(Mis)Adventures in Programming Language Design: Anecdotes from a Slacker's Odyssey (So Far)

Adrian King

8 February 2016

Prologue:
Where Are All the Good Statically Typed Lisps?

Once Upon a Time...

- Lisp is cool.
- Static typing is cool.
- Where is the intersection between them?

- Lisp is usually understood to embody the *untyped* lambda calculus
- but could be broadly understood as a language:
  - whose syntax tree is easy to manipulate
  - has quote and eval

Inventing the Wheel (Any Excuse Will Do)

- There are Lisps with static typing:
  - Clojure
  - Racket
- but static typing not as well integrated with dynamic typing as you might hope.
- Guess I'll have to make my own!
- (I wanted to anyway.)
- But maybe I should learn something about programming language design first
  - including type systems...

Another Problem: I Hate Lisp

- I mean, I love Lisp
- but I hate parentheses.
- Humans are good at understanding infix operators
- but Lisp insists on prefix operators.
- Got to do something about that syntax.
- Not an original idea.
- My quest: discover the long-last land of the m-expressions
- or pretend I did.

Part 0:
The (Im)propriety of the S-Expression

One Data Structure to Rule Them All

```
()

atom

(oneItem)

(two items)

(list of three)

((embedded list)
 in a list)
```

- The s-expression is a general-purpose data structure.
- Lisp uses it for everything, including syntax trees.
- It resembles the Rose tree:

```
data RoseTree a =
  RoseTree a [RoseTree a]
```

except that s-expressions can be empty, but Rose trees can't.

Proper or Improper

- The s-expression can be thought of as:

```
data Sexp a =
  Nil |
  Atom a |
  Pair (Sexp a) (Sexp a)
```

where a is the type of atomic data.

- But the formulation on the left allows improper pairs, where the s-expression on the right is an atom.
- Improper pair notation: (a . b)
- Alternatively, proper s-expressions:

```
data Sexp a =
  Atom a | Liss [Sexp a]
```

Proper Rules

- The ability to form improper lists is more annoying than helpful.
- Need to handle improper case wherever you want to handle a list.
- But hard to avoid creating improper lists in a system without static types.
- The language I'm working on uses *proper* s-expressions as its initial syntax tree.

Of Course There's a Monad

- You can make a monad for either formulation of s-expression in the obvious way.
- But flattening (joining) doesn't flatten the original tree structure.
- (The same is true of the Rose tree.)
- It's not clear why you'd care about this.
- But, uh, Haskell.

Part 1:
Know Your Audience

Power to the People

- If I'm making up a programming language, I want it to be understandable by people like me (as of a few years ago, anyway).
- Someone who knows C, C++, or Java
- but not necessarily type theory or category theory.
- It's frustrating to know you need to say something (*this list is never empty, these two lists have the same length*), but you can't express it in the type system.
- Could read the documentation
- but nobody does.

Flamethrowers for Five-Year-Olds

- Existing dependently typed languages are aimed at mathematicians
- or at least programmers with a background in proof theory.

- Can you really give a powerful type system to someone without a serious mathematical background?
- And expect them to understand it?
- Use it?
- Like it?
- Not hurt themselves?
- We'll see...

Give the People What They Want

- Mostly, what they want is refinement types:

  ```
  l: List String $ len l > 0
  ```

- and some form of subtyping or implicit conversion (at least for refinements)
- and type/term inference where it's not too much trouble.
- I expect refinements cover most of what typical programmers want for increased expressivity
- but in the end, probably need the whole ball of wax
- that is, general propositions and proofs via Curry-Howard.

It's All in the Presentation

- Part of the trick of persuading people who consider themselves practical programmers to use more powerful type systems: don't tell them how powerful the systems are.

- Introduce programmers to a language by using it to do something (play Minesweeper, animate a bunch of dancing kittens)
- not prove something
- not demonstrate the versatility of category theory.

More Programming, Less Proving

- You don't need to know a language completely to use it.
- You can teach complex concepts without telling people what they are.
- Unveil the really good stuff when people are ready for it.
- “Hello, World” > “this diagram commutes”.

Part 2:
Aesthetic Considerations

Infix Syntax without Keywords

- I've come up with generic rules for infix syntax.
- The syntax rules define a translation to s-expressions, not a language.
- It's language-dependent how the language treats the s-expressions.
- Implemented: a logic programming language (like Prolog) that uses the syntax.
- Planned: a dependently typed language with first-class type refinements.

Types of Operators

- Three groupings of operators by precedence:
  - Right-associative unary prefix (highest precedence)
  - Left-associative unary prefix
  - Left- and right-associative binary infix (lowest precedence)

- Operator precedence and associativity are based only on the text of each operator (not declared).
- This makes grammar context-free—you can read code without tracking down the operator declarations.

