Scheme To Dependent Type theory In 100 Lines

Gershom Bazerman

Do Types Have the Lisp Nature?

Types Do Not Have Lisp Nature

- To Have Lisp Nature Everything Must Be Data
- To Have Lisp Nature Everything Must Be Code
- To Have Lisp Nature Your Program Must be One
- To Have Lisp Nature Compilation and Evaluation Must Be One

My Claim:
Dependent Types Have Lisp Nature

Dependent Types as a DSL for Logic

- ~~A Programming Language with Powerful Types~~
- ~~A system for proving mathematical statements~~
- A system for representing knowledge equipped with an algorithmic procedure for verification.

Dependent Types as a DSL for Logic

- Propositional Logic: A or B implies C
- First Order Logic: forall x. P(x) implies Q(x).
- Higher Order Logic: forall P. P(x) implies exists y. P(y)

Dependent Types as a DSL for Logic

- Logic with quantifiers includes symbols representing things not just drawn from logic, but drawn from the world.
- Lisp is our world, and we equip it with some logical symbols — And, Or, Implies.
- We name them as Product (*), Coproduct (+), and Function (→).

Dependent Types as a DSL for Logic

- Now we introduce quantifiers — forall, and exists.
- We write them as pi (Π) and sigma (Σ) — dependent product, and dependent sum.
- Alternate pronunciations: "If given a…" and "There is given a…" (or even "Give me a…" and "I have a…").

Syntax and Semantics

- forall (x : Nat). (x > 10) → (x > 5)
- forall (x : Nat). (x > 10) → (x < 5)
- forall (x : Cheese). not x → Crackers
- A syntax for logic is not yet a *language* for logic.

Syntax and Semantics

"Colorless green ideas sleep furiously"
— Noam Chomsky, 1955

The BHK Interpretation

Propositions as Problems, Proofs as Constructions
[Brouwer — 1908, 1924; Heyting —1934; Kolmogorov — 1932]

A proof of A $\wedge$ B is given by presenting two proofs -- one of A, one of B
A proof of A $\vee$ B is given by presenting either a proof of A or a proof of B
A proof of A $\rightarrow$ B is a construction to transform a proof of A into a proof of B
A proof of $\neg$ A is a proof of (A $\rightarrow$ False) where False has no proof
A proof of forall (a elem A). P(a), is a procedure that converts any element of A into a proof of P(a).

(This interpretation can hold with or without the excluded middle.)

Realizability

Formulae as Specifications, Proofs as Numerical Realizers [Kleene, 1945]

Constructions are numbers, read as partial functions. We encode functions with Gödel codes.

The Curry-Howard Correspondence

Propositions as Types, Proofs as Lambda Terms (encoded as Sets) [Howard, 1968-69]

Constructions are typed lambda terms (encoded set theoretically). In this construction, propositions directly correspond to types of these terms. Howard also introduced the world's first *dependent* type system, in which types may depend on terms. In this system, equality may be stated between terms.

DeBruijn, in constructing Automath, contemporaneously and independently, also introduced a dependent type system.

Intuitionistic Type Theory

Propositions as Types, Proofs as Lambda Terms (Directly) [Martin-Löf, 1970, 1972…]

Constructions are typed lambda terms. Types may contain terms ***and*** terms may operate on types. Equality may be stated between terms, and between types.

While Howard's system operates on Heyting arithmetic, Martin-Löf's was designed to operate on the whole of mathematics.

Computational Type Theory

Propositions as (Refinement) Types, Proofs as Untyped Lambda Terms
[Constable et al., 1985]

Constructions are *un*typed lambda terms. Types may contain terms *and* terms may operate on types.

Other Important Variations

The Calculus of Constructions [Coquand, Huet, 1984]

The Calculus of Inductive Constructions [Coquand, Paulin, 1990]

The LF Logical Framework [Harper, Honsell, Plotkin, 1987]

And others…

MESS

Martin-Löf Extensible Specification and Simulator
[Being Presented Now, 2015]

Scheme-as-a-Logical-Framework

A construction to transform a proof of A into a proof of B is a scheme function that transforms an object we judge to be a proof of A into an object we judge to be a proof of B.

What do we mean by Judgment?

Officially: “On the Meanings of the Logical Constants and the Justification of Logical Laws,” Martin-Löf, 1983

For our purposes: A judgment is a semi-decision procedure whose domain is one or more Scheme terms (and whose range is, naturally, #t, #f, or loop).

