Typed Lisp, A Primer

Musa Al-hassy

2019-08-21

Contents

Abstract

Let's explore Lisp's fine-grained type hierarchy!

We begin with a shallow comparison to Haskell, a rapid tour of type theory, try in vain to defend dynamic approaches, give a somewhat humorous account of history, note that you've been bamboozled —type's have always been there—, then go into technical details of some Lisp types, and finally conclude by showing how *macros permit typing*.

Goals for this article:

1. Multiple examples of type constructions in Lisp.
2. Comparing Lisp type systems with modern languages, such as Haskell.
3. Show how algebraic polymorphic types like `Pair` and `Maybe` can be defined in Lisp. Including heterogeneously typed lists!
4. Convey a passion for an elegant language.
5. Augment Lisp with functional Haskell-like type declarations ;-)

Unless suggested otherwise, the phrase "Lisp" refers to Common Lisp as supported by Emacs Lisp. As such, the resulting discussion is applicable to a number of Lisp dialects —I'm ignoring editing types such as buffers and keymaps, for now.

1 "Loving Haskell & Lisp is Inconsistent"

I have convinced a number of my peers to use Emacs/Spacemacs/Doom-emacs, but my efforts to get them to even consider trying Lisp have been met with staunch rejection. These peers are familiar with Haskell, and almost all know Agda, so you'd think they'd be willing to try Lisp —it's there, part of their editor— but the superficial chasm in terms of syntax and types is more than enough apparently. In this article, I aim to explore the type system of (Emacs) Lisp and occasionally make comparisons to Haskell. Perhaps in the end some of my Haskell peers would be willing to try it out.

https://imgs.xkcd.com/comics/lisp_cycles.png

- !! I almost never use Haskell for any day-to-day dealings.

  !! The ideas expressed by its community are why I try to keep updated on the language.

- !! No one around me knows anything about Lisp, but they dislike it due to the parens.

  !! I love it and use it for Emacs configuration and recently to prototype my PhD research.

- !! I love that I can express a complicated procedure compactly in both by using zips, unzips, filters, and maps `( _ )`

  – Lately, in Lisp, I'll write a nested loop (gasp!) then, for fun, try to make it a one-liner! Sometimes, I actually think the loop formulation is clearer and I leave it as a loop —Breaking news: Two Haskell readers just died.

  https://i.stack.imgur.com/jvSOG.png

- **What I like and why:**

Haskell	⇒	Executable category theory; compact & eloquent
Lisp	⇒	Extensible language; malleable, uniform, beautiful

- **Documentation?**

Haskell	⇒	Hoogle; can search by type alone!
Emacs Lisp	⇒	Self-documenting; `M-x apropos`

- **How has using the language affected me?**

Haskell	I almost always think in-terms compoistionality, functors, & currying
Lisp	Documentation strings, units tests, and metaprogramming are second nature

It may not be entirely accurate to say that Lisp's type system is more expressive than Haskell's as it's orthogonal in many respects; although it is closer to that of Liquid Haskell.

2 Why Bother with Types? A Terse Tutorial on Type Systems

Types allow us to treat objects according a similar structure or interface. Unlike Haskell and other statically typed systems, in Lisp we have that types can overlap. As such, here's our working definition.

A **type** is a collection of possible objects.
To say "e has type τ" one writes $e : \tau$, or in Lisp: `(typep e ' τ )`.

Haskellers and others may append to this definition the following, which we will not bother with: *Type membership is determined by inspecting syntactic structure and so is decidable.*

> !! Typing is one of the simplest forms of "assertion-comments": Documenting a property of your code in a way that the machine can verify.
>
> If you're gonna comment on what kind of thing you're working with, why not have the comment checked by the machine.

Table 1: Lisp's type hierarchy is a "complemented lattice"

Common types	`integer, number, string, keyword, array, cons, list, vector, macro, function, atom`
Top	`t` has everything as an element
Unit	`null` has one element named `nil`
Bottom	`nil` has no elements at all
Union	`(or` τ_0 τ_1 ... τ_n `)` has elements any element in any type τ_i
Intersection	`(and` τ_0 τ_1 ... τ_n `)` has elements that are in all the types τ_i
Complement	`(not` τ `)` has elements that are *not* of type τ
Enumeration	`(member x` $_0$ ... `x` $_n$ `)` is the type consisting of only the elements `x` $_i$
Singleton	`(eql x)` is the type with only the element `x`
Comprehension	`(satisfies p)` is the type of values that satisfy predicate `p`

Let's see some examples:

```
;; The universal type "t", has everything as its value.
(typep 'x 't) ;;  ⇒  true
(typep 12 't) ;;  ⇒  true

;; The empty type: nil
(typep 'x 'nil) ;;  ⇒  false; nil has no values.

;; The type "null" contains the one value "nil".
(typep nil 'null) ;;  ⇒  true
(typep () 'null)  ;;  ⇒  true

;; "(eql x)" is the singelton type consisting of only x.
(typep 3 '(eql 3)) ;;  ⇒  true
(typep 4 '(eql 3)) ;;  ⇒  false
```

```
;; "(member x 0 ... x n )" denotes the enumerated type consisting of only the x i .
(typep 3 '(member 3 x "c"))  ;;  ⇒  true
(typep 'x '(member 3 x "c")) ;;  ⇒  true
(typep 'y '(member 3 x "c")) ;;  ⇒  false

;; "(satisfies p)" is the type of values that satisfy predicate p.
(typep 12 '(satisfies (lambda (x) (oddp x)))) ;;  ⇒  false
(typep 12 '(satisfies evenp) )                ;;  ⇒  true

;; Computation rule for comprehension types.
;; (typep x '(satisfies p))  ≈  (if (p x) t nil)
```

Here's a convenient one: `(booleanp x)` ≈ `(typep x '(member t nil))`.

```
(booleanp 2)   ;;  ⇒  false
(booleanp nil) ;;  ⇒  true
```

Strongly typed languages like Haskell allow only a number of the type formers listed above. For example, Haskell does not allow unions but instead offers so-called sum types. Moreover, unlike Haskell, Lisp is non-parametric: We may pick a branch of computation according to the type of a value. Such case analysis is available in languages such as C# —c.f., is is as or is as is. Finally, it is important to realise that `cons` is a monomorphic type —it just means an (arbitrary) element consisting of two parts called `car` and `cdr`— we show how to form a polymorphic product type below.

