Towards Palatable Functional Programming with Dependent Type Theories

Donovan Crichton

August 2023

Preliminaries

- Slides and Examples available at:
  https://github.com/donovancrichton/Talks
- This talk: BFPG/PalatableProgrammingWithDTT

About me

Australian National University

- PhD Candidate
- Computing Foundations
- School of Computing

Griffith UNIVERSITY

- Visiting Scholar
- Trusted Systems Lab
- IIIS

ASD AUSTRALIAN SIGNALS DIRECTORATE

- ASD Co-Lab Scholar

Recalling some definitions to get us started

Rose Tree Maps: Haskell vs Idris

The Idris code will not pass the totality checker.

```
module HaskellRose where
{-# LANGUAGE GADTs #-}

data Rose :: * -> *
  where
  MkRose :: a -> [Rose a]
    -> Rose a

instance Functor Rose
  where
  fmap f
   (MkRose node children)
      = MkRose (f node)
        (fmap (fmap f)
          children)
```

```
module IdrisRoseBad
%default total -- WHY!??

data Rose : Type -> Type
  where
  MkRose : a -> List (Rose a)
    -> Rose a

implementation Functor Rose
  where
  map f
   (MkRose node children)
      = MkRose (f node)
        (map (map f)
          children)
```

What is a dependent type?

Dependent Type(s): $\Pi_{x \in A}.B(x)$ instead of $\lambda_{x \in A}.B$

A dependent type is a type that *depends* on a specific value of its input.

$\lambda_{x \in A}.B$ in Idris code, in general.

```
f : a -> b
f x = ?somedefinition
```

$\Pi_{x \in A}.B(x)$ in Idris code, in general.

```
module PiGeneralExample
P : a -> Type
P = ?sometypedefinition

f : (x : a) -> P x
f x = ?somevaluedefinition
```

What is a dependent type? (concrete examples)

$\lambda_{x \in A}.B$ in Idris code, concretely.

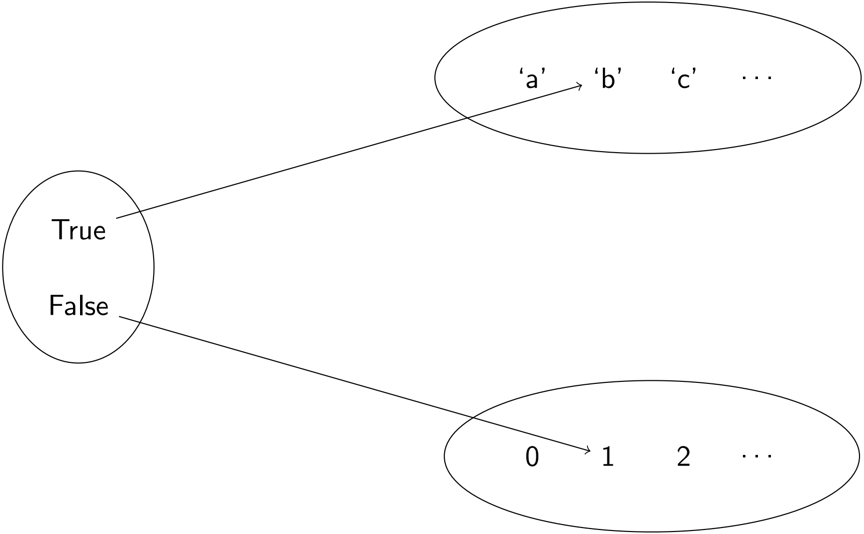
```
f : a -> List
f x = [x]
```

$\Pi_{x \in A}.B(x)$ in Idris code, concretely.

```
module PiSpecificExample
data Vector : Nat -> Type -> Type where
  Nil : Vector 0 a
  (::) : a -> Vector k a -> Vector (S k) a

trues : (k : Nat) -> Vector k Bool
trues Z = []
trues (S k) = True :: trues k
```

What is a dependent type? (diagrammatically)



What is a Dependent Type? (diagrammatic example)

Did you guess the right Idris type?

```
module PiDiagramExample
CharOrNat : Bool -> Type
CharOrNat True = Char
CharOrNat False = Nat

bOrOne : (b : Bool) -> CharOrNat b
bOrOne True = 'b'
bOrOne False = 1
```

What about $\Sigma_{x \in A}.B(x)$?

Dependent pairs can be constructed from Π types if your language supports recursive algebraic data type definitions.

Why do we mean by totality? (good)

Total Functions

A functions is *total* if it terminates and is defined for all possible values for its domain.

Functions that are total.

```
module TotalExample
contradiction1 : Bool -> Bool
contradiction1 x = False

contradiction2 : Bool -> Bool
contradiction2 True = False
contradiction2 False = False
```

Why do we mean by totality? (bad)

Total Functions

A functions is *total* if it terminates and is defined for all possible values for its domain.

Functions that are *not* total.

```
module TotalBadExample
contradiction1 : Bool -> Bool
contradiction1 True = False

contradiction2 : Bool -> Bool
contradiction2 x = not $ contradiction x
```

Why do we care about totality?

