Type Driven Development with Idris

Lecture 4: Towards an Implementation of Idris in Idris

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

MGS, April 12th 2018

Idris

In today's talk:

- The core language of Idris, *TT*
- Implementation challenges
  - Elaboration, high level constructs
- A new implementation
  - Progress so far
  - Term representation and *unification*

The Core Language, TT

High level Idris programs *elaborate* to a core language, *TT*:

- TT allows *only* data declarations and top level pattern matching definitions
- Limited syntax:
  - Variables, application, binders (λ, $\forall$, `let`, patterns), constants
- All terms *fully explicit*
- Advantage: type checker is small ($\approx$ 500 lines) so not many lines to go wrong
- Challenge: how to build TT programs from Idris programs?

Elaboration Example

Vectors, high level Idris

```
data Vect : Nat -> Type -> Type where
     Nil  : Vect Z a
     (::) : a -> Vect k a -> Vect (S k) a
```

Elaboration Example

Vectors, high level Idris

```
data Vect : Nat -> Type -> Type where
     Nil  : Vect Z a
     (::) : a -> Vect k a -> Vect (S k) a
```

Vectors, TT

```
Nil  : (a : Type) -> Vect a Z
(::) : (a : Type) -> (k : Nat) ->
         a -> Vect k a -> Vect (S k) a
```

sicsa*

Elaboration Example

Vectors, high level Idris

```
data Vect : Nat -> Type -> Type where
     Nil  : Vect Z a
     (::) : a -> Vect k a -> Vect (S k) a
```

Vectors, TT

```
Nil  : (a : Type) -> Vect a Z
(::) : (a : Type) -> (k : Nat) ->
         a -> Vect k a -> Vect (S k) a
```

Example

```
(::) Char (S Z) 'a' ((::) Char Z 'b' (Nil Char))
      -- ['a', 'b']
```

Elaboration Example

Pairwise addition, high level Idris

```
vAdd : Num a => Vect n a -> Vect n a -> Vect n a
vAdd []        []        = []
vAdd (x :: xs) (y :: ys) = x + y :: vAdd xs ys
```

Elaboration Example

Step 1: Add implicit arguments

```
vAdd : (a : _) -> (n : _) ->
       (Num a) -> Vect n a -> Vect n a -> Vect n a
vAdd _ _ c (Nil _) (Nil _) = Nil _
vAdd _ _ c ((::) _ _ x xs) ((::) _ _ y ys)
       = (::) _ _ ((+) _ x y) (vAdd _ _ _ xs ys)
```

sicsa*

Elaboration Example

Step 2: Solve implicit arguments

```
vAdd : (a : Type) -> (n : Nat) ->
       (Num a) -> Vect n a -> Vect n a -> Vect n a
vAdd a Z c (Nil a) (Nil a) = Nil a
vAdd a (S k) c ((::) a k x xs) ((::) a k y ys)
       = (::) a k ((+) c x y) (vAdd a k c xs ys)
```

Elaboration Example

Step 3: Make pattern bindings explicit

```
vAdd : (a : Type) -> (n : Nat) ->
       (Num a) -> Vect n a -> Vect n a -> Vect n a
pat a : Type, c : Num a .
  vAdd a Z c (Nil a) (Nil a) = Nil a
pat a : Type, k : Nat, c : Num a .
pat x : a, xs : Vect k a, y : a, ys : Vect k a .
  vAdd a (S k) c ((::) a k x xs) ((::) a k y ys)
       = (::) a k ((+) c x y) (vAdd a k c xs ys)
```

Elaborating Declarations

Top level declarations

```
f : S1 -> ... -> Sn -> T
f x1 ... xn = e
```

Elaborating Declarations

Top level declarations

```
f : S1 -> ... -> Sn -> T
f x1 ... xn = e
```

- Elaborate the type, and add `f` to the context
- Elaborate the lhs
  - Any names beginning with a lower case character are assumed to be *pattern* variables
- Elaborate the rhs *in the scope of the pattern variables from the lhs*
- Check that the lhs and rhs have the same type

Elaborating Interfaces

High level Idris

```
interface Show a where
    show : a -> String

implementation Show Nat where
    show Z = "Z"
    show (S k) = "S (" ++ show k ++ ")"
```

Elaborating Interfaces

Elaborated TT

```
data Show : (a : Set) -> Set where
    ShowImpl : (show : a -> String) -> Show a

show : (Show a) -> a -> String
show (ShowImpl show') x = show' x

implShowNat : Show Nat
implShowNat = ShowImpl showNat where
    showNat : Nat -> String
    showNat Z = "Z"
    showNat (S k) = "S (" ++ show implShowNat k ++ ")"
```

