

The Implementation of Idris 2

Part 2: Term Representation

Edwin Brady (ecb10@st-andrews.ac.uk)

University of St Andrews, Scotland

@edwinbrady

SPLV, 17th August 2020

- Some implementation details
 - The **Core** “monad”
- Core language, **TT**
 - Cut down **QTT** (no quantities or **let**)
 - Terms, definitions, case trees
 - Syntax only! *Typing rules* come tomorrow
- Term representation
 - Dealing with variable names
 - Term manipulation: weakening, contraction, substitution. . .

Two most important parts of the module hierarchy:

- **Core**: the core type theory (TT)
 - **Core.Core**: The “monad” carrying all the context
 - **Core.TT**: TT terms (more on this tomorrow)
 - **Core.CaseTree**: Compiled case trees, for evaluation
 - **Core.Context**: Storing definitions
 - **Core.Normalise**: Evaluation
 - **Core.Unify**: Unification
- **TTImp**: the surface language (TT + implicits)
 - **TTImp.Elab.Term**: Elaboration to TT
 - **TTImp.ProcessDecl**: Elaborating top level declarations

$t ::=$	x	(Variables)	$b ::=$	λ	(Lambda)
	$b\ x : t . t$	Binders		Π	(Function)
	$t_1\ t_2$	(Application)		pat	(Pattern variable)
	Type	(Type of types)		pty	(Pattern type)
	-	(Erased term)			

...and that's all!

$t ::= x$	(Variables)	$b ::= \lambda$	(Lambda)
$b\ x : t . t$	(Binders)	Π	(Function)
$t_1\ t_2$	(Application)	pat	(Pattern variable)
Type	(Type of types)	pty	(Pattern type)
$-$	(Erased term)		

...and that's all! Full *QTT* also has:

- Quantities on the binders
- **let** binding
- “As” patterns as terms
- Explicit **Force** and **Delay** for laziness

Function definitions consist of a *type declaration* and *pattern bindings*:

$$x : t$$
$$t_{lhs1} = t_{rhs1}$$
$$t_{lhs2} = t_{rhs2}$$
$$\dots$$
$$t_{lhsn} = t_{rhsn}$$

Function definitions consist of a *type declaration* and *pattern bindings*:

```
x : t
tlhs1 = trhs2
tlhs2 = trhs2
...
tlhsn = trhsn
```

In the clauses, variables are explicitly bound by **pat** binders, e.g.:

```
plus :  $\prod x : \text{Nat}. \prod y : \text{Nat}. \text{Nat}$ 
pat y : Nat . plus Z y = y
pat k : Nat . pat y : Nat . plus (S k) y = S (plus k y)
```

Data declarations consist of a *type constructor* and zero or more *data constructors*

data D : t where

C_1 : t_1

C_2 : t_2

...

C_n : t_n

For evaluation (and ease of compilation and coverage checking), pattern matching definitions compile to *case trees*:

c	$::=$	<code>case</code>	$x : t$	<code>of</code>	\vec{alt}	(Case split)
			t			(Expression)
		<code>missing</code>				(Missing case)
		<code>impossible</code>				(Unreachable case)
alt	$::=$	C	$\vec{x} \Rightarrow$	c		(Constructor application)
		$-$	\Rightarrow	c		(Match anything)

For evaluation (and ease of compilation and coverage checking), pattern matching definitions compile to *case trees*:

c	$::=$	<code>case</code>	$x : t$	<code>of</code>	\vec{alt}	(Case split)
			t			(Expression)
		<code>missing</code>				(Missing case)
		<code>impossible</code>				(Unreachable case)
alt	$::=$	C	$\vec{x} \Rightarrow$	c		(Constructor application)
		$-$	\Rightarrow	c		(Match anything)

(See: The Implementation of Functional Programming Languages, Simon Peyton Jones, Chapter 5 by Philip Wadler

<https://www.microsoft.com/en-us/research/publication/the-implementation-of-functional-programming-languages/>)

During elaboration, we often need to manipulate and inspect syntax, e.g.:

- Rename variables (α conversion)

During elaboration, we often need to manipulate and inspect syntax, e.g.:

- Rename variables (α conversion)
- Compare two terms for equality
 - Are they in the same *scope*

During elaboration, we often need to manipulate and inspect syntax, e.g.:

- Rename variables (α conversion)
- Compare two terms for equality
 - Are they in the same *scope*
- Substitute a term into the scope of a binder
 - Will the result be well scoped?

During elaboration, we often need to manipulate and inspect syntax, e.g.:

- Rename variables (α conversion)
- Compare two terms for equality
 - Are they in the same *scope*
- Substitute a term into the scope of a binder
 - Will the result be well scoped?
- “Weaken” a term: that is, add a new variable
 - e.g. when elaborating under a binder, to refer to types in the outer scope

During elaboration, we often need to manipulate and inspect syntax, e.g.:

- Rename variables (α conversion)
- Compare two terms for equality
 - Are they in the same *scope*
- Substitute a term into the scope of a binder
 - Will the result be well scoped?
- “Weaken” a term: that is, add a new variable
 - e.g. when elaborating under a binder, to refer to types in the outer scope
- “Contraction”: drop an unused variable

During elaboration, we often need to manipulate and inspect syntax, e.g.:

- Rename variables (α conversion)
- Compare two terms for equality
 - Are they in the same *scope*
- Substitute a term into the scope of a binder
 - Will the result be well scoped?
- “Weaken” a term: that is, add a new variable
 - e.g. when elaborating under a binder, to refer to types in the outer scope
- “Contraction”: drop an unused variable

Lesson from Idris 1 (and every other language implementation...):
Naming is hard!