Can we type check this?

$f : \mathbb{N} \to \mathbb{N}$
$f = \lambda(x \in \mathbb{N}).f\ x$

$v : Vect\ k\ \alpha - > Vect\ (f\ k)\ \alpha$
$v\ xs = xs$

Infinite recursion in types in Idris

```
module ForeverExample
import Data.Vect

f : Nat -> Nat
f x = f x

v : Vect k a -> Vect (f k) a
v xs = xs
```

Using the totality pragma.

How Idris reduces Π application in types.

Idris will *only* reduce function application in types if the function has passed the totality checker.

The totality checker blocks evaluation:

```
module ForeverExampleFail
import Data.Vect
%default total
f : Nat -> Nat
f x = f x

v : Vect k a -> Vect (f k) a
v xs = xs
```

Back to our rose tree example.

Why does our rose tree Functor instance fail?

If map on lists is total, and our RoseTree is total. Why do we get errors when writing the functor instance?

Map on lists is total:

```
module TotalMapList
%hide Prelude.Functor
%hide Prelude.map
%default total
interface Functor f where
  map : (a -> b) -> f a -> f b
implementation Functor List where
  map f [] = []
  map f (x :: xs) = f x :: map f xs
```

Back to our rose tree example (2).

The definition of rose tree is total

```
module TotalMapList
%hide Prelude.Functor
%hide Prelude.map
%default total
interface Functor f where
  map : (a -> b) -> f a -> f b
implementation Functor List where
  map f [] = []
  map f (x :: xs) = f x :: map f xs
```

Is there a problem with our map definition?

```
map f (MkRose x xs) = MkRose (f x) (map (map f) xs)
```

How totality checking actually works

Syntactic Guardedness

Idris (and all other dependently typed proof assistants) check termination by means of a *syntactic guard* condition.

Recursive calls must be *guarded* by a destructor.

```
module GuardedRecursiveCall
%default total
len : List a -> Nat
len (x :: xs) = 1 + length xs
len [] = 0
```

How totality checking actually works (2)

Corecursive calls must be *guarded* by a constructor, and correctly *annotated* as Inf.

```
module GuardedCorecursiveCall
%hide Prelude.Stream.Stream
%default total
data Stream : Type -> Type where
  (::) : a -> Inf (Stream a) -> Stream a
implementation Functor Stream where
  map f (x :: xs) = f x :: map f xs

ones : Stream Nat
ones = 1 :: ones

twos : Stream Nat
twos = map (+1) ones
```

Summary of totality via syntactic guards

The (map f) call on our RHS does not meet the syntactic guard restriction.

```
map f (MkRose x xs) = MkRose (f x) (map (map f) xs)
```

- Syntactic guards do not work well with HOF.
- Inf for corecursion is broken.
- Most corecursive approaches are broken (sized types, delay/force, infinity) what about co-patterns?
- Manual proofs of well-foundedness can be onerous.

Introducing Type Theory

Components of a Type Theory

A Type Theory consists of a Grammar, a set of judgement rules, a set of desirable properties that are proven to hold, and a given semantics.

Why do we need all this?

In a dependently typed setting, the PAT (Propositions-As-Types) interpretation is assumed to hold. Alterations to any of these features (i.e extra syntactic annotations, different type checking algorithms, etc) can break the desirable properties and in the worst case, allow you to prove things that are not true.

Grammar (Parsed Lexemes)

What is a grammar?

A formal set of syntax used in the definition of your type theory, usually given in BNF.

An example grammar for a dependent type theory

$$\begin{aligned}
\Gamma &::= \diamond \mid \Gamma, x : S \\
S, T, U &::= (x : S) \to T \mid (x : S) \otimes T \mid El(M) \\
&\quad \mid Bool \mid Unit \mid \mathcal{U} \\
P, M, N &::= x \mid \Pi(x : S).M \mid M\ N \mid True \mid False \\
&\quad \mid \star \mid fst\ M \mid snd\ M \mid If\ P_{Bool}\ M\ N
\end{aligned}$$

Rules - Type Formers (Type Constructors)

Type Formers

A type formation rule is analogous to a type constructor.

List Type Constructor in Idris

```
data List : (s : Type) -> Type where
```

$$\frac{\Gamma \vdash S : \mathcal{U}}{\Gamma \vdash \mathrm{List}\ S : \mathcal{U}}(\mathit{ListTy})$$

Rules - Term Formers (Data Constructors)

Term Formers

A term formation rule is analogous to a data constructor.