We may ask for *the* 'primitive type' of an object; which is the simplest built-in type that it belongs to, such as integer, string, cons, symbol, record, subr, and a few others. As such, *Lisp objects come with an intrinsic primitive type*; e.g., `'(1 "2" 'three)` is a list and can only be treated as a value of another type if an explicit coercion is used. In Lisp, rather than variables, it is values that are associated with a type. One may optionally declare the types of variables, like in OCaml.

Lisp (primitive) types are inferred!
"Values have types, not variables." —Paul Graham, ANSI Common Lisp

Let's review some important features of type systems and how they manifest themselves in Lisp.

2.1 Obtaining & Checking Types

The typing relationship ":" is usually deterministic in its second argument for static languages: `e` : $\tau \wedge$ `e` : $\tau' \Rightarrow \tau \approx \tau'$. However this is not the case with Lisp's `typep`.

Table 2: Where are the types & *when* are they checked?

Style	Definition	Examples
Static	Variables have a fixed type; compile time	Haskell & C#
Dynamic	Values have a fixed type; runtime	Lisp & Smalltalk

In some sense, dynamic languages make it easy to produce polymorphic functions. Ironically, the previous sentences is only meaningful if you acknowledge the importance of types and type variables.

In Lisp, types are inferred and needn't be declared. However, the declaration serves as a nice documentation to further readers ;-)

```
(setq ellew 314)
(type-of ellew) ;;  ⇒  integer

(setq ellew "oh my")
(type-of ellew) ;;  ⇒  string
```

- The `type-of` function returns the type of a given object.

- Re variables: Static $\Rightarrow$ only values can change; dynamic $\Rightarrow$ both values and types change.

We may check the type of an item using `typep`, whose second argument is a "type specifiers" —an expressions whose value denotes a type; e.g., the `or` expression below forms a 'union type'.

There's also `check-type`: It's like `typep` but instead of yielding true or false, it stays quiet in the former and signals a type error in the latter.

```
(check-type 12 integer)               ;;  ⇒  nil, i.e., no error
(check-type 12   (or symbol integer)) ;; nil; i.e., no error
(check-type "12" (or symbol integer)) ;; Crash: Type error!
```

In summary:

```
(equal  τ  (type-of e))   ≈   (typep e  τ )
(check-type e  τ )        ≈   (unless (typep e ' τ ) (error " ··· "))
```

(Note: `(unless x y)` $\approx$ `(when (not x) y)` .)

2.2 Statics & Dynamics of Lisp

> Types are the central organising principle of the theory of programming languages. Language features are manifestations of type structure. The syntax of a language is governed by the constructs that define its types, and its semantics is determined by the interactions among those constructs.
>
> — Robert Harper, Practical Foundations for Programming Languages

Besides atoms like numbers and strings, the only way to form new terms in Lisp is using "modus ponens", or "function application". Here's a first approximation: One reads such a fraction as follows: If each part of the numerator —the 'hypothesises'— is true, then so is the denominator —the 'conclusion'.

An *abstract syntax tree*, or 'AST', is a tree with operators for branches and arguments for children. A tree is of kind τ if the topmost branching operator has τ as its resulting type. Here's an improved rule:

A Lisp top-level then may execute or interpret such a form to obtain a value: When we write `e` at a top-level, it is essentially `(eval e)` that is invoked.

However, we may also protect against evaluation.

We have the following execution rules, where '' denotes "reduces to".

A conceptual model of Lisp is `eval`.

2.3 Variable Scope

There's also the matter of "scope", or 'life time', of a variable.

Table 3: Local variables temporarily mask global names ...

Style	Definition	Examples
Lexical	... only in visible code	Nearly every language!
Dynamic	... every place imaginable	Bash, Perl, & allowable in some Lisps

That is, dynamic scope means a local variable not only acts as a global variable for the rest of the scope but it does so even in the definitions of pre-defined methods being invoked in the scope.

```
(setq it "bye")
(defun go () it)
(let ((it 3)) (go)) ;;  ⇒  3; even though "it" does not occur textually!

;; Temporarily enable lexical binding in Emacs Lisp
(setq lexical-binding t)
(let ((it 3)) (go)) ;;  ⇒  bye; as most languages would act
```

Dynamic scope lets bindings leak down into all constituents in its wake.

That is fantastic when we want to do unit tests involving utilities with side-effects: We simply locally re-define the side-effect component to, say, do nothing. ( )

2.4 Casts & Coercions

Table 4: The frequency of implicit type coercions

Style	Definition	Examples
Strong	Almost never	Lisp & Haskell
Weak	Try as best as possible	JavaScript & C

Strong systems will not accidentally coerce terms.

Lisp has a coerce form; but coercion semantics is generally unsound in any language and so should be used with tremendous caution. (Though Haskell has some sensible coercions as well as unsafe one.) We have a magical way to turn elements of type α to elements of type β. Some languages call this *type casting*.

Here's a cute example.

```
(coerce '(76 105 115 112) 'string) ;;  ⇒  Lisp
```

2.5 Type Annotations

We may perform type annotations using the form `the`; e.g., the Haskell expression `(1 :: Int) + 2` checks the type annotation, and, if it passes, yields the value and the expression is computed. Likewise, `(the type name)` yields `name` provided it has type `type`.

```
(+ (the integer 1)
   (the integer 2)) ;;  ⇒  3

(+ (the integer 1)
   (the integer "2")) ;;  ⇒  Type error.
```

Computationally, using `or` as a control structure for lazy sequencing with left-unit `nil`:

$$\texttt{(the } \tau \texttt{ e)} \quad \approx \quad \texttt{(or (check-type e } \tau \texttt{) e)}$$