The Long Shadow of ASCII

- I assume an aesthetic preference for ASCII operators.
- Unicode is beautiful, but often visually ambiguous and still too poorly supported.

- You can use arbitrary Unicode punctuation characters in operators, but there is no default precedence if an infix operator doesn't start with an ASCII character (use parentheses).

Operator Syntax Rules

- Right-associative prefix operators begin with a backquote.
- Left-associative prefix operators begin with an alphanumeric character. They don't need to have operands, so they can be used as operands.
- Infix operators start with punctuation characters (basically anything that's not alphanumeric, whitespace, or a delimiter like parentheses or quotation marks).
- Example: `` `-2 * cos x ``

Binary Operator Precedence and Associativity

```
 ;
 ~
 |
 $
 \
 ,
 :
 ?
 &
 !
< = >
 #
+ -
* / %
 ^
 `
 @
 .
```

- Binary operator precedence is determined by the operator's first character.
- To the left are precedence groups from lowest to highest.
- An operator is right-associative if it ends with one of:

  `; ~ | \ , : ? & ! ^`

- A left-associative unary operator prefixed with backquote turns into a binary operator with the precedence shown for ` (example: `a `or b`).

Groupers

- Subexpressions can be grouped in ( ), [ ], or { }.
- Parentheses just control precedence.
- Square brackets contain whitespace-separated lists of expressions (operators are not special). Example:

  [1 2 3 + a b c ?]

- Semicolons are inferred between curly braces when a newline occurs between left-associative unaries (idea stolen from Scala).
- Semicolons are expected to separate declarations or statements.

Whitespace: The Final Frontier

- Another way of grouping expressions is to omit whitespace within them (when surrounding expressions contain whitespace).
- People do this informally in other languages to show precedence:

```
a + b*c
```

- The rule is that an infix operator with no whitespace on either side, or a prefix operator with no space following, binds tighter:

```
a+b * c
```

is equivalent to:

```
(a + b) * c
```

The Big Four Operators

	Term	Type
Function	var\ body (v: T)\ body	In? Out (a: A)? B a
Conjunction (Tuple)	first, second	A **&** B (a: A) & B a

- Commonly used operators from type theory are single-ASCII-character infix.
- Where's disjunction? Less commonly used, may come in more than one variety, and there's more than one constructor for terms.

Part 3:
Type Theory in a Big Fat Hurry

The Usual Suspects, Type-Theoretically

- Intuitionistic type theory, originally developed by logicians (not programmers), has a number of variants known by a variety of names.
- Uses dependent types and implements a constructive logic (no law of excluded middle).
- Dependent function types:

(++): (a: Type)? (m n: Nat)? Vec a m? Vec a n? Vec a (m + n)

- Dependent pair (product) types:

$$\frac{A\text{: Type} \qquad a\text{: A} \qquad B\text{: A? Type} \qquad b\text{: B a}}{a,b\text{: (a: A) \& B a}}$$

Other More-or-Less Standard Built-In Data Types

- 0 or False or ⊥: the empty type (has no constructors).
- 1 or True or Unit: the type with just one constructor.
- 2 or Boolean: the type with two constructors.
- Coproduct (sum, disjunction, tagged union, Either) types:
  - Left a: A \/ B
  - Right b: A \/ B
- Natural numbers (constructors: zero and successor).
- *W*-types (indexed inductive types).

Trickier Types

- Propositions: like data types, but less so.
- Equality types: the type of *propositions* that two things are equal:

  $a =_A b$ or $\mathrm{Id}_A(a,b)$

- Equalities are a world of trouble, or fun (see Homotopy Type Theory).
- Universe types: U_n
- If you decide that the type of types (Type) is an element of itself, you run into paradoxes.
- You can fix the paradoxes by making types at one level belong to the next higher level.

Curry and Howard and Their Famous Correspondence

Dear Dr. Curry,
I'm your #1 fan! Please write me back!
Sincerely,
William Alvin Howard

- Curry and Howard were among a number of logicians who figured this out.

- A type can be understood as a logical proposition. Values of the type are proofs of the proposition.
- Function types represent predicates or implications.

Dear Dr. Howard,
What do you want? If you can't say something constructive, just go away!
Sincerely,
Haskell Curry

Inductive Principles

- Each type comes with introduction (type constructor) and elimination rules.
- The elimination rules take the form of inductive principles, that is, magic functions that recurse (dependently) on values of the type. They are the *only* way to do recursion.