Elaborating Interfaces

Interfaces constraints are a special kind of implicit argument

- Ordinary implicit arguments solved by *unification*
- Constraint arguments solved by a *tactic*
    - Looks for a local solution first
    - Then looks for globally defined implementations
    - May give rise to further constraints

Current Implementation: Problems

- Efficiency problems
  - *Tactic-based* elaborator
  - Unconstrained *overloading*
  - Too much *evaluation*

Current Implementation: Problems

- Efficiency problems
  - *Tactic-based* elaborator
  - Unconstrained *overloading*
  - Too much *evaluation*
- Inference limitations
  - `case` construct
  - `where` blocks

Towards a new implementation: Blodwen

- `https://github.com/edwinb/Blodwen`
- Current status
  - Core language *TT*, with `data`, pattern matching and *linearity* annotations
  - An intermediate language *TTImp* with *implicit syntax*, *unification* and *auto implicit* arguments
  - A high level notation which desugars to *TTImp*
    - TODO: Interfaces, Records

Towards a new implementation: Blodwen

- `https://github.com/edwinb/Blodwen`
- Current status
  - Core language *TT*, with `data`, pattern matching and *linearity* annotations
  - An intermediate language *TTImp* with *implicit syntax*, *unification* and *auto implicit* arguments
  - A high level notation which desugars to *TTImp*
    - TODO: Interfaces, Records
- Implemented in Idris
  - How does Type Driven Development help?

Towards a new implementation: Blodwen

- `https://github.com/edwinb/Blodwen`
- Current status
  - Core language *TT*, with `data`, pattern matching and *linearity* annotations
  - An intermediate language *TTImp* with *implicit syntax*, *unification* and *auto implicit* arguments
  - A high level notation which desugars to *TTImp*
    - TODO: Interfaces, Records
- Implemented in Idris
  - How does Type Driven Development help?
- Plans
  - Focus early on *efficiency* and *inference*
  - Export *TTImp* as a library

Demonstration: Blodwen in “action”

Challenge: Higher Order Unification

Example

```
Op : (interpTy a -> interpTy b -> interpTy c) ->
     Lang gam a -> Lang gam b -> Lang gam c

Op plus (Var Stop) (Var (Pop Stop))
        : Lang [TyNat, TyNat] TyNat
```

Challenge: Higher Order Unification

Example

```
Op : {a : _} -> {b : _} -> {c : _} -> {gam : _} ->
     (interpTy a -> interpTy b -> interpTy c) ->
     Lang gam a -> Lang gam b -> Lang gam c
```

Interpreting types

```
interpTy : Ty -> Type;
interpTy TyNat = Nat;
interpTy (Arrow s t) = interpTy s -> interpTy t
```

Higher Order Unification Constraints

Example

```
Op plus (Var Stop) (Var (Pop Stop))
        : Lang [TyNat, TyNat] TyNat
```

Unification constraints

```
interpTy a =?= Nat -- from type of plus
interpTy b =?= Nat
interpTy c =?= Nat
```

sicsa*

Higher Order Unification Constraints

Example

```
Op plus (Var Stop) (Var (Pop Stop))
        : Lang [TyNat, TyNat] TyNat
```

Unification constraints

```
interpTy a =?= Nat -- from type of plus
interpTy b =?= Nat
interpTy c =?= Nat

a =?= TyNat  -- from type Lang [TyNat, TyNat] TyNat
b =?= TyNat
c =?= TyNat
```

Higher Order Unification Constraints

Example

```
Op plus (Var Stop) (Var (Pop Stop))
        : Lang [TyNat, TyNat] TyNat
```

Unification constraints

```
interpTy TyNat =?= Nat -- substituting a, b and c
interpTy TyNat =?= Nat
interpTy TyNat =?= Nat
```

Higher Order Unification Constraints

Unification constraints

```
Nat =?= Nat -- by normalising interpTy
Nat =?= Nat
Nat =?= Nat
```

Term Representation

- Several ways dependent types could help:
  - Index terms by *number of bound variables*
  - Index terms by their *names in scope*
  - Index terms by names in scope and the *types of those names*
  - Index terms by their *type*

Term Representation

- Several ways dependent types could help:
  - Index terms by *number of bound variables*
  - Index terms by their *names in scope*
  - Index terms by names in scope and the *types of those names*
  - Index terms by their *type*
- Totality requirements (self-imposed):
  - Everything `total`, where possible
  - Everything `covering`, without exception
  - Helps decide level of *precision* in types

