The Implementation of Idris 2

Part 3: Type Checking and Elaboration

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

SPLV, 19th August 2020

Today's Lecture

- Typing rules for *TT*
- *TTImp* syntax and representation
- Representing *values* (head normal forms)
- Elaboration
  - Processing *TTImp* declarations
  - Translating *TTImp* terms to a `Term`

Reminder: TT term syntax

$$
\begin{array}{lcll}
t & ::= & x & \text{(Variables)} \\
 & \mid & b\,x\,:\,t\,.\,t & \text{Binders} \\
 & \mid & t_1\ t_2 & \text{(Application)} \\
 & \mid & \texttt{Type} & \text{(Type of types)} \\
 & \mid & \_ & \text{(Erased term)}
\end{array}
\qquad
\begin{array}{lcll}
b & ::= & \lambda & \text{(Lambda)} \\
 & \mid & \Pi & \text{(Function)} \\
 & \mid & \texttt{pat} & \text{(Pattern variable)} \\
 & \mid & \texttt{pty} & \text{(Pattern type)}
\end{array}
$$

TT typing rules (environments)

Typechecking a term is relative to an *environment*, which consists of a list of binders:

$$\frac{}{\vdash \underline{\mathrm{valid}}} \qquad \frac{\Gamma \vdash S \ : \ \mathtt{Type}}{\Gamma; \lambda x \ : \ S \vdash \underline{\mathrm{valid}}} \qquad \frac{\Gamma \vdash S \ : \ \mathtt{Type}}{\Gamma; \Pi x : S \vdash \underline{\mathrm{valid}}}$$

TT typing rules (environments)

Typechecking a term is relative to an *environment*, which consists of a list of binders:

$$\frac{}{\vdash \underline{\mathrm{valid}}} \qquad \frac{\Gamma \vdash S \;:\; \mathtt{Type}}{\Gamma; \lambda x \;:\; S \vdash \underline{\mathrm{valid}}} \qquad \frac{\Gamma \vdash S \;:\; \mathtt{Type}}{\Gamma; \Pi x : S \vdash \underline{\mathrm{valid}}}$$

In code (`Core.Env`)

```
data Env : (tm : List Name -> Type) ->
           List Name -> Type where
     Nil : Env tm []
     (::) : Binder (tm vars) ->
            Env tm vars -> Env tm (x :: vars)
```

TT typing rules (conversion)

$$\frac{\Gamma \vdash x \;:\; S \qquad \Gamma \vdash T \;:\; \texttt{Type} \qquad \Gamma \vdash S \simeq T}{\Gamma \vdash x \;:\; T}$$

TT typing rules (conversion)

$$\frac{\Gamma \vdash x \ : \ S \qquad \Gamma \vdash T \ : \ \mathtt{Type} \qquad \Gamma \vdash S \simeq T}{\Gamma \vdash x \ : \ T}$$

That is, if . . .

- x has type S
- S and T are *convertible*
  - informally: have the same normal form

. . . then x also has type T

TT typing rules (variables and application)

$$\frac{(\lambda x\ :\ S) \in \Gamma}{\Gamma \vdash x\ :\ S} \qquad \frac{(\Pi x : S) \in \Gamma}{\Gamma \vdash x\ :\ S}$$

TT typing rules (variables and application)

$$\frac{(\lambda x \;:\; S) \in \Gamma}{\Gamma \vdash x \;:\; S} \qquad \frac{(\Pi x\!:\!S) \in \Gamma}{\Gamma \vdash x \;:\; S}$$

$$\frac{\Gamma \vdash f \;:\; \Pi x\!:\!S \to T \qquad \Gamma \vdash s \;:\; S}{\Gamma \vdash f\ s \;:\; T[s/x]}$$

TT typing rules (variables and application)

$$\frac{(\lambda x \;:\; S) \in \Gamma}{\Gamma \vdash x \;:\; S} \qquad \frac{(\Pi x : S) \in \Gamma}{\Gamma \vdash x \;:\; S}$$

$$\frac{\Gamma \vdash f \;:\; \Pi x : S \rightarrow T \qquad \Gamma \vdash s \;:\; S}{\Gamma \vdash f\ s \;:\; T[s/x]}$$

That is, if . . .