2.6 Type-directed Computations

Sometimes a value can be one of several types. This is specified using union types; nested unions are essentially flattened —which is a property of 'or', as we shall come to see.

```
(typep 12 'integer)  ;;  ⇒  t
(typep 'a 'symbol)   ;;  ⇒  t

(setq woah 12)
(typep woah '(or integer symbol)) ;;  ⇒  t

(setq woah 'nice)
(typep woah '(or integer symbol)) ;;  ⇒  t
```

When given a union type, we may want to *compute according to the type of a value.*

- Case along the possible types using `typecase`.
- This returns a `nil` when no case fits; use `etypecase` to have an error instead of `nil`.

```
(typecase woah
  (integer  (+1 woah))
  (symbol   'cool)
  (t        "yikes"))
```

2.7 Type Specifiers: On the nature of types in Lisp

> Types are not objects in Common Lisp. There is no object that corresponds to the type `integer`, for example. What we get from a function like `type-of`, and give as an argument to a function like `typep`, is not a type, but a type specifier. A type specifier is the name of a type. —Paul Graham, ANSI Common Lisp

Type specifiers are essentially transformed into predicates as follows.

```
(typep x ' τ )                ≈ ( τ p x)  ;; E.g., τ ≈ integer
(typep x '(and  τ 1  ...  τ n ))    ≈ (and (typep x  τ 1 ) ... (typep x  τ n ))
(typep x '(or   τ 1  ...  τ n ))    ≈ (or (typep x  τ 1 ) ... (typep x  τ n ))
(typep x '(not  τ ))          ≈ (not (typep x  τ ))
(typep x '(member e 1  ... e n )) ≈ (or (eql x e 1 ) ... (eql x e n ))
(typep x '(satisfies p))      ≈ (p x)
```

Type specifiers are thus essentially 'characteristic functions' from mathematics.

2.8 Making New Types with `deftype`

If we use a type specifier often, we may wish to abbreviate it using the deftype macro —it is like `defmacro` but expands into a type specifier instead of an expression.

We can define new types that will then work with `typecase` and friends as follows:

1. Define a predicate `my-type-p`.
2. Test it out to ensure only the elements you want satisfy it.
3. Register it using deftype.

   You could just do number 3 directly, but it's useful to have the predicate form of a type descriptor.

For example, here's the three steps for a type of lists of numbers drawn from `(-..9]`.

```
;; Make the predicate
(defun small-number-seq-p (thing)
  (and (sequencep thing)
       (every #'numberp thing)
       (every (lambda (x) (< x 10)) thing)))

;; Test it
(setq yes '(1 2  4))
(setq no  '(1 20 4))
(small-number-seq-p yes) ;;  ⇒  t

;; Register it
(deftype small-number-seq ()
  '(satisfies small-number-seq-p))

;; Use it
(typep yes 'small-number-seq) ;;  ⇒  true
(typep no 'small-number-seq)  ;;  ⇒  false
```

Arguments are processed the same as for `defmacro` except that optional arguments without explicit defaults use `*` instead of `nil` as the default value. From the deftype docs, here are some examples:

```
(cl-deftype null () '(satisfies null))     ; predefined
(cl-deftype list () '(or null cons))       ; predefined

(cl-deftype unsigned-byte (&optional bits)
  (list 'integer 0 (if (eq bits '*) bits (1- (lsh 1 bits)))))

;; Some equivalences
(unsigned-byte 8)  | (integer 0 255)
(unsigned-byte)    | (integer 0 *)
unsigned-byte      | (integer 0 *)
```

- Notice that type specifiers essentially live in their own namespace; e.g., `null` is the predicate that checks if a list is empty yet `null` is the type specifying such lists.

Let's form a type of pairs directly —which is not ideal! This is a polymorphic datatype: It takes two type arguments, called `a` and `b` below.

```
(deftype pair (a b &optional type)
  `(satisfies (lambda (x) (and
      (consp x)
      (typep (car x) (quote ,a))
      (typep (cdr x) (quote ,b))))))

(typep '("x" . 2) '(pair string integer)) ;;  ⇒  true
(typep '("x" . 2) '(pair symbol integer)) ;;  ⇒  false
(typep nil '(pair integer integer))       ;;  ⇒  false
(typep 23 '(pair integer integer))        ;;  ⇒  false

(setq ss "nice" nn 114)
(typep `(,ss . ,nn) '(pair string integer)) ;;  ⇒  true
(typep (cons ss nn) '(pair string integer)) ;;  ⇒  true

;; The following are false since ss and nn are quoted symbols!
(typep '(ss . nn)    '(pair string integer)) ;;  ⇒  false
(typep `(cons ss nn) '(pair string integer)) ;;  ⇒  false
```

Exercise: Define the polymorphic `maybe` type such that `(maybe` τ `)` has elements being either `nil` or a value of τ .

Let's define type `list-of` such that `(list-of` τ `)` is the type of lists whose elements are all values of type τ .

```
;; Make the predicate
(defun list-of-p ( τ  thing)
  (and (listp thing) (every (lambda (x) (typep x  τ )) thing)))

;; Test it
(list-of-p 'integer '(1 2   3)) ;;  ⇒  true
(list-of-p 'integer '(1 two 3)) ;;  ⇒  false
(list-of-p 'string '())         ;;  ⇒  true
(list-of-p 'string '(no))       ;;  ⇒  false

;; Register it
```

```
(deftype list-of ( τ )
  `(satisfies (lambda (thing) (list-of-p (quote , τ ) thing))))

;; Use it

(typep '(1 2  ) 'list) ;;  ⇒  true
(typep '(1 two) 'list) ;;  ⇒  true

(typep '(1 2)   '(list-of integer)) ;;  ⇒  true
(typep '(1 "2") '(list-of string))  ;;  ⇒  false
(typep '(1 "2") '(list-of (or integer string)))  ;;  ⇒  true
```

Notice that by the last example we can **control the degree of heterogeneity** in our lists! So cool!

Here's some more exercises. The first should be nearly trivial, the second a bit more work, and the last two have made me #sad.

1. Define a type `(rose` τ `)` whose elements are either τ values or rose trees of type τ.
2. Define a type `record` so that `(record` $\tau_1 \ \ldots \ \tau_n$ `)` denotes a record type whose i component has type τ_i.
3. Define a type constructor such that, for example, `(` τ `(pair integer` τ `)` denotes the type of pairs where the first components are integers and the second components all have the same type τ, but we do not know which one.

