar 0
           convGen q defs env tm tm'

Convert NF where ...
Convert Term where ...
Convert Closure where ...
```

# Elaboration

**Applying the conversion rule (`TTImp.Elab.Term`)**

```
checkExp : {auto c : Ref Ctxt Defs} ->
           Env Term vars ->
           (term : Term vars) ->
           (got : Glued vars) ->
           (expected : Maybe (Glued vars)) ->
           Core (Term vars, Glued vars)
```

# Elaboration

## Applying the conversion rule (TTImp.Elab.Term, TinyIdris-v1)

```
checkExp env term got Nothing = pure (term, got)
checkExp env term got (Just exp)
    = do defs <- get Ctxt
         True <- convert defs env !(getNF got) !(getNF exp)
              | _ => throw (CantConvert env
                                            !(getTerm got)
                                            !(getTerm exp))
         pure (term, exp)
```

# Conversion

We check conversion on *values*, not *terms*:

## Conversion of Terms

```
Convert Term where
  convGen q defs env x y
     = convGen q defs env
                 !(nf defs env x)
                 !(nf defs env y)

Convert Closure where
  convGen q defs env x y
     = convGen q defs env
                 !(evalClosure defs x)
                 !(evalClosure defs y)
```

# Conversion

We check conversion on *values*, not *terms*:

## Conversion of Terms

```
Convert Term where
  convGen q defs env x y
     = convGen q defs env
                 !(nf defs env x)
                 !(nf defs env y)

Convert Closure where
  convGen q defs env x y
     = convGen q defs env
                 !(evalClosure defs x)
                 !(evalClosure defs y)
```

(Question: are there optimisations possible here?)

# Conversion: Constructors

## Conversion checking constructors

```
allConv : Ref QVar Int -> Defs -> Env Term vars ->
          List (Closure vars) -> List (Closure vars) ->
          Core Bool

convGen q defs env
        (NDCon nm tag _ args)
        (NDCon nm' tag' _ args')
    = if tag == tag'
         then allConv q defs env args args'
         else pure False
```

# Conversion: Binders

## Conversion checking binders

```
convGen q defs env
            (NBind x b sc)
            (NBind x' b' sc')
        = do let c = ?help
             if !(convBinders q defs env b b')
                then do bsc <- sc defs c
                        bsc' <- sc' defs c
                        convGen q defs env bsc bsc'
                else pure False
```

# Conversion: Binders

## Conversion checking binders

```
convGen q defs env
            (NBind x b sc)
            (NBind x' b' sc')
        = do var <- genName "conv"
             let c = MkClosure [] env (Ref Bound var)
             if !(convBinders q defs env b b')
                then do bsc <- sc defs c
                        bsc' <- sc' defs c
                        convGen q defs env bsc bsc'
                else pure False
```

# Extending to Support Unification

## Applying the conversion rule (`Core.Unify`)

```
interface Unify (tm : List Name -> Type) where
  unify : {auto c : Ref Ctxt Defs} ->
          {auto u : Ref UST UState} ->
          Env Term vars ->
          tm vars -> tm vars ->
          Core UnifyResult
```

# Extending to Support Unification

## Applying the conversion rule (`Core.Unify`)

```
interface Unify (tm : List Name -> Type) where
  unify : {auto c : Ref Ctxt Defs} ->
          {auto u : Ref UST UState} ->
          Env Term vars ->
          tm vars -> tm vars ->
          Core UnifyResult
```

Similar to convert, but:

- Maintains a *unification state* `UState`
- Returns a `UnifyResult`:
  - Essentially: *Yes*, *No*, or *Yes but...*

# Unification results: Yes, but...

## Well-typed terms

```
data Term : Vect k Ty -> Ty -> Type where
     Val : (x : interpTy a) -> Term gam a
     ...

x : Term [] TyNat
x = Val 94
```

# Unification results: Yes, but. . .

### Well-typed terms

```
data Term : Vect k Ty -> Ty -> Type where
     Val : (x : interpTy a) -> Term gam a
     ...