- For example, the type of the inductive principle for Nat (the natural numbers) looks like:

```
(P: Nat? Type)?
  P 0?
  ((n: Nat)? P n? P (S n))?
  ((n: Nat)? P n)
```

where Type is an abbreviation for some U_n, and 0 and S are Nat's constructors.

Part 4:
Wait, What?
Why Type Theory Is a Lie

But... But...

- You may have noticed that this presentation of type theory doesn't look like any programming language you're used to (even a language that uses dependent types).
- The theory that underlies *usable* programming languages differs in several respects from what mathematicians like to think of as type theory.

Inductive Types

- Real languages don't just have built-in types; they let you define your own types.
- “Inductive types” is the usual name for the generalization of algebraic data types (sums of product types, like Haskell's) to dependent types.
- The mathematical type theory types I presented earlier can be used to construct types isomorphic to inductive types.
- But two isomorphic inductive types are not usually considered equal.

Pattern Matching vs. Inductive Principles

- No real language makes you program in raw inductive principles.
- Instead, you can pattern-match on inductive type values. This is more flexible and less dependent on the exact phrasing of an inductive principle.
- Inductive principles (used in proofs) in tools like Coq are defined in terms of pattern matching.
- Coq inductive principles are correct because they typecheck!

General Recursion and Function Termination Checks

- Induction principles are functions that are guaranteed to terminate.
- Real languages let you combine pattern matching with general recursion.
- General recursion isn't guaranteed to terminate.
- But a function that doesn't terminate is not a valid proof of an an implication (so isn't a valid element of its function type).
- Need to combine general recursion with a termination check (usually that an argument decreases).

Strict Positivity

- It turns out that if an inductive type constructor parameter type includes the type being defined as an input, you can use it to create functions that recurse indefinitely.
- So the type being defined is not allowed in that position. This is called the “strict positivity” constraint.

Universe Polymorphism

- I said before that there isn't really a single type of types, but a hierarchy of type universes of different levels, U_n.
- You often want polymorphism in universe levels, but it's a pain to do it explicitly.
- Many programming languages provide universe polymorphism by default. You can pretend that there's a single type of types (often named `Type`), and the language automatically figures out what universe level to use (or complains if no such level exists).

Propositons and the Paradoxes That Aren't

- In addition to the predicative data type universes U_n, some languages also provide an impredicative universe of propositions (often named `Prop`).
- Propositions have just one level.
- A proposition about propositions is just a proposition, not a proposition at the next higher level.
- Paradoxes don't occur because you can't pattern match on a proposition value.

Part 5:
Language Design in Layers

One Thing at a Time

- Doing everything at once is as bad an idea in software as in the rest of life.
- So compile a language in phases (at least conceptually).

A Language Built in Layers

source (with infix operators)	⇒ s-expressions
s-expressions	⇒ s-expressions (macros)
s-expressions	⇒ typed intermediate language with named variables (more macros)
typed intermediate language with named variables	⇒ typed intermediate language with DeBruijn indices
typed intermediate language with DeBruijn indices	⇒ typed intermediate language with DeBruijn indices (typechecking and inference)
typed intermediate language with DeBruijn indices	⇒ untyped lambda calculus (augmented with type information as necessary for reflection/pattern matching)
untyped lambda calculus	⇒ interpreted, or compiled to machine code

A Type System Built in Layers

- If you start loading up a type system with novel features, it can be hard to figure out whether it's consistent.
- Idea: incrementally translate a more complicated type system into one that's well understood.

Program with refinements, subtyping, inference

↓ *Convert refinements to dependent pairs*

Program with subtyping and inference

↓ *Insert coercions to subtypes*

Program with inference

↓ *Infer missing terms*

Program in a well-studied type system

- Properties of the whole language are properties of the phases plus those of the final type system.

Part 6:
Consistently Inconsistent

Effects: Packaging vs. Tracking

- The Haskell philosophy is to package code with side effects into category-theory-inspired structures.
- This works great, but is intimidating to newcomers to the language.
- Effects tracking is clearly necessary if you want to distinguish code that corresponds to a valid logic from code that doesn't.
- But, except when you want explicit control over composition of effects, the category-theoretic approach may be overkill.

Tame Types and Wild Types

- It may be enough (most of the time) to distinguish between tame types (those that represent valid proofs under Curry-Howard) and wild types (of code that does something but doesn't represent a valid proof).
- Every valid term has a wild type; only well-behaved terms have a tame type.
- Judgments:
  - M: T when M is well-behaved (is a valid proof of T; always terminates, reduces to the same value).
  - M:! T when M, *if* it returns, returns a value of T.