   My idea was to let τ denote the type of the first occurrence of a value at that location, then all subsequent checks now refer to this value of τ.

   Sadly, I could not define this type :'(

   Upon further reading, this may be doable using a variable watcher.
4. Produce a record for monoids and keep-track of the monoid instances produced. Define a the predicate `(monoid` τ `)` to check if any of the monoid instances has τ as its carrier type. In this way we could simulate Haskell typeclasses.

Let me know if you do cool things!

2.9 Algebraic Data Types a la Haskell

Consider the Haskell expression type, example, and integer evaluator.

```
data Expr a = Var a | Expr a :+: Expr a | Neg (Expr a) deriving Show

ex :: Expr Int
ex = Var 5 :+: (Var 6 :+: Neg (Var 7))

int :: Expr Int -> Int
int (Var n)    = n
int (l :+: r)  = int l + int r
int (Neg e)    = - (int e)

{- int ex  ⇒  4 -}
```

If we view a constructor declaration `C a` $_1 \ \ldots$ `a` $_n$ with superfluous parenthesis as `(C a` $_1 \ \ldots$ `a` $_n$ `)`, then a translation to Lisp immediately suggests itself:

A nearly direct translation follows.

```
(defun exprp ( τ  thing)
    (pcase thing
       (`(var ,n)    (typep n  τ ))
       (`(add ,l ,r) (and (exprp  τ  l) (exprp  τ  r)))
       (`(neg ,e)    (exprp  τ  e))))

(setq ex '(add (var 5) (add (var 6) (neg (var 7)))))
(exprp 'integer ex) ;;  ⇒  true

; This declarion "declare-type" is defined near the end of this article.
(declare-type int : (expr-of integer) integer)
(defun int (thing)
    (pcase thing
       (`(var ,n)    n)
       (`(add ,l ,r) (+ (int l) (int r)))
       (`(neg ,e)    (- (int e)))))

(int ex) ;;  ⇒  4
```

There are of-course much better ways to do this in Lisp; e.g., use `identity, +, -` in-place of the `var, add, neg` tags to produce "syntax that carries its semantics" or express the interpreter `int` as a one liner by replacing the formal tags with their interpretations then invoking Lisps `eval`. I doubt either of these are new ideas, but the merit of the former seems neat —at a first glance, at least.

Support for ADTs in Common Lisp along with seemingly less clunky pattern matching can be found here —which I have only briefly looked at.

The Haskell presentation has type-checking baked into it, yet our Lisp interpreter `int` does not! This seems terribly worrying, but that `declare-type` declaration actually handles type checking for us!

```
;; Register the type
(deftype expr-of ( τ )
  `(satisfies (lambda (thing) (exprp (quote , τ ) thing))))

;; Try it out
(typep '(1 2)   '(expr-of integer)) ;;  ⇒  nil
(typep ex   '(expr-of integer))     ;;  ⇒  true

;; This invocation, for example, now yields a helpful error message.
(int '(var 6 4))
;;
;;  ⇒  int: Type mismatch! Expected (expr-of integer) for argument 0  Given cons (var 6 4).
;;
;; Which is reasonable since the `var' constructor only takes a single argument.
```

Notice that invalid cases yield a helpful (run-time) error message!

3 In Defence of Being Dynamically Checked

Lisp gets a bad rap for being untyped; let's clarify this issue further!

It is important to realise that nearly every language is typed —albeit the checking may happen at different stages— and so, as Benjamin Pierce says: *Terms like "dynamically typed" are arguably misnomers and should probably be replaced by "dynamically checked," but the usage is standard.*

In particular, dynamically typed is *not* synonymous with untyped, though some people use it that way since nearly every language is typed —possibly with a single anonymous type.

Some people in the Haskell community, which I love, say things like *"if it typechecks, ship it"* which is true more often than not, but it leads some people to avoid producing unit tests. For example, the following type checks but should be unit tested.

```
mcbride :: [Int] -> Int
mcbride xs = if null xs then head xs else 666
```

Regardless, I love static type checking and static analysis in general. However, the shift to a dynamically checked setting has resulted in greater interest in unit testing. For example, Haskell's solution to effectful computation is delimited by types, as any Haskeller will proudly say (myself included); but ask how are such computations unit tested and the room is silent (myself included).

Interestingly some unit tests check the typing of inputs and output, which is a mechanical process with no unknowns and so it should be possible to produce a syntax for it using Lisp macros. This is one of the goals of this article and we'll return to it later.

Even though I like Lisp, I'm not sure why dynamic typing is the way to go —c.f. Dynamic Languages are Static Languages which mentions the unjust tyranny of unityped systems. Below are two reasons why people may dislike static types.

First: The de-facto typing rule do binary choice is usually:

That means valid programs such as `if True then 1 else "two"` are rejected; even though the resulting type will always be an integer there is no way to know that statically —the choice needs to be rewritten, evaluated at run time.

Indeed, in Haskell, we would write `if True then Left 1 else Right "two"` which has type `Either Int String`, and to use the resulting value means we need to pattern match or use the eliminator `(|||)` —from Haskell's `Control.Arrow`.

Second: Some statically typed languages have super weak type systems and ruin the rep for everyone else. For example, `C` is great and we all love it of-course, but it's a shame that we can only express the polymorphic identity function $id : \alpha.\alpha ß \alpha = x ß x$, by using the C-preprocessor —or dismiss the types by casting pointers around.

Maybe this video is helpful, maybe not: The Unreasonable Effectiveness of Dynamic Typing for Practical Programs

> (For the algebraist: Dynamic typing is like working in a monoid whose composition operation is partial and may abruptly crash; whereas static typing is working in a category whose composition is proudly typed.)

Overall I haven't presented a good defence for being dynamically checked, but you should ignore my blunder and consider trying Lisp yourself to see how awesome it is.

4 With its hierarchy of types, why isn't Lisp statically typed?

I haven't a clue. Here are two conjectures.

First: Code that manipulates code is difficult to type.