Some Ground Terms

```
; value formers
(struct lam-pi (var vt body))
(struct app (fun arg))
; primitives
(struct closure (typ body))

; type formers
(struct type-fun (dom codom))
; one basic type
(define type-unit 'type-unit)
; dependency
(struct type-pi (var dom codom))
(define type-type ‘type) ;inconsistent!
```

Judgement 1: "T is a type."

```
(define (type? cxt t)
  (match (red-eval cxt t)
    [(type-fun a b) (and (type? cxt a) (type? cxt b))]
    ['type-unit #t]
    [(? symbol? vname) #:when (eq? type-type (find-cxt vname cxt)) #t]
    [(type-pi var a b)
     (and (type? cxt a)
          (extend-cxt var a cxt (newvar newcxt) (type? newcxt (b newvar))))]
    ['type #t]
    [t (type?-additional cxt t)]))
```

Aside: Reduction

```
(define/match (reduce cxt body)
  [(_ (app (lam-pi var vt  b) arg))
   (if (hasType? cxt arg vt) (reduce cxt (b arg)) (error)]
  ; application of closures (primitives) is non-strict.
  [(_ (app (closure ty b) arg))
   (closure (app-type cxt (red-eval cxt ty) arg) (lambda (cxt) (app (b cxt) arg)))]
  [(_ (app fun arg)) (if (or (not fun) (symbol? fun)) (error)
                         (reduce cxt (app (reduce cxt fun) arg)))]
  [(_ _) body])

(define (red-eval cxt x)
  (match (reduce cxt x)
    [(closure typ b) (red-eval cxt (b cxt))]
    [v v]))
```

Aside: some syntactic sugar

```
(define apps
  (lambda (fun . args)
    (foldl (lambda (arg acc) (app acc arg)) fun args)))

(define-syntax-rule (lam  (x t) body) (lam-pi  (quote x) t (lambda (x) body)))

(define-syntax-rule (pi    (x t) body)  (lam-pi  (quote x) t (lambda (x) body)))

(define-syntax-rule (pi-ty (x t) body) (type-pi (quote x) t (lambda (x) body)))

(define-syntax-rule (close    t  body)  (closure t body))
```

Judgement 2: “X has type T.”

```
(define (hasType? cxt x1 t1)
  (match* ((reduce cxt x1) (red-eval cxt t1))
    [((closure typ b) t) (eqType? cxt typ t)]
    [((? symbol? x) t) #:when (eqType? cxt t (find-cxt x cxt)) #t]
    [((lam-pi vn vt body) (type-fun a b))
     (and (eqType? cxt vt a)
          (extend-cxt vn vt cxt (newvar newcxt) (hasType? newcxt (body newvar) b))))]
    [(x 'type-unit) (null? x)]
    [((lam-pi vn vt body) (type-pi _ a b))
     (and (eqType? cxt vt a)
          (extend-cxt vn vt cxt (newvar newcxt)
                      (hasType? newcxt (body newvar) (reduce newcxt (b newvar)))))]
    [(x 'type) (type? cxt x)]
    [(x t) (hasType?-additional cxt x t)]))
```

Zooming In: Typechecking Functions

Checking a function has a result type at all arguments of a type is to check that it has a type at a *generic* argument of that type.

```
[((lam-pi vn vt body) (type-fun a b))
    (and (eqType? cxt vt a)
        (extend-cxt vn vt cxt (newvar newcxt)
                (hasType? newcxt (body newvar) b)))]
```

Checking a dependent function means reducing as we check:

```
 [((lam-pi vn vt body) (type-pi _ a b))
    (and (eqType? cxt vt a)
        (extend-cxt vn vt cxt (newvar newcxt)
                (hasType? newcxt (body newvar)
                        (reduce newcxt (b newvar)))))]
```

Judgement 3: "T1 and T2 are equal as types."

```
(define (eqType? cxt t1 t2)
  (match* ((red-eval cxt t1) (red-eval cxt t2))
    [((type-fun a b) (type-fun a1 b1))
     (and (eqType? cxt a a1) (eqType? cxt b b1))]
    [((type-pi v a b) (type-pi v1 a1 b1))
     (and (eqType? cxt a a1)
          (extend-cxt v a cxt (newvar newcxt)
                      (eqType? newcxt (b newvar) (b1 newvar))))]
    [((? symbol? vname) (? symbol? vname1)) (eq? vname vname1)]
    [(a b) (and a b (or (eqType?-additional cxt a b) #f))]))
```