List Data Constructors in Idris

```
data List : (s : Type) -> Type where
  Nil : List a
  (::) : a -> List a -> List a
```

$$\frac{\Gamma \vdash S : \mathcal{U} \qquad \Gamma \vdash \mathsf{List}\ S : \mathcal{U}}{\Gamma \vdash [] : \mathsf{List}\ S}(\mathit{Nil})$$

$$\frac{\Gamma \vdash S : \mathcal{U} \qquad \Gamma \vdash \mathsf{List}\ S : \mathcal{U} \qquad \Gamma \vdash x : S \qquad \Gamma \vdash xs : \mathsf{List}\ S}{\Gamma \vdash x :: xs : \mathsf{List}\ S}(\mathit{Cons})$$

Properties

Normalisation

Prove that all terms reduce to an irreducible normal form. I.e show that evaluation of terms eventually stops.

Canonicity

Prove that every closed computation (evaluation in an empty context) of a type results in a singular canonical value. i.e for every term of $\diamond \vdash x : \mathbb{N}$ we get a natural number literal when we evaluate.

Consistency

Prove that we cannot derive $\bot$, which in dependent type theories is the type with no terms.

Decidability

Give a decision procedure for type checking, i.e an algorithm that for all $\Gamma \vdash x : A$ we can determine if $x \in A$ or not.

Semantics

If we're just manipulating symbols, how can we prove theorems in 'real' mathematics?

We need a *semantics* for our syntax. A mapping of some area of mathematics to our symbols and back. Common semantics are denotational, operational, and axiomatic. See e.g [Winskel, 1993]

Which one do we use?

One of many category theoretic semantics used for dependent types is categories with families...also used are categories with attributes, comprehension categories, display map categories, natural models, contextual categories.
Sadly beyond the scope of this talk (and me, for the moment!)

Guarded Dependent Type Theory

All this talk of type theory...what about our Totality problem?

A promising alternative approach may be a type-directed approach to guardedness, instead of a syntactic one.

Guarded Dependent Type Theory [Bizjak et al., 2016]

"We present guarded dependent type theory, gDTT, an extensional dependent type theory with a 'later' modality and clock quantifiers for programming and proving with guarded recursive and coinductive types. The later modality is used to ensure the productivity of recursive definitions in a modular, type based, way."

Extensionality and Intensionality

Extensionality

Two objects may be judged to be equal if they have the same external properties. Best seen in functional extensionality: $f = g \iff \forall x, f(x) = g(x)$

Intensionality

Two objects may be judged to be equal if they have the same internal properties. In practice, two expressions may be judged equal if they reduce to the same normal form.

Intensional Equality in Idris

```
module IntensionalEquality -- single = is prohibited.
data (==) : (a : Type) -> (b : Type) -> Type where
  Refl : a == a
```

Observational Type Theory

GDTT is extensional but Idris is intensional?

Fortunately, not all is lost. There have been recent advanced in other dependent type theories that promise *just* enough extensionality to get the job done.

Observational Type Theory [Altenkirch et al., 2007] [Pujet and Tabareau, 2022]

“Building on the recent extension of dependent type theory with a universe of definitionally proof-irrelevant types, we introduce TT_{obs}, a new type theory based on the setoidal interpretation of dependent type theory. TT_{obs} equips every type with an identity relation that satisfies function extensionality, propositional extensionality, and definitional uniqueness of identity proofs (UIP).”

Quantitative Type Theory

Idris has quantities?

Idris is (mostly) based on a linear dependent type theory with the hope that programmers can use this to formally prove properties about execution times and memory usage. This means we're still missing a step!

Quantitative Type Theory [Atkey, 2018]

"We present Quantitative Type Theory, a Type Theory that records usage information for each variable in a judgement, based on a previous system by McBride. The usage information is used to give a realizability semantics using a variant of Linear Combinatory Algebras, refining the usual realizability semantics of Type Theory by accurately tracking resource behaviour."

Towards $GOLD_{tt}$ - Construction via Formalisation

Finally! I can talk about my PhD!

Work towards a Guarded, Observational, Linear, Dependent, Type Theory ($GOLD_{tt}$) that allows us to combine features from all theory type theories.

Formalisation

To *Formalise* or *Mechanise* a type theory is to encode it in a proof assistant where machine checked proofs are given for as many of the components that make up a type theory as possible.

Methodology

Formalise each type theory in turn to get deep insights about the inner workings.
Then Formalise $GOLD_{tt}$.
Then Give category theoretic semantics.

Closing and ANU Plug

ANU is looking for PhD Students in Formal Methods!

The Computing Foundations cluster at the School of Computing, The Australian National University, have multiple PhD scholarships available in areas relevant to the types community, including logic, programming languages, systems, formal methods, theory and software engineering.

If you want to do cutting edge research in any of these fields, or a mix thereof, visit our page and get in touch with any of the staff listed.

Please also contact Michael Norrish (cluster lead) for further details.

Thank you!

References

T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, 2007.

R. Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 56–65, 2018.

A. Bizjak, H. B. Grathwohl, R. Clouston, R. E. Møgelberg, and L. Birkedal. Guarded dependent type theory with coinductive types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 20–35. Springer, 2016.

L. Pujet and N. Tabareau. Observational equality: now for good. *Proceedings of the ACM on Programming Languages*, 6(POPL): 1–27, 2022.

G. Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.