Is the type of `'(+ x 2)` a numeric code expression? Or just an arbitrary code expression? Am I allowed to "look inside" to inspect its structure or is it a black box? What about the nature of its constituents? If I'm allowed to look at them, can I ask if they're even defined?

What if `c` is a code element that introduces an identifier, say `it`. What is type of `c`? What if it doesn't introduce and thus avoids accidentally capturing identifiers? Are we allowed only one form or both? Which do we select and why?

I may be completely wrong, but below I mention a bunch of papers that suggest it's kind hard to type this stuff.

Second: The type theory just wasn't in place at the time Lisp was created.

Here's a probably wrong account of how it went down.

1913ish Bertrand Russel introduces a hierarchy of types to avoid barber trouble; e.g., $\texttt{Type}_i : \texttt{Type}_{i+1}$.

1920s A Polish guy & British guy think that's dumb and collapse the hierarchy.

1940s Alonzo Church says arrows are cool.

1958 With his awesome hairdo, John McCarthy gifts the world an elegant piece of art: Lisp (••)

- Lisp is currently the 2 oldest high-level language still in use after Fortran.
- Maxwell's equations get jealous.

Lisp introduces a bunch of zany ideas to CS:

- Introduced if-then-else "McCarthy's Conditional"; 1 class functions & recursion
- macros $\approx$ compiler plugins
- symbols $\approx$ raw names which needn't have values
- variables $\approx$ pointers
- code $\approx$ data; statements $\approx$ expressions
- `read, eval, load, compile, print` are all functions!

1959 My man JM thinks manual memory is lame —invents garbage collection!

- Later, 2001, he writes The Robot & The Baby.

1960s Simula says OOPs!

1970s Smalltalk popularises the phrase "oop". (B has a child named C.)

1970s Simple -calculus is a fashion model for sets and functions.

1970s Milner and friends demand *variables are for types too, not just terms!*

1970s Per Martin-Lof tells us it's okay to depend on one another; , types.

1982 A Lisp ummah is formed: "Common Lisp the Language"

- In order to be hip & modern, it's got class with CLOS.
- Other shenanigans: Scheme 1975, Elisp 1985, Racket 1995, Clojure 2007

1984 A script of sorcerous schemes lords lisp over mere mortals

1990s A committee makes a sexy camel named Haskell; Professor X's school make their own camel.

- Their kids get on steroids and fight to this day; Agda !! !! !! Coq.

2000s X's camel .<becomes .~(self .<aware>.)>. —the other camel [does| the same].

- In 2015, the cam ls married Lisp and Lux was born.
- In 2016, Haskell & Lisp get involved with Prolog; Shen is born.

2019: Coq is self-aware; Agda is playing catch-up.

A more informative historical account of Lisp & its universal reverence can be read at: How Lisp Became God's Own Programming Language. `https://imgs.xkcd.com/comics/lisp.jpg`

5 Lisp Actually Admits Static Typing!

Besides Common Lisp, "Typed Lisps" include an optional type system for Clojure —see also Why we're no longer using Core.typed— Typed Racket, and, more recently, Lux ≈ Haskell + ML + Lisp and Shen ≈ Haskell + Prolog + Lisp.

For example, Common Lisp admits strong static typing, in SBCL, as follows.

```
; Type declaration then definition.
(declaim (ftype (function (fixnum)) num-id))
(defun num-id (n) n)

(defun string-id (s) (declare (string s)) (num-id s))
; in: DEFUN STRING-ID
;     (NUM-ID S)
;
; caught WARNING:
;   Derived type of S is
;     (VALUES STRING &OPTIONAL),
;   conflicting with its asserted type
;     FIXNUM.
```

Such annotations mostly serve as compiler optimisation annotations and, unfortunately, Emacs Lisp silently ignores Common Lisp declarations such as ftype —which provides function type declarations. However, Emacs Lisp does provide a method of dispatch filtered by classes rather than by simple types. Interestingly, Lisp methods are more like Haskell typeclass constituents or C# extensible methods rather than like Java object methods —in that, *Lisp methods specialise on classes* whereas Java's approach is *classes have methods*.

Here's an example.

```
(defmethod doit ((n integer)) "I'm an integer!")
(defmethod doit ((s string)) "I'm a string!")
(defmethod doit ((type (eql :hero)) thing) "I'm a superhero!")

(doit 2)             ;;  ⇒  I'm an integer!
(doit "2")           ;;  ⇒  I'm a string!
(doit 'x)            ;;  ⇒  Error: No applicable method
(doit :hero 'bobert) ;;  ⇒  I'm a superhero!

;; C-h o cl-defmethod  ⇒  see extensible list of specialisers Elisp supports.
```

We can of-course make our own classes:

```
(defclass person  () ((name)))
(defmethod speak ((x person)) (format "My name is %s." (slot-value x 'name)))
(setq p (make-instance 'person))
(setf (slot-value p 'name) "bobert")
(speak p) ;;  ⇒  My name is bobert.

;; Inherits from 'person' and has accessor & constructor methods for a new slot
(defclass teacher (person) ((topic :accessor teacher-topic :initarg :studying)))

(defmethod speak ((x teacher))
  (format "My name is %s,and I study %s." (slot-value x 'name) (teacher-topic x)))

(setq ins (make-instance 'teacher :studying "mathematics"))
(setf (slot-value ins 'name) "Robert")
(speak ins) ;;  ⇒  My name is Robert, and I study mathematics.
```

Later in this article, we'll make something like the `declaim` above but have it be effectful at run-time. *Typing as Macros!*

> (If you happen to be interested in looking under the hood to see what compiler generated code looks like use `disassemble`. For example, declare `(defun go (x) (+ 1 x) 'bye)` then invoke `(disassemble 'go)` to see something like `varref x add1 discard constant bye return`.)

6 ELisp's Type Hierarchy

Each primitive type has a corresponding Lisp function that checks whether an object is a member of that type. Usually, these are the type name appended with `-p`, for multi-word names, and `p` for single word names. E.g., `string` type has the predicate `stringp`.

Type Descriptor Objects holding information about types.

This is a `record`; the `type-of` function returns the first slot of records.

This section is based GNU Emacs Lisp Reference Manual, §2.3 "Programming Types".