ITT is an Open System

We extend it with types by describing how to extend judgments over them, as well as how to **introduce** and **eliminate** them.

```
(define type-judgments '())
(define (type?-additional cxt t)
  (for/or ([p type-judgments]) (p cxt t)))

(define hasType-judgments '())
(define (hasType?-additional cxt x t)
  (for/or ([p hasType-judgments]) (p cxt x t)))

(define eqType-judgments '())
(define (eqType?-additional cxt t1 t2)
  (for/or ([p eqType-judgments]) (p cxt t1 t2)))
```

ITT is an Open System

We extend it with types by describing how to extend judgments over them, as well as how to **introduce** and **eliminate** them.

```
(define intro-true #t)
(define intro-false #f)

(define bool-induct
  (pi (p (type-fun type-bool type-type))
  (lam (x (app p #t))
  (lam (y (app p #f))
  (pi (bl type-bool)
  (close (app p bl) (lambda (cxt) (if (red-eval cxt bl) x y))))))))
```

ITT is an Open System

```
(define intro-pair (pi (a type-type) (pi (b type-type)
  (lam (x a)
  (lam (y b)
  (close (pair a b) (lambda (cxt) (cons a b))))))))

(define pair-induct (pi (a type-type) (pi (b type-type)
  (pi (p (type-fun (pair a b) type-type))
  (lam (f type-pi (x a) (type-pi (y b) (app p (pair x y))))
  (pi (z (pair a b)
  (close (app p z) (lambda (cxt) (let [(z-eval (red-eval cxt z))]
                                   (apps f (car z-eval) (cdr z-eval)))))))))))
```

Mathematical Propositions involve Equality

The identity type (initially derived from Howard):

for all types A, and B, A /\ B is a type.
for all types A, and B, A \/ B is a type.
for all types A, and B, A -> B is a type.
for all types A, and values x of type A, and functions p : A -> Type, pi_x:A P(x) is a type.
for all types A, and _values_ x, y of type A, x ==_A y is a type.

To check if x ==_A y is a type, we must now know that x and y are equal as *values* of type A.

Judgement 4: "X and Y are equal as values at type T."

```
(define (eqVal? cxt typ v1 v2)
  (match* ((red-eval cxt typ) (red-eval cxt v1) (red-eval cxt v2))
    [((type-fun a b) (lam-pi x xt body) (lam-pi y yt body2))
     (and (eqType? cxt a xt) (eqType? cxt a yt)
          (extend-cxt x xt cxt (newv newcxt)
                      (eqVal? newcxt b (body newv) (body2 newv)))))]
    [((type-pi v a b) (lam-pi x xt body) (lam-pi y yt body2))
     (and (eqType? cxt a xt) (eqType? cxt a yt)
          (extend-cxt x xt cxt (newv newcxt)
                      (eqVal? newcxt (b newv) (body newv) (body2 newv)))))]
    [('type-unit _ _) #t]
    [('type a b) (eqType? cxt a b)]
    [(_ (? symbol? x) (? symbol? y)) #:when (eq? x y) #t]
    [(rtyp x y) (eqVal?-additional cxt rtyp x y)]))
```

ITT is an Open System

```
(define (new-form type-judgment hasType-judgment eqType-judgment
eqVal-judgment)
  (cond [type-judgment
   (set! type-judgments      (cons type-judgment    type-judgments))])
  (cond [hasType-judgment
   (set! hasType-judgments (cons hasType-judgment hasType-judgments))])
  (cond [eqType-judgment
   (set! eqType-judgments  (cons eqType-judgment  eqType-judgments))])
  (cond [eqVal-judgment
    (set! eqVal-judgments    (cons eqVal-judgment   eqVal-judgments))]) )
```

(struct type-eq (type v1 v2))

```
(new-form
(match-lambda** ; type?
  [(cxt (type-eq type v1 v2) (and (hasType? cxt v1 type) (hasType? cxt v2 type))]
  [(_ _) #f])

(match-lambda** ; hasType?
  [(cxt _(type-eq type v1 v2)) (eqVal? cxt type v1 v2)] ;note we ignore the term
  [(_ _ _) #f])

(match-lambda** ; eqType?
  [(cxt (type-eq t1t t1a t1b) (type-eq t2t t2a t2b))
   (and (eqType? cxt t1t t2t) (eqVal? cxt t1t t1a t2a) (eqVal? cxt t1t t1b t2b))]
  [(_ _ _) #f])

(match-lambda** ; eqVal?
  [(cxt (type-eq t a b) _ _) #t] ;this is mysteriously always true and does not internalize
  [(_ _ _ _) #f]))
```

