Cool ideas on PL design

A PLCT report of my trip to Pittsburgh

Tesla Zhang

May 29, 2023

The HoTT '23 conference took place in CMU!

I was there ↓:

https://hott.github.io/HoTT-2023//photo/

I met (in random order) András Kovács, Thierry Coquand, Anders Mörtberg, Frank Dai, Favonia, Astra Kolomatskaia, Andrew Swan, Reed Mullanix, David Berry, Joseph Hua, 王竹阳, Bob Harper, Carlo Angiuli, Daniel Gratzer, Harrison Grodin, David Jaz Myers, Egbert Rijke, Matteo Spadetto, Cipriano Cioffo, and many others!

The HoTT book first edition is released by Andrej Bauer!

Open problems in cubical type theory:

1. Regularity
2. How to discuss syntax problems with Jon Sterling
3. How to pronounce the last name of Reed

What did I learn?

- Synthetic algebraic geometry in HoTT
- Synthetic homotopy theory (very little)
- Synthetic differential geometry
  - Maybe just smooth infinitesimal analysis 🤣
- Some category theory
- Modalities in type theory
- Cubical sets and cubical type theory
- How to throw a disc
- Some cool PL ideas!

Cool PL ideas!

- Runtime code generation for dependent types
  - Credits to András Kovács
- 4 applications of extension types, with different types of cofibrations
  - Credits to Reed Mullanix
  - I messaged Ningning Xie 'bout this but she didn't reply 🥲
- These slides are reviewed by Trebor

Assumptions

- Basic impression on DT
- Runtime codegen part:
  - Compilers, interpreters, etc.
- Ext types part:
  - Basic impression on propositional logic
  - Typing rules

Staging

Staging

- “Multi-stage programming” by another name
- Mainly studied in the ML community
- Very good but never popular
- Some sort of partial evaluation
  - Mainly in the Lisp community

Staging

- Split programs into a sequence of stages
- Expand all code “until” a specified stage
- Simple example: meta level + object level

Staging

Unstaged vs. staged:

```
val x1 : Int = 114 + 514
val x2 : Code[Int] = Delay { 114 + 514 }

assert(x2.splice == x1) // 628
```

Staging

Staged programs are composeable:

```
val x3 = Delay { x2.splice + x2.splice }
```

How is it not lazy evaluation?

- Compiler can unfold the metalevel code completely without touching the object level code
- Metalevel code can be thought of as `constexpr` or `comptime`
- After unfold, all `Code[T]` become `T` and the rest is gone

You are already doing staging

- PLCT folks do compilers/interpreters
- These are staged programs (cf. 二村 projection)!
  - `Code[T]` is the generated code
  - `T` is the code in your compiler
- Interpreter: full evaluation (i.e. LLVM-based interpreter)

Why not staging now?

- It may seem easy to do right now, just inline some code
- Most compilers can do inline optimization

Why not staging now?

- We may see `Code[A → B]` with meta code in it:

```
def wind (s : S¹) : Int
| base => 0
| loop => ...

def main = Delay {
  λ (x : S¹). print(wind(x))
}
```

Difficulity

- It would be too much to ask for a language to inline code in closures
- This is known as *partial evaluation*

```
def fib (n : Int) = if (n < 2) 1 else fib(n - 1) + fib(n - 2)
def main {
  print(fib(114))
  print(fib(514))
}
```

- We don't usually *inline* recursive functions, but staging requires so

Difficulty

- What if meta level code crashes?
  - Stop the code generation – seems reasonable
- What if meta level code loops?

Dependent Types

- DT is (almost) Turing incomplete: we strictly forbid infinite loops
- DT has compile time code execution including recursion
  - Cf. my old talk on `printf`
  - Type checking DT requires running code
  - Totality guarantee is good for critical code
  - Same goal, same pain, but DT already solved it!
- DT + staging seems good, but…

Dependent Types

- Jason Hu: dependent type is **not** practical.
    - Doing IO outside a monad is essential for real world programming
    - Pure FP is not practical, while DT depends on it
    - A server, say, requires infinite loops

Type theory separation

- Credits to András Kovács, when we had a lunch together
  - Same day he lost his name card, later found by Frank Dai

Type theory separation

- Meta level: dependent types
- Require the object level code to be **simply typed** after unfolding the meta level
- Object level can loop
- Meta level is responsible for **generating** object level code

Relation to FFI

- Object level functions can be regarded as foreign functions
- Semantics of object level functions are inaccessable

Relation to Idris/Lean style DT

- The object level can be *anything*, e.g. Java-like OOP with subtyping polymorphism
- Unsafe features can be really unsafe in object level
- IMO this is better than having two separate semantics (runtime vs. compile time) for the same language
  - Lean4, Idris2, etc.

Industry adoption of DT

- A new two-stage language where the object level is a subset of an existing, well-established language
  - We perform an almost identical translation
- The meta level is an insanely safe DT language
  - For our experimentation 😈

Open questions

- Semantics? Maybe some weird modalities?

Extension types

Extension types

- Classical idea from higher type theories
  - Cf. Orton–Pitts, Riehl–Shulman
  - Cf. Angiuli–Cavallo–Favonia–Mullanix–Sterling
  - Cf. Aya development team

Syntax of extension types

- Suppose φ denotes a proposition
  - N.b.! Not in the sense of Curry–Howard
- Let's call it a *cofibration*

Syntax of extension types

- Type checking under cofibrations:

$$\Gamma, \varphi \vdash a : A$$

- It means "when φ is true, the judgment $a : A$ holds".