6.1 Number

Numbers, including fractional and non-fractional types.

`integer` `float` `number` `natnum` `zero` `plus` `minus` `odd` `even`

The relationships between these types are as follows:

```
(numberp x)  ≈  (or (integerp x) (floatp x))
(natnump x)  ≈  (and (integerp x) ( 0 x))
(zerop x)  ≈  (equal 0 x)
(plusp x)  ≈  (< 0 x)
(minusp x)  ≈  (> 0 x)
(evenp x)  ≈  (zerop (mod x 2))
(oddp x)  ≈  (not (oddp x))
```

- **Integer**: Numbers without fractional parts.

  There is no overflow checking.

```
(expt 2 60) ;;  ⇒  1,152,921,504,606,846,976
(expt 2 61) ;;  ⇒  -2,305,843,009,213,693,952
(expt 2 62) ;;  ⇒  0
```

  Numbers are written with an optional sign '+' or '-' at the beginning and an optional period at the end.

```
-1  ≈  -1.  1  ≈  +1  ≈  1.
```

  They may also take *inclusive* (and exclusive) ranges: The type list `(integer LOW HIGH)` represents all integers between `LOW` and `HIGH`, inclusive. Either bound may be a list of a single integer to specify an exclusive limit, or a `*` to specify no limit. The type `(integer * *)` is thus equivalent to `integer`. Likewise, lists beginning with `float`, `real`, or `number` represent numbers of that type falling in a particular range. (The Emacs Common Lisp Documentation)

```
(typep 4 '(integer 1 5)) ;;  ⇒  true since 1  4  5.
(typep 4 '(integer 1 3)) ;;  ⇒  nil  since 1  4  3.

(typep 12 'integer) ;;  ⇒  t
```

```
(typep 12 'number) ;; ⇒ t

(typep 23 'odd)  ;; ⇒ t

(typep 12 '(integer * 14)) ;; ⇒ t, since 12 ≤ 14, but no lower bound.
(typep 12 '(integer 0 *)) ;; ⇒ t; the '*' denotes a wild-card; anything.

(typep -1 '(not (integer 0 *))) ;; ⇒ t
(typep  1 '(not (integer 0 *))) ;; ⇒ nil

(typep 1 '(integer  1 2))   ;; ⇒ t, including lower bound
(typep 1 '(integer (1) 2))  ;; ⇒ nil, excluding lower bound

(typep 1.23 '(float 1.20 1.24)) ;; ⇒ t

;; Here's a slighly organised demonstration:

(typep 1.23 'number) ;; ⇒ t
(typep 123  'number) ;; ⇒ t
(typep 1.23 'real) ;; ⇒ t
(typep 123  'real) ;; ⇒ t

(typep 1.23 'integer) ;; ⇒ nil
(typep 123  'integer) ;; ⇒ t
(typep 1.23 'fixnum) ;; ⇒ nil
(typep 123  'fixnum) ;; ⇒ t

(typep 1.23 'float) ;; ⇒ t
(typep 123 'float) ;; ⇒ nil
(typep 123.0 'float) ;; ⇒ t
```

- **Floating-Point**: Numbers with fractional parts; expressible using scientific notation. For example, `15.0e+2` ≈ `1500.0` and `-1.0e+INF` for negative infinity.
- **Aliases:** The type symbol `real` is a synonym for `number`, `fixnum` is a synonym for `integer`, and `wholenum` is a synonym for `natnum`.
- The smallest and largest values *representable* in a Lisp integer are in the constants `most-negative-fixnum` and `most-postive-fixnum`

```
;; Relationship with infinities
(< -1e+INF most-negative-fixnum most-positive-fixnum 1e+INF) ;; ⇒ t
```

6.2 Character

Representation of letters, numbers, and control characters.

A character is just a small integers, up to 22 bits; e.g., character `A` is represented as the integer 65.

One writes the character 'A' as `?A`, which is identical to 65. Punctuations `()[]\;"|'‘#` must be escaped; e.g.,

`?\(` ≈ `40` `?\\` ≈ `92`

Whereas `?.` ≈ `46`.

```
(characterp ?f) ;; ⇒ t
(characterp t)  ;; ⇒ nil
```

Emacs specfic characters control-g `C-g`, backspace `C-h`, tab `C-i`, newline `C-j`, space, return, del, and escape are expressed by ? ?, ?, ?, ?, ?, ?, ?, ?.

Generally, control characters can be expressed as `?\^` ≈ `?\C-`, and meta characters by `?\M-`; e.g., `C-M-b` is expressed `?\M-\C-b` ≈ `?\C-\M-b`.

Finally, `?\S-` denotes shifted- characters. There are also `?\H-, ?\A-, ?\s-` to denote Hyper- Alt- or Super-modified keys; note that lower case 's' is for super whereas capital is for shift, and lower case with no dash is a space character.

6.3 Symbol

A multi-use object that refers to functions and variables, and more.

A symbol is an object with a name; different objects have different names.

```
(typep 'yes 'symbol) ;;  ⇒  true
(symbolp 'yes)       ;;  ⇒  true

(typep 12   'symbol) ;;  ⇒  false
(symbolp 12)         ;;  ⇒  false
```

`symbol` ≈ Is it a symbol?
`bound` ≈ Does it refer to anything?

```
(typep 'xyz 'bound) ;;  ⇒  nil

(setq xyz 123)
(typep 'xyz 'bound) ;;  ⇒  t
```

See this short docs page for more info on when a variable is void.