- f has a function type, $\Pi x : S \rightarrow T$
- s has the argument type S

. . . then f s has the result type T, with s substituted for x

TT typing rules (function types)

$$\frac{\Gamma; \Pi x : S \vdash T \ : \ \texttt{Type} \qquad \Gamma \vdash S \ : \ \texttt{Type}}{\Gamma \vdash \Pi x : S \rightarrow T \ : \ \texttt{Type}}$$

TT typing rules (function types)

$$\frac{\Gamma; \Pi x : S \vdash T \ : \ \texttt{Type} \qquad \Gamma \vdash S \ : \ \texttt{Type}}{\Gamma \vdash \Pi x : S \rightarrow T \ : \ \texttt{Type}}$$

That is, if ...

- The type of T, in an environment extended with x:S, is `Type`
- S also has type `Type`

...then $\Pi x : S \rightarrow T$ also has type `Type`

TT typing rules (lambda)

$$\frac{\Gamma; \lambda x : S \vdash e : T \qquad \Gamma \vdash \Pi x{:}S \rightarrow T : \texttt{Type}}{\Gamma \vdash \lambda x : S.e : \Pi x{:}S \rightarrow T}$$

TT typing rules (lambda)

$$\frac{\Gamma; \lambda x \ : \ S \vdash e \ : \ T \qquad \Gamma \vdash \Pi x : S \rightarrow \ T \ : \ \mathtt{Type}}{\Gamma \vdash \lambda x \ : \ S.e \ : \ \Pi x : S \rightarrow \ T}$$

That is, if ...

- The type of e, in an environment extended with x:S, is T
- The function type is indeed a `Type`

...then the lambda binding has type $\Pi x : S \rightarrow \ T$

TTImp

- The input language *TTImp* is a higher level syntax which *elaborates* to *TT*
- Essentially a *desugared* Idris
  - Desugaring (in `Idris.Desugar` in full Idris 2) is a purely syntactic transformation of the input program
  - Other front ends are possible!

TTImp

- The input language *TTImp* is a higher level syntax which *elaborates* to *TT*
- Essentially a *desugared* Idris
  - Desugaring (in `Idris.Desugar` in full Idris 2) is a purely syntactic transformation of the input program
  - Other front ends are possible!
  - Aside: I want to try:
    - an imperative(ish) front end
    - high level syntax for declaring effects
    - better syntactic support for linearity
    - ... but probably not in Idris 2 itself

Environments and Contexts

Checking *TTImp* happens relative to:

- an *environment* of local variables, `Env`
    - We've already seen this

Environments and Contexts

Checking *TTImp* happens relative to:

- an *environment* of local variables, `Env`
    - We've already seen this
- a global *context*, `Defs`, containing:
    - Type constructors (e.g. `Nat`)
    - Data constructors (e.g. `Z`, `S`)
    - Function declarations and definitions (e.g. `plus`)

Contexts

Definitions (Core.Context)

```
data Def : Type where
     None : Def
     PMDef : (args : List Name) ->
             (treeCT : CaseTree args) ->
             Def
     DCon : (tag : Int) -> (arity : Nat) -> Def
     TCon : (tag : Int) -> (arity : Nat) -> Def
     Hole : Def
     Guess : (guess : Term []) ->
             (constraints : List Int) -> Def
```

Contexts

Definitions (Core.Context)

```
record GlobalDef where
   constructor MkGlobalDef
   type : Term []
   definition : Def
```

Contexts (Core.Context)

```
Defs : Type
Defs = SortedMap Name GlobalDef
```

Note: Full Idris 2 carries a lot more information about definitions, e.g. call graph, size changes, search hints. . .

TTImp

TTImp, in code (`TTImp.TTImp`)

```
data RawImp : Type where
     IVar : Name -> RawImp
     IPi : PiInfo -> Maybe Name ->
           (argTy : RawImp) -> (retTy : RawImp) ->
           RawImp
     ILam : PiInfo -> Maybe Name ->
            (argTy : RawImp) -> (scope : RawImp) ->
            RawImp
     IPatvar : Name -> (ty : RawImp) -> (scope : RawImp) ->
               RawImp
     IApp : RawImp -> RawImp -> RawImp
     Implicit : RawImp
     IType : RawImp
```