Syntax of extension types

- Truthness of φ:
  - Must be efficiently decidable
  - May affect the RHS of the judgment

Syntax of extension types

- Let $\varphi := (i = 1)$, so it holds only when $i \equiv 1$. Then:

$$\Gamma, \varphi \vdash (i + 2 \equiv 3) : A$$

Applications of extension types

1. Cubical type theory (beyond the scope of this presentation)
2. (Dependent) records
   - Metaprogramming, e.g. deriving
   - Definitional projections (cf. Aya classes)
3. Controlling unfolding in DT
   - Cf. Gratzer–Sterling–Angiuli–Coquand–Birkedal

Controling unfolding

- In DT, unfolding is arbitrary
- Leads to mysterious error messages and dependency problems

```
_ : intLoop (negsuc 1) ≡ refl
_ = refl
```

Controling unfolding

- Behold the error message:

```
hcomp (doubleComp-faces (λ _ → base) (λ i → loop (~ i)) i)
(intLoop (negsuc 0) i)
!= base of type S¹
```

Controling unfolding

- Expectation:

```
sym loop • sym loop != refl of type S¹
```

- But • is implemented in a complicated way (`hcomp` things)

Controling unfolding

- Implicit dependency on computation rules
- In industry languages, changing type signatures causes compatibility issues
  - But changing the implementation is fine
- In DT, changing the implementation may also cause compatibility issues

Scenario

- `f` is a function, where `f x = a + b` for some `a` and `b`
- Suppose `+` is propositionally commutative (but not definitionally)
- Now `g` proves `f x = a + ?` for some `?`
- Change `f x = b + a` seems legit, but `g` is now broken
- If we only do local type checking, you don't get notified for such bad changes
    - But keep running global type checking is expensive

Scenario

- This is not block chain, we expect to make breaking changes
  - Optimizations usually involve breaking changes

Controlling unfolding to the rescue

- Explicitly annotate which functions are we planning to unfold
- E.g. `g` is defined as `g : xx unfolds f`
- Definitions give rise to a graph of “dependencies”
- Infer type checking order from this graph

Implementation

- Let φ be a collection of names of definitions
- Calling a definition $f : A = a$ looks like $\mathrm{call}(f) : \{A \mid [f] \mapsto a\}$
- If $f \in \varphi$ then $[f]$ is considered true

$$\begin{aligned} \Gamma, f \vdash \mathrm{call}(f) \equiv a : A & \quad (\mathrm{unfold}) \\ \Gamma \vdash \mathrm{call}(f) : A & \qquad (\mathrm{do\ not\ unfold}) \end{aligned}$$

Records

Records

```
struct WrapInt(u32);
struct EncapInt(u32);
```

- Different in nominal typing but the same in structural typing
  - Compare types by their names
  - Compare types by comparing the underlying structure

Records

- Nominal typing bad for metaprogramming
  - Difficult to have a uniform interface for reflected AST
- Structural typing generates sophisticated error messages
  - Type mismatch: “long type expression” $\neq$ “long type expression”

Records

- Can we have both pros?
- Yes, with extension types!
  - Credits to Reed Mullanix

Records

- We translate records to tuples after type checking
- We use extension types to optionally do such translation in advance
  - Access some equations that only hold after translation
- Metaprogramming only operates on the tuple representation

Definitional projection

Definitional projection

```
class Precat
| Ob : Type
| Hom : Ob -> Ob -> Type
| Hom-set (A B : Ob) : isSet (Hom A B)
| id (A : Ob) : Hom A A
| ....
```

Definitional projection

- `Precat` is the type for all instances of the class `Precat`.

Definitional projection

- `Precat` is the type for all instances of the class `Precat`.
- `Precat { Ob := Group }` is the type for all instances of the class `Precat` whose `Ob` field is `Group`.

Definitional projection

- `Precat` is the type for all instances of the class `Precat`.
- `Precat { Ob := Group }` is the type for all instances of the class `Precat` whose `Ob` field is `Group`.
- `Precat { Ob := Group, Hom := GroupHom }` is the type for all instances of the class `Precat` whose:
  - `Ob` field is `Group`, and
  - `Hom` field is `GroupHom`.

Definitional projection

- This is called *anonymous class extension*
- Already available in Arend

Definitional projection

We write `Precat { Ob := Group }` as `Precat Group` for simplicity.

Definitional projection

We further want *definitional projection*:

- Suppose `A : Precat Group`, then `A.Ob` computes to `Group`.
- Suppose `A : Precat Group GroupHom`, then `A.Hom` computes to `GroupHom`.

Definitional projection

- We can do this with extension types!

$$A : \{\mathrm{Precat} \mid .\mathrm{Ob} \mapsto \mathrm{Group}\}$$

Definitional projection

$$\frac{\Gamma \vdash M : \text{Precat} \qquad \Gamma \vdash M.m \equiv N}{\Gamma \vdash M : \{\text{Precat} \mid .m \mapsto N\}}$$

Definitional projection

$$\frac{\Gamma \vdash M : \{\text{Precat} \mid .\, m \mapsto N\}}{\Gamma \vdash M.\, m \equiv N}$$

Q&A

Thanks for listening!