x : Term [] TyNat
x = Val {gam = ?gam_meta} {a = ?a_meta} 94
```

# Unification results: Yes, but...

## Well-typed terms

```
data Term : Vect k Ty -> Ty -> Type where
     Val : (x : interpTy a) -> Term gam a
     ...

x : Term [] TyNat
x = Val {gam = ?gam_meta} {a = ?a_meta} 94
```

Generates constraints:

- `interpTy ?a_meta = Nat`

# Unification results: Yes, but. . .

## Well-typed terms

```
data Term : Vect k Ty -> Ty -> Type where
     Val : (x : interpTy a) -> Term gam a
     ...

x : Term [] TyNat
x = Val {gam = ?gam_meta} {a = ?a_meta} 94
```

Generates constraints:

- `interpTy ?a_meta = Nat`
- `Term ?gam_meta ?a_meta = Term [] TyNat`

# Unification State

**Definitions (Core.Context)**

```
data Def : Type where
     ...
     Hole : Def
     Guess : (guess : Term []) ->
             (constraints : List Int) -> Def
```

UState contains:

- Names of unsolved *holes* and *guesses*
- A global list of *constraints*, referred to by an Int

# Holes and Guesses

## Generating terms for Holes and Guesses

```
-- Create a metavariable applied to an environment
newMeta : {auto c : Ref Ctxt Defs} ->
          {auto u : Ref UST UState} ->
          Env Term vars -> Name -> (ty : Term vars) ->
          Core (Term vars)

-- Create a constant, guarded by constraints
newConstant : {auto c : Ref Ctxt Defs} ->
              {auto u : Ref UST UState} ->
              Env Term vars ->
              (tm : Term vars) -> (ty : Term vars) ->
              (constrs : List Int) ->
              Core (Term vars)
```

# Unification Constraints

## Constraints (Core.UnifyState)

```
data Constraint : Type where
     MkConstraint : Env Term vars ->
                    (x : Term vars) ->
                    (y : Term vars) ->
                    Constraint
     MkSeqConstraint : Env Term vars ->
                       (xs : List (Term vars)) ->
                       (ys : List (Term vars)) ->
                       Constraint
```

# Unification Constraints

## Constraints (Core.UnifyState)

```
data Constraint : Type where
     MkConstraint : Env Term vars ->
                    (x : Term vars) ->
                    (y : Term vars) ->
                    Constraint
     MkSeqConstraint : Env Term vars ->
                       (xs : List (Term vars)) ->
                       (ys : List (Term vars)) ->
                       Constraint
```

## Solving Constraints (Core.Unify)

```
solveConstraints : {auto c : Ref Ctxt Defs} ->
                   {auto u : Ref UST UState} ->
                   Core ()
```

# Unification: Constructors

## Unifying constructor applications

```
unify : Env Term vars -> NF vars -> NF vars ->
        Core UnifyResult

unify env nx@(NDCon n t a args) ny@(NDCon n' t' a' args')
    = if t == t'
         then unifyArgs env args args'
         else convertError env nx ny
```

# Unification: Blocked applications

## Unifying blocked applications

```
unifyApp :
      {auto c : Ref Ctxt Defs} ->
      {auto u : Ref UST UState} ->
      Env Term vars ->
      NHead vars ->                -- blocked application head
      List (Closure vars) ->  -- blocked arguments
      NF vars ->                   -- value we're unifying with
      Core UnifyResult
```

- e.g. Unifying `plus x Z` with `Z`
  - head is `plus`
- Unifying `?var x` with `x`
  - head is `?var x`, blocked arguments empty

# Unification: Metavariables

**Unifying metavariables (see `Core.Unify`)**

```
unifyApp env (NMeta n margs) fargs tmnf = ...
```

- Checks that the arguments margs, fargs are *distinct variables*
  - "Pattern condition"
- If so, tries to update the definition of n to:
  - `n margs fargs = tmnf`
  - Will only succeed if variables in `tmnf` occur in `margs`, `fargs`
  - Indexing by `vars` keeps us right!

# Unification: Blocked applications

## Other blocked applications