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)
```

- We will implement this by following the *TT* typing rules
- But:
  - What about the *conversion* rule?
  - And what is `Glued` anyway?
- Each of these concern *Values*: normal forms

Values

- Elaborating `TTImp` regularly involves looking at how terms *reduce*, e.g.
  - are `Vect (2 + n) Int` and `Vect (S (S n)) Int` convertible?
  - is `NatOp` a function type, if `NatOp = Nat -> Nat`?
- We check this by comparing *values*
  - Separate representation
  - Guaranteed to be a head normal form *by construction*

What is a Value?

Informally:

- A *constructor* applied to some arguments
  - Arguments not yet reduced
  - "Cons should not evaluate its arguments"
- A *binder*
  - Scope not yet reduced
- A "stuck" *application*
  - e.g. `plus x (S (S Z))`
- A primitive
  - In TinyIdris, just `Type` or `_`

We only need to look at the *head*

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
```

Evaluation

Evaluating and quoting (`Core.Normalise`)

```
nf    : Defs -> Env Term vars -> Term vars ->
        Core (NF vars)
quote : Defs -> Env Term vars -> tm vars ->
        Core (Term vars)
```

- for details look up *normalisation by evaluation*
- Aside: why in `Core`?

Conversion

Checking conversion (Core.Normalise)

```
interface Convert (tm : List Name -> Type) where
  convert : {vars : _} ->
            Defs -> Env Term vars ->
            tm vars -> tm vars -> Core Bool

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

Details (with the related `unify`) tomorrow

Glued terms

- Elaboration, in an environment `Env Term vars`, turns a `RawImp` into:
  - a well-typed term, `Term vars`,
  - ...with its type
- How should we represent the type?
  - As a `Term vars`?
    - e.g. if we've looked up a name, stored as a `Term` in the environment or global context
    - Might never need to reduce
  - As a `NF vars`?
    - e.g. if we've had to normalise anyway for the conversion check?
    - save converting back and forth between `Term` and `NF`
  - Answer: why not *both*?

Glued terms

- A *Glued* term is constructed from either `Term` or `NF`, whichever we have available
- Converted to `NF` or `Term` lazily, if necessary
- Advantages:
    - Build what is convenient
    - Don't evaluate to `NF` unless we really have to

Glued terms

Glued terms (Core.Normalise)

```
data Glued : List Name -> Type where
     MkGlue : Core (Term vars) ->
              (Ref Ctxt Defs -> Core (NF vars)) ->
              Glued vars

getTerm : Glued vars -> Core (Term vars)
getNF   : {auto c : Ref Ctxt Defs} ->
        Glued vars -> Core (NF vars)

gnf   : Env Term vars -> Term vars -> Glued vars
glueBack  : Defs -> Env Term vars ->
            NF vars -> Glued vars
```

Elaboration

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)
```

- Implemented by following the typing rules for each of `RawImp`'s constructors

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

An IVar in RawImp could be either a local or a global (if valid)

Elaborating local variables (TTImp.Elab.Term)

```
checkTerm env (IVar n) exp
   = case defined n env of
         Just (MkIsDefined p) =>
           let binder = getBinder p env in
               checkExp env (Local _ p)
                        (gnf env (binderType binder))
                        exp
         ...
```

Elaboration

Elaborating global variables (`TTImp.Elab.Term`)

```
checkTerm env (IVar n) exp
   = case defined n env of
          ...
          Nothing =>
            do defs <- get Ctxt
               Just gdef <- lookupDef n defs
                    | Nothing => throw (UndefinedName n)
               let nt = case definition gdef of
                             DCon t a => DataCon t a
                             TCon t a => TyCon t a
                             _ => Func
               checkExp env (Ref nt n)
                        (gnf env (embed (type gdef)))
                        exp
```

Elaboration

Elaborating applications (TTImp.Elab.Term)

```
checkTerm env (IApp f a) exp
  = do (ftm, gfty) <- checkTerm env f Nothing
       fty <- getNF gfty
       case fty of
         NBind x (Pi _ ty) sc =>
           do defs <- get Ctxt
              (atm, gaty) <-
                    checkTerm env a
                              (Just (glueBack defs env ty))
              sc' <- sc defs (toClosure env atm)
              checkExp env (App ftm atm)
                           (glueBack defs env sc')
                           exp
         _ => throw (GenericMsg "Not a function type")
```