Wild Things

- M:! T if M: T
- (x: A)\! M: A?! B when it can be shown that the function, if it returns, returns a B.
- (x: A)\ M: A? B only when evaluation of M can be proven to terminate (always with the same value for a given x).
- (fun:! A?! B) (arg:! A):! B

Tame Things That Look Wild

- Not everything we think of as an effect makes the code that invokes it wild.
- In particular, logging or terminal output (if it can't cause indefinite delays) does not prevent code the code that invokes it from returning, or change the value it computes.

Taming a Function

- If you have:

  f: A?! B

  you can get

  f′: A? B

  from it by supplying a proof that it always terminates and always takes equals to equals.

- How to construct those proofs? Still working on it...

Permissive Positivity

- The strict positivity condition, as usually phrased, is stricter than necessary.
- No paradox arises from the *existence* of a constructor parameter that violates the condition.
- Trouble occurs only when the constructor in question is *used*.
- (Maybe someone can help me figure out *what* uses are problematic...)
- Code that uses such a constructor can be given a wild type.

Part 7:
Code You Can Play With

Homoiconicity

- The point of using s-expressions as a program representation is to make it easy for code to manipulate code.
- The first macro pass translates s-expressions to s-expressions.
- You can also manipulate code at a lower level by constructing terms directly in a typed lambda calculus.
- You can `quote` or `eval` either s-expressions or lambda calculus expressions.

Code That Talks about Itself and to Itself

- Beyond homoiconicity, the point of dependent types is to allow code to talk about itself.
- The more self-referential and self-descriptive a language is, the more powerful it is.

Part 8:
Code You Can't Play With

Reflecting Badly

- If self-reference is a good thing in a language, then runtime reflection must be a good thing.
- But some types of reflection lead to logical inconsistencies.

Functoriality

- A very desirable type system property is *functoriality*: a given function takes equal arguments to equal results.
- A logic without functoriality would be pretty useful for reasoning about programs.

- Another desirable property is *function extensionality*: two functions that take equal arguments to equal results are considered equal.

You Can't Inspect Functions

- Given function extensionality, the ability to inspect the code of a function at runtime would violate functoriality.

- Consider these two identity functions:

```
a\ a
a\ (b\ b) a
```

- If you can see their implementations, you can distinguish them, and construct a boolean function `f` that returns a different value for each. But function extensionality says they're equal, so (by functoriality) `f` must return the same result for each.

You Can't Inspect Types

- Inspecting types can be dangerous, too.
- Consider the types:

  (A: Type)? (m n: Nat)? Vec A m+n

  and

  (A: Type)? (m n: Nat)? Vec A n+m

- m+n = n+m, so the types are equal. But their structure isn't.

So How Can You eval?

- You might think there is a contradiction between the inability to inspect code and the ability to create code at runtime.
- For example:

```
evalquote (a\ (b\ b) a)
```

- But in fact, eval doesn't provide any proof that it created a function identical to what you gave it. It's free to act as if you said:

```
evalquote (a\ a)
```

instead. It only needs to give you something *extensionally* equal to its input.

Part 9:
Natural Deduction

Natural Deduction Is Logic Programming

- Rules in the natural deduction style:

$$\frac{\Gamma \vdash A \colon U_i \qquad \Gamma, x \colon A \vdash B \colon U_i}{\Gamma \vdash ((x \colon A) \to B) : U_i}$$

  can be translated directly into Prolog and executed.
- But Prolog is ugly and full of 1970s imperative crap.
- I've been playing with a logic programming language, Rofl, that uses the syntax rules I described earlier.
- Rofl has no global state or imperative features.

Rules in Rofl

- This:

$$\frac{\Gamma \vdash A\colon U_i \quad \Gamma, x\colon A \vdash B\colon U_i}{\Gamma \vdash ((x\colon A) \to B) : U_i}$$

  in Rofl:

```
(ctx |~ ((x: a)? b): U i) ~|
  (ctx |~ a: U i), ((x: a) :: ctx |~ b: U i)
```

- Turn the underline into ~|, put the conclusion first, and separate the premises with a comma.

Logic Programming vs. the Real World

- Logic programming is cool because you can run the rules forwards to typecheck a term, or backwards to infer a term.
- But...
- Improperly constrained variables can result in very large search spaces.
- Naked type theory rules won't give you any error messages if they fail.
- Specialized constraint solvers may be a lot faster for some queries.
- Still, this is a fun way to prototype.

Part 10:
Always Distinguish?

Breadcrumbs

- I keep running across recommendations against type system that collapse certain distinctions.
- For example:
  - Equirecursion (as opposed to isorecursion)
  - Subsumptive (as opposed to coercive) subtyping
  - Jugmental equality where a propositional equality would do (because this erases explicit paths between things that are equal)
- Keeping track of such distinctions makes for intermediate code full of coercion functions that don't do anything.
- Should I care?