Names have a tremendously flexible syntax.

```
(setq +*/-_~!@$%^&:<>{}? 23)
(setq \+1            23) ;; Note +1  ≈  1, a number.
(setq \12            23)
(setq this\ is\ woah 23) ;; Escaping each space!
(+ this\ is\ woah 1)     ;;  ⇒  24
```

If the symbbol name starts with a colon ':', it's called a keyword symbol and automatically acts as a constant.

```
(typep :hello 'keyword) ;;  ⇒  t
```

Symbols generally act as names for variables and functions, however there are some names that have fixed values and any attempt to reset their values signals an error. Most notably, these include `t` for true or the top-most type, `nil` for false or the bottom-most type, and keywords. These three evaluate to themselves.

```
t      ;;  ⇒  t
nil    ;;  ⇒  nil
:hello ;;  ⇒  :hello

(setq t   12) ;;  ⇒  Error: Attempt to set a constant symbol
(setq nil 12) ;;  ⇒  Error: Attempt to set a constant symbol
(setq :x  12) ;;  ⇒  Error: Attempt to set a constant symbol

;; :x  'x
(set 'x 12) ;;  ⇒  12
x           ;;  ⇒  12
```

```
;; They're self-evaluating
(equal t   't)   ;;  ⇒  t
(equal nil 'nil) ;;  ⇒  t
(equal :x  ':x)  ;;  ⇒  t

(equal :x 'x)  ;;  ⇒  nil
```

In particular, `:x  'x`!

6.4 Sequence

The interface for ordered collections; e.g., the `(elt sequence index)` function can be applied to any sequence to extract an element at the given index.

`sequence  seq`

The latter is an extensible variant of the former —for when we declare our own sequential data types.

```
(typep '(1 2 3) 'sequence) ;;  ⇒  t
```

There are two immediate subtypes: `array` and `cons`, the latter has `list` as a subtype.

```
(typep  [1 2 3] 'array)       ;;  ⇒  t
(typep '(1 2 3) 'cons)        ;;  ⇒  t
(typep '(1 "2" 'three) 'list) ;;  ⇒  t
```

Array Arrays include strings and vectors.

Vector One-dimensional arrays.

Char-Table One-dimensional sparse arrays indexed by characters.

Bool-Vector One-dimensional arrays of `t` or `nil`.

Hash Table Super-fast lookup tables.

```
(typep "hi" 'string) ;;  ⇒  true
(typep 'hi  'string) ;;  ⇒  false
```

Cons cell type Cons cells and lists, which are chains of cons cells.

These are objects consisting of two Lisp objects, called `car` and `cdr`. That is they are pairs of Lisp objects.

Notice that when there is no '.', then a list is just a nested cons chain ending in 'nil'. Note that `'(x` $_0$ `. x` $_1$ `. x)` is meaningless.

Cons cells are central to Lisp and so objects which are not a cons cell are called *atoms*.

```
;; An atom is not a cons.
(typep 123 'atom) ;;  ⇒  t
(typep 'ni 'atom) ;;  ⇒  t
```

Computationally:

```
    (atom x)
≈   (typep x 'atom)
≈   (not (consp x))
≈   (not (typep x 'cons))
≈   (typep x '(not cons))
```

Interestingly, one writes `atom`, not `atomp`.

6.5 Function

Piece of executable code.

A non-compiled function in Lisp is a lambda expression: A list whose first element is the symbol `lambda`.

```
(consp     (lambda (x) x))        ;;  ⇒  true
(functionp (lambda (x) x))        ;;  ⇒  true

(functionp (lambda is the first)) ;;  ⇒  true
(typep (lambda stuff) 'function)  ;;  ⇒  true
```

It may help to know that a `defun` just produces an alias for a function:

```
  (defun name (args) "docs" body)
≈ (defalias (quote name) (function (lambda (args) docs body)))
```

Here's some more examples.

```
(typep #'+   'function) ;;  ⇒  true
(typep 'nice 'function) ;;  ⇒  false

(defun it (x) (format "%s" (+1 x)))
(typep #'it   'function) ;;  ⇒  true
(functionp #'it)         ;;  ⇒  true
```

6.6 Macro

A method of expanding an expression into another expression.

Like functions, any list that begins with `macro`, and whose `cdr` is a function, is considered a macro as long as Emacs Lisp is concerned.

```
(macrop '(macro (lambda (x) x))) ;;  ⇒  true
```

Since `defmacro` produces an alias, as follows,

```
  (defmacro name (args) "docs" body)
≈ (defalias (quote name) (cons (quote macro) (function (lambda (args) docs body))))
```

You may be concerned that `(macrop x)  (equal 'macro (car x))`, and so if a user gives you a macro you might think its a cons cell of data. Fortunately this is not the case:

```
(defmacro no-op () )

(macrop #'no-op)    ;;  ⇒  true
(consp  #'no-op)    ;;  ⇒  false; whence it's also not a list.
(functionp #'no-op) ;;  ⇒  false

(typep #'no-op '
       (satisfies (lambda (x) (and (listp x) (equal 'macro (car x))))) ;;  ⇒  false
```

Why not? Well, you could think of a macro as a 'record' whose label is `macro` and its only element is the associated function.

6.7 Record

Compound objects with programmer-defined types.

They are the underlying representation of `defstruct` and `defclass` instances.

For example:

```
(defstruct person
  name age)
```

The `type-of` operator yields the `car` of instances of such declartions.

$$\texttt{(record}\ \ \tau\ \ \texttt{e}_0\ \ \ldots\ \ \texttt{e}_n\ \texttt{)}\ \ \approx\ \ \texttt{\#s(}\ \tau\ \ \texttt{e}_0\ \ \ldots\ \ \texttt{e}_n\ \texttt{)}$$

```
(setq bobert (make-person :name "bobby" :age 'too-much))
(type-of bobert) ;; ⇒ person
```

Componenets may be indexed with `aref`.

```
(aref bobert 1)      ;; ⇒ bobby
(person-name bobert) ;; ⇒ bobby
```

A record is considered a constant for evaulation: Evaluating it yields itself.

```
(type-of #s(person "mark" twelve)) ;; ⇒ person
(recordp #s(nice))                 ;; ⇒ t
```

7 Typing via Macros & Advice

Checking the type of inputs is tedious and so I guessed it could be done using macros and advice. Looking at Typed Racket for inspiration, the following fictitious syntax would add advice to `f` that checks the optional arguments $\texttt{x}_i$ have type σ_i and the mandatory positional arguments have type τ_i according to position, and the result of the computation is of type τ. To the best of my knowledge, no one had done this for Emacs Lisp —I don't know why.