A *binder* is either λ , Π , or a pattern binding. It's convenient to be generic in term representation:

```
data Binder : Type -> Type where
  Lam  : PiInfo -> ty -> Binder ty
  Pi   : PiInfo -> ty -> Binder ty
  PVar : ty -> Binder ty
  PVTy : ty -> Binder ty
```

PiInfo is either `Implicit` or `Explicit` (this is useful during elaboration)

Terms with explicit names

```
data Term : Type where
  Var : Name -> Term
  Bind : Name -> Binder Term -> Term -> Term
  App : Term -> Term -> Term
  TType : Term
```

Terms with explicit names

```
data Term : Type where
  Var : Name -> Term
  Bind : Name -> Binder Term -> Term -> Term
  App : Term -> Term -> Term
  TType : Term
```

Problems:

- Name clashes, α -conversion, distinction between *local* and *global* names
- No help from the type system

Terms with de Bruijn indexed locals

```
data Term : Type where
  Local  : Int -> Term
  Ref    : Name -> Term
  Bind   : Name -> Binder Term -> Term -> Term
  App    : Term -> Term -> Term
  TType  : Term
```

Terms with de Bruijn indexed locals

```
data Term : Type where
  Local  : Int -> Term
  Ref    : Name -> Term
  Bind   : Name -> Binder Term -> Term -> Term
  App    : Term -> Term -> Term
  TType  : Term
```

Problems:

- Manipulating de Bruijn indices is *hard*
 - Idris 1 does this, and got it wrong *a lot*
- Still no help from the type system

Well-scoped terms with de Bruijn indexed locals

```
data Term : Nat -> Type where
  Local  : Fin n -> Term n
  Ref    : Name -> Term n
  Bind   : Name -> Binder (Term n) -> Term (S n) ->
           Term n
  App    : Term n -> Term n -> Term n
  TType  : Term n
```

Well-scoped terms with de Bruijn indexed locals

```
data Term : Nat -> Type where
  Local  : Fin n -> Term n
  Ref    : Name -> Term n
  Bind   : Name -> Binder (Term n) -> Term (S n) ->
           Term n
  App    : Term n -> Term n -> Term n
  TType  : Term n
```

- Some help from the type system
 - e.g. *Weakening* has a more helpful type
weaken : Term n -> Term (S n)

Aside: The Well-Typed Interpreter

Types

```
data Ty = TyNat | TyFun Ty Ty
```

Well-typed terms

```
data Term : Vect k Ty -> Ty -> Type where
  Var : HasType i t gam -> Term gam t
  Val : (x : interpTy a) -> Term gam a
  Lam : Term (s :: gam) t ->
        Term gam (TyFun s t)
  App : Term gam (TyFun s t) ->
        Term gam s -> Term gam t
```


Aside: The Well-Typed Interpreter

Types

```
data Ty = TyNat | TyFun Ty Ty
```

Well-typed terms

```
data Term : Vect k Ty -> Ty -> Type where
  Var : HasType i t gam -> Term gam t
  Val : (x : interpTy a) -> Term gam a
  Lam : Term (s :: gam) t ->
        Term gam (TyFun s t)
  App : Term gam (TyFun s t) ->
        Term gam s -> Term gam t
```

Can we do this for *TT*?

- We index terms by the *names in scope*
- Use de Bruijn indices, with a proof that they refer to a name in scope

- We index terms by the *names in scope*
- Use de Bruijn indices, with a proof that they refer to a name in scope

Mapping de Bruijn indices to a name in scope

```
data IsVar : Name -> Nat -> List Name -> Type where
  First  : IsVar n Z (n :: ns)
  Later  : IsVar n i ns -> IsVar n (S i) (m :: ns)
```

Well-scoped terms with explicit names in the type

```
data Term : List Name -> Type where
  Local : (idx : Nat) ->
    (0 p : IsVar name idx vars) ->
      Term vars
  Ref : NameType -> Name -> Term vars
  Meta : Name -> List (Term vars) -> Term vars
  Bind : (x : Name) ->
    Binder (Term vars) ->
      Term (x :: vars) ->
        Term vars
  App : Term vars -> Term vars -> Term vars
  TType : Term vars
  Erased : Term vars
```

Well-scoped case trees

```
data CaseTree : List Name -> Type where
  Case : {name, vars : _} ->
    (idx : Nat) ->
    (0 p : IsVar name idx vars) ->
    (scTy : Term vars) ->
    List (CaseAlt vars) ->
    CaseTree vars
  STerm : Term vars -> CaseTree vars
  Unmatched : (msg : String) -> CaseTree vars
  Impossible : CaseTree vars

data CaseAlt : List Name -> Type where
  ConCase : Name -> (tag : Int) -> (args : List Name) ->
    CaseTree (args ++ vars) -> CaseAlt vars
  DefaultCase : CaseTree vars -> CaseAlt vars
```

Some operations on Terms

```
weaken    : Term vars -> Term (x :: vars)
contract  : Term (x :: vars) -> Maybe (Term vars)
embed     : Term vars -> Term (vars ++ ns)
subst     : Term vars -> Term (x :: vars) -> Term vars
rename    : CompatibleVars xs ys -> Term xs -> Term ys
```



Demonstration: Term manipulation