```
unifyApp env f args tm
    = do defs <- get Ctxt
         if !(convert defs env (NApp f args) tm)
            then pure success
            else postpone env (NApp f args) tm
```

# Updating the conversion rule

## Applying the conversion rule (TTImp.Elab.Term)

```
checkExp : {auto c : Ref Ctxt Defs} ->
           {auto u : Ref UST UState} ->
           Env Term vars ->
           (term : Term vars) ->
           (got : Glued vars) ->
           (expected : Maybe (Glued vars)) ->
           Core (Term vars, Glued vars)
```

## Applying the conversion rule (`TTImp.Elab.Term`, TinyIdris-v2)

```
checkExp env term got Nothing = pure (term, got)
checkExp env term got (Just exp)
   = do defs <- get Ctxt
        ures <- unify env !(getNF got) !(getNF exp)
        case constraints ures of
             [] => do when (holesSolved ures)
                              solveConstraints
                      pure (term, exp)
             cs => do cty <- getTerm exp
                      ctm <- newConstant env term cty cs
                      pure (ctm, got)
```

# Generating Metavariables

## Elaborating Implicits (TTImp.Elab.Term)

```
checkTerm env Implicit (Just exp)
  = do expty <- getTerm exp
       nm <- genName "_"
       metaval <- newMeta env nm expty Hole
       pure (metaval, exp)
```

## Implicit Arguments: Sketch

What about implicit arguments in types? e.g.

```
reverse : {a: Type} -> List a-> List a
```

- When checking applications:
  - Look at the *function*'s type
  - If there's an implicit argument, create a metavariable for it
  - Continue checking the application

# Binding Implicits: Sketch

Rather than `IPatVar`, we have (in full Idris 2):

```
data RawImp : Type where
     ...
     IBindVar : Name -> RawImp
```

When `checkTerm` encounters an `IBindVar`:

- Note the *name* and *expected type*
- Create a "pattern" metavariable for it

# Binding Implicits: Sketch

Rather than IPatVar, we have (in full Idris 2):

```
data RawImp : Type where
     ...
     IBindVar : Name -> RawImp
```

When checkTerm encounters an IBindVar:

- Note the *name* and *expected type*
- Create a "pattern" metavariable for it

At the end of elaboration, pat bind all the names we noted (and any names which depend on them).

- This involves sorting variables into dependency order
- No (okay, not much) problem with local variables due to vars in the type!

# References (a selection)

On *Quantitative Type Theory*:

- *I Got Plenty o' Nuttin'* — Conor McBride, 2016
- *The Syntax and Semantics of Quantitative Type Theory* — Robert Atkey, 2018

On *typechecking dependent types*:

- *An algorithm for type-checking dependent types* — Thierry Coquand, 1996
- *Towards a Practical Programming Language Based on Dependent Type Theory* — Ulf Norell (thesis) 2007
- *A tutorial implementation of a dependently typed lambda calculus* — Andres Löh, Conor McBride, Wouter Swiersta, 2010

# References (a selection)

More on *typechecking*:

- Epigram Reloaded: A Standalone Typechecker for ETT — James Chapman, Thorsten Altenkirch, Conor McBride, 2005
- Type checking in the presence of meta-variables — Ulf Norell, Catarina Coquand, 2007
- Type-and-Scope Safe Programs and their Proofs — Guillaume Allais, James Chapman, Conor McBirde, James McKinna, 2017

On *Unification*:

- *Unification Under a Mixed Prefix* — Dale Miller, 1992
- *Higher Order Constraint Simplification in Dependent Type Theory* — Jason Reed, 2009
- *Type Inference, Haskell and Dependent Types* — Adam Gundry (thesis) 2013

# References (a selection)

Other methods for *implementation*:

- Fast Elaboration for Dependent Type Theories (talk) — András Kovács, 2019
- Bidirectional Typing — J Dunfield, Neel Krishnaswami, 2019
- Checking Dependent Types with Normalization by Evaluation: A Tutorial — David Christiansen `http://davidchristiansen.dk/tutorials/nbe/`