Term Representation

- Several ways dependent types could help:
  - Index terms by *number of bound variables*
  - Index terms by their *names in scope*
  - Index terms by names in scope and the *types of those names*
  - Index terms by their *type*
- Totality requirements (self-imposed):
  - Everything `total`, where possible
  - Everything `covering`, without exception
  - Helps decide level of *precision* in types
- Biggest challenge, from experience: *variable names*

Term Representation

We choose to index terms by their *names in scope*

Core TT Terms

```
data Term : List Name -> Type where
     Local : Elem x vars -> Term vars
     Ref   : NameType -> (fn : Name) -> Term vars
     Bind  : (x : Name) -> Binder (Term vars) ->
             Term (x :: vars) -> Term vars
     App   : Term vars -> Term vars -> Term vars
     TType : Term vars
```

Term Representation

Core TT Definitions

```
record GlobalDef where
   type : Term []
   visibility : Visibility
   definition : Def

data Def : Type where
   PMDef : (args : List Name) -> CaseTree args -> Def
   Hole : Def
   Guess : (guess : Term []) ->
           (constraints : List Name) -> Def
```

Term Manipulation

Substitute a term for a name

```
subst : Term vars -> Term (x :: vars) -> Term vars
```

Term Manipulation

Substitute a term for a name

```
subst : Term vars -> Term (x :: vars) -> Term vars
```

Weaken the scope of a term

```
weaken : Term vars -> Term (x :: vars)
```

Term Manipulation

Substitute a term for a name

```
subst : Term vars -> Term (x :: vars) -> Term vars
```

Weaken the scope of a term

```
weaken : Term vars -> Term (x :: vars)
```

"Thin" the scope of a term

```
thin : (n : Name) -> Term (outer ++ inner) ->
       Term (outer ++ n :: inner)
```

Term Manipulation

"Thin" the scope of a term

```
insertElem : Elem x (outer ++ inner) ->
             Elem x (outer ++ n :: inner)

thin : (n : Name) -> Term (outer ++ inner) ->
       Term (outer ++ n :: inner)
thin n (Local prf) = Local (insertElem prf)
thin {outer} n (Bind x b sc)
   = let sc' = thin {outer = x :: outer} n sc in
         Bind x (map (thin n) b) sc'
thin n (App f a) = App (thin n f) (thin n a)
```

Term Manipulation

Type check a “raw” term in an environment

```
infer : Env vars -> Raw ->
        Maybe (Term vars, Term vars)
check : Env vars -> Raw -> Term vars ->
        Maybe (Term vars)
```

Normalise a term with free variables

```
nf : Env Term free -> Term free -> Term free
```

Term Manipulation

Evaluate a term with free variables

```
eval : Env Term free ->
       LocalEnv free vars -> Stack free ->
       Term (vars ++ free) -> NF free

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

data LocalEnv : List Name -> List Name -> Type where
     Nil  : LocalEnv free []
     (::) : Closure free -> LocalEnv free vars ->
          LocalEnv free (x :: vars)
```

Unification Problems

Representing Constraints

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

data Constraint : Type where
  MkConstraint : (env : Env Term vars) ->
     (x : Term vars) -> (y : Term vars) -> Constraint
```

Summary

- Indexing terms over names has (so far!) proved effective
  - More confidence in *correctness* of evaluation, unification, etc
  - *Helping* rather than *hindering* development. . .
  - . . . despite occasional small *proof obligations*

Summary

- Indexing terms over names has (so far!) proved effective
  - More confidence in *correctness* of evaluation, unification, etc
  - *Helping* rather than *hindering* development...
  - ...despite occasional small *proof obligations*
- Everything except evaluation and unification is `total`
  - Everything else `covering`...
  - ...so no crashes (run-time system permitting!)

Summary

- Indexing terms over names has (so far!) proved effective
  - More confidence in *correctness* of evaluation, unification, etc
  - *Helping* rather than *hindering* development...
  - ...despite occasional small *proof obligations*
- Everything except evaluation and unification is `total`
  - Everything else `covering`...
  - ...so no crashes (run-time system permitting!)
- Future work:
  - Immediate future: High level constructs (Interfaces, Records)
  - Compilation: How soon can it compile itself?
  - Coverage and totality checking
  - Longer term: export as a library? Other high level languages?

Further Reading

- Towards a practical programming language based on dependent type theory
  - *Ulf Norell*, 2007
- Type Inference, Haskell and Dependent Types
  - *Adam Gundry*, 2013
- Type-and-Scope Safe Programs and their Proofs
  - *Guillaume Allais et al*, CPP 2017
- Idris, a general-purpose dependently typed programming language: Design and implementation
  - *Edwin Brady*, JFP 23(5), 2013