Equality Rules

```
(define equal-intro (pi (a type-type)
    (pi (x a) (close (type-eq a x x) (lambda (cxt) 'refl)))))

(define equal-induct (pi (a type-type)
  (pi (c (pi-ty (x a) (pi-ty (y a) (type-fun (type-eq a x y) type-type))))
  (lam (f (pi-ty (z a) (apps c z z 'refl)))
  (pi (m a)
  (pi (n a)
  (pi (p (type-eq a m n))
  (close (apps c m n p) (lambda (cxt) (app f m)))))))))) ; n is ignored!
```

An Actual Simple Proof

- ```
  (define not-bool (apps bool-elim type-bool #f #t))
  (define not-not-bool (lam (x type-bool) (app not-bool (app not-bool x))))
  (define id-bool (lam (x type-bool) x))
  ```

- ```
  ; not-not-is-id
  (define nnii-fam
     (lam (x type-bool) (type-eq type-bool (app id-bool x) (app not-not-bool x))))
  (hasType? '() nnii-fam (type-fun type-bool type-type))
  ```

- ```
  (define nnii-type (pi-ty (x type-bool) (app nnii-fam x)))
  (define nnii (pi (x type-bool)
       (apps bool-induct nnii-fam (apps refl type-bool #t) (apps refl type-bool #f) x)))
  ```

- ```
  (hasType? '() nnii nnii-type)
  ```

Extensionality, Axioms, and Computation

- (define eta-axiom (pi (a type-type) (pi (b type-type)
  (pi (f (type-fun a b))
  (pi (g (type-fun a b))
  (pi (prf (pi-ty (x a) (type-eq a (app f x) (app g x))))
  (trustme (type-eq (type-fun a b) f g) ‘eta-axiom)))))))

- (define nnii-extensional
  (type-eq (type-fun type-bool type-bool) id-bool not-not-bool))

- (define nnii-extensional-term
  (apps eta-axiom type-bool type-bool id-bool not-not-bool nnii))

- (hasType? '() nnii-extensional-term nnii-extensional)

The Homotopy Interpretation

Types are (Homotopy) Spaces, Elements are Points and Paths
(alternately: Formulae are Spaces, Constructions are Points)
[Awodey and Warren — 2006, Voevodsky — 2006]

Univalence Axiom
("Equivalence is Equivalent to Equality")
("All functions on the universe act continuously")
("Equivalent things are indiscernible things")
[Voevodsky 2009]

Earlier Work: The Groupoid Interpretation
[Hofmann and Streicher, 1995]

Aside: Nonstandard Models

Pick an equational theory, such as the following

Terms composed of:
$T, F, \wedge, \vee, \neg$

Formulae hold such as:
$A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$
$\neg \neg A = A$
etc.

The "standard model" is boolean logic — but it isn't the only one!

Aside: Nonstandard Models

Add the axiom that: exists A. $A = \neg A$

The this enforces a U such that $\neg U = U$.

One way to make this consistent is to have:
$T \vee U = T$, $F \vee U = U$, $T \wedge U = U$, $F \wedge U = F$

This axiom forces a nonstandard model.

Now consider the axiom: forall A. $A \vee \neg A = T$

This axiom rules out such a nonstandard model!

Aside: Nonstandard Models

We have nonstandard models of Peano Arithmetic (which are fun), Real Numbers (which are ***useful***), and soforth.

Nonstandard models typically come from ***extending*** or ***quotienting*** standard models.

We can "inflate" or "flatten" possibilities through additional axioms.

Nonstandard Models and Type Theory

Unique-Identity-Proofs:

```
(pi-ty (a type) (pi-ty (x a) (pi-ty (y a)
  (pi-ty (p (type-eq a x y))
  (pi-ty (q (type-eq a x y))
  (type-eq (type-eq a x y) p q))))))
```

Compatible with ITT, but not provable within it. Rules out certain models!

Question: What is the axiom that might *necessitate* such a model?

The Univalence Axiom

```
(define (fun-comp a f g) (lam (x a) (app f (app g a))))

(define (type-homotopy a p f g)
 (pi (x a) (type-eq (app p x) (app f x) (app g x))))

(define (type-isequiv a b f) (pair-ty
   (sig-ty (g (type-fun b a)) (type-homotopy b (lam (x a) b) (fun-comp b f g) (lam (x b) x)))
   (sig-ty (h (type-fun b a)) (type-homotopy a (lam (x b) a) (fun-comp a h f) (lam (x a) x)))))

(define (type-equiv a b)
  (sig-ty (f (type-fun a b)) (type-isequiv a b f)))

(define type-ua (pi-ty (a type-type) (pi-ty (b type-type)
  (type-equiv (type-equiv a b) (type-eq type-type a b)))))
```

Two New Judgments

- “P is a path at type T between points X and Y”
  (This fills in the place where we ignored the equality term before)
- “In type T, between points X and Y, P and Q are equal as paths”
  (And this fills in the other place where we ignored the equality term)

And … new computation rules

```
(define equal-induct (pi (a type-type)
  (pi (c (pi-ty (x a) (pi-ty (y a) (type-fun (type-eq a x y) type-type))))
  (lam (f (pi-ty (z a) (apps c z z 'refl)))
  (pi (m a)
  (pi (n a)
  (pi (p (type-eq a m n))
  (close (apps c m n p) (lambda (cxt) _?_?_?_?_ )))))))))
```

HoTT enables *synthetic* Mathematics

- In particular, but not only: Synthetic Homotopy Theory
- Additionally: more straightforward category theory
- Work is ongoing

References on Writing Dependent Type Systems

- A simple type-theoretic language: Mini-TT
  http://www.cse.chalmers.se/~bengt/papers/GKminiTT.pdf
- Simply Easy
  http://strictlypositive.org/Easy.pdf
- Simpler, Easier
  http://augustss.blogspot.com/2007/10/simpler-easier-in-recent-paper-simply.html
- PTS
  http://hub.darcs.net/dolio/pts
- Pi-Forall
  https://github.com/sweirich/pi-forall
- Mess:
  https://github.com/gbaz/mess

References on Type Theory

- Software Foundations (Pierce et. al, ongoing)
  http://www.cis.upenn.edu/~bcpierce/sf/current/toc.html

- Programming in Martin-Löf's Type Theory (Nordström, Petersson, and Smith, 1990)
  http://www.cse.chalmers.se/research/group/logic/book/book.pdf

- Papers of Per Martin-Löf
  https://github.com/michaelt/martin-lof
  (see in particular "Intuitionistic Type Theory," "Constructive mathematics and computer programming," and "On the Meanings of the Logical Constants and the Justification of Logical Laws").

- "A Framework for Defining Logics," (Harper, Honsell, and Plotkin, 1991)
  http://www.lfcs.inf.ed.ac.uk/reports/91/ECS-LFCS-91-162/

- "The Theory of LEGO," (Pollack, 1994)
  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.2610

- "Type Checking with Universes" (Harper and Pollack, 1991)
  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.8166

Dependently Typed Languages in Use

- Coq: https://coq.inria.fr/
- Agda: http://wiki.portal.chalmers.se/agda/pmwiki.php
- Idris: http://www.idris-lang.org/
- Nuprl: http://www.nuprl.org/

References on Homotopy Type Theory

- http://homotopytypetheory.org/
- http://homotopytypetheory.org/book/
- https://github.com/HoTT/HoTT
- https://github.com/HoTT/HoTT-Agda
- http://www.math.cornell.edu/~hatcher/AT/ATpage.html
  (A general textbook on Algebraic Topology including Homotopy Theory)

Appendix: Universe Checking

```
(define/match (check-u cxt x) ; something along these lines generates a dependency graph
  [(_ (lam-pi a at body)) ; this code is evocative but not yet fully correct
   (let*
       ([au (check-u cxt at)]
        [vu (check-u (cons (cons a (car au)) (cdr au)) a)]
        [bu (extend-cxt a (car vu) (cdr vu) (nvar newcxt)
            (check-u newcxt (body nvar)))])
     (cons (maxs (car bu) (car vu)) (cdr bu)))]
  [(_ (app fun arg))
   (let*
       ([funu (check-u cxt fun)]
        [argu (check-u (cdr funu) arg)])
     (cons (car funu) (cons (cons (car funu) (car argu)) (cdr argu))))]
  [(_ (? symbol? vname)) #:when (find-cxt vname cxt)
                        (cons vname cxt)]
  [(_ x) (cons 0 cxt)])
```