$$\texttt{(declare-type 'f ((:x}_1\ \ \sigma_1\ \texttt{)}\ \boxed{\ldots}\ \texttt{(:x}\ \ \sigma\ \texttt{))}\ \texttt{(}\ \tau_1\ \ \boxed{\ldots}\ \ \tau_n\ \ \tau\ \texttt{))}$$

To modify a variable, or function, we may simply redefine it; but a much more elegant and powerful approach is to "advise" the current entity with some new behaviour. In our case of interest, we will *advise functions to check their arguments before executing their bodies.*

Below is my attempt: `declare-type`. Before you get scared or think it's horrendous, be charitable and note that about a third of the following is documentation and a third is local declarations.

```
(cl-defmacro declare-type (f key-types &rest types)
  "Attach the given list of types to the function ‘f’
   by advising the function to check its arguments' types
   are equal to the list of given types.

   We name the advice ‘f-typing-advice’ so that further
   invocations to this macro overwrite the same advice function
   rather than introducing additional, unintended, constraints.

   Using type specifiers we accommodate for unions of types
   and subtypes, etc    .

   ‘key-types’ should be of the shape (:x0 t0 ⋯ :xn tn );
    when there are no optional types, use symbol “:”.
```

```
  E.g., (declare-type my-func (:z string :w integer) integer symbol string)
"

;; Basic coherency checks. When there aren't optional types, key-types is the ":" symbol.
(should (and (listp types) (or (listp key-types) (symbolp key-types))))

(letf* ((pairify (lambda (xs) (loop for i in xs by #'cddr          ;; Turn a list of flattenned pairs
                                    for j in (cdr xs) by #'cddr    ;; into a list of explicit pairs.
                                    collect (cons i j))))          ;; MA: No Lisp method for this!?
        (result-type  (car (-take-last 1 types)))
        (types        (-drop-last 1 types))
        (num-of-types (length types))
        (key-types-og (unless (symbolp key-types) key-types))
        (key-types    (funcall pairify key-types-og))
        (advice-name  (intern (format "%s-typing-advice" f)))
        (notify-user  (format "%s now typed %s → %s → %s."
                              `,f key-types-og types result-type)))

    `(progn
       (defun ,advice-name (orig-fun &rest args)

         ;; Split into positional and key args; optionals not yet considered.
         (letf* ((all-args
                    (-split-at
                      (or (--find-index (not (s-blank? (s-shared-start ":" (format "%s" it)))) args)
                       args)) ;; The "or" is for when there are no keywords provided.
                 (pos-args  (car all-args))
                 (key-args  (funcall ,pairify (cadr all-args)))
                 (fun-result nil)
                 ((symbol-function 'shucks)
                    (lambda (e τ  e g)
                      (unless (typep g e τ )
                        (error "%s: Type mismatch! Expected %s %s  Given %s %s."
                               (function ,f) e τ  e (type-of g) (prin1-to-string g))))))

         ;; Check the types of positional arguments.
         (unless (equal ,num-of-types (length pos-args))
           (error "%s: Insufficient number of arguments; given %s, %s, but %s are needed."
                  (function ,f) (length pos-args) pos-args ,num-of-types))
         (loop for (ar ty pos) in (-zip pos-args (quote ,types) (number-sequence 0 ,num-of-types))
               do (shucks ty (format "for argument %s" pos) ar))

         ;; Check the types of *present* keys.
         (loop for (k . v) in key-args
               do (shucks (cdr (assoc k (quote ,key-types))) k v))

         ;; Actually execute the orginal function on the provided arguments.
         (setq fun-result (apply orig-fun args))
         (shucks (quote ,result-type) "for the result type (!)" fun-result)

         ;; Return-value should be given to caller.
         fun-result))
```

```
    ;; Register the typing advice and notify user of what was added.
    (advice-add (function ,f) :around (function ,advice-name))
    ,notify-user )))
```

There are some notable shortcomings: Lack of support for type variables and, for now, no support for optional arguments. Nonetheless, I like it —of course. (Using variable watchers we could likely add support for type variables as well as function-types.)

We accidentally forgot to consider an argument.

```
(declare-type f₁ (:z string :w list) integer symbol string)
;; ⇒ f₁ now typed (:z string :w integer) → (integer symbol) → string.

(cl-defun f₁ (x y &key z w) (format "%s" x))
;; ⇒ f₁ now defined

(f₁ 'x) ;; ⇒ f₁ : Insufficient number of arguments; given 2, (x), but 3 are needed.
```

The type declaration said we needed 3 arguments, but we did not consider one of them.

We accidentally returned the wrong value.

```
(declare-type f (:z string :w list) integer symbol string)
(cl-defun f (x y &key z w) x)

(f 144 'two)
;; ⇒ f: Type mismatch! Expected string for the result type (!) Given integer 144.
```

We accidentally forgot to supply an argument.

```
(declare-type f (:z string :w list) integer symbol string)
(cl-defun f (x y &key z w) (format "%s" x))

(f 144)
;; ⇒ f: Insufficient number of arguments; given 1, (144), but 2 are needed.
```

A positional argument is supplied of the wrong type.

```
(f 'one "two")
;; ⇒ f: Type mismatch! Expected integer for argument 0 Given symbol one.

(f 144 "two")
;; ⇒ f: Type mismatch! Expected symbol for argument 1 Given string "two".
```

Notice: When multiple positional arguments have type-errors, the errors are reported one at a time.

A keyword argument is supplied of the wrong type.

```
(f 1 'two :z 'no₀ :w 'no₁ )
;; ⇒ f: Type mismatch! Expected string :z Given symbol no₀ .

(f 1 'two :z "ok" :w 'no₁ )
;; ⇒ f: Type mismatch! Expected string :w Given symbol no₁ .

(f 1 'two :z "ok" :w 23)
;; ⇒ f: Type mismatch! Expected string :w Given integer 23.

(f 1 'two :z "ok" :w '(a b 1 2)) ;; ⇒ okay; no type-error.
```

We have no optional arguments.

```
(declare-type f : integer symbol string)
(cl-defun f (x y &key z w) (format "%s" x))

(f 144 'two :z "bye")
;;  ⇒   f: Type mismatch! Expected nil :z  Given string "bye".
;; ( We shouldn't have any keyword :z according to the type declaration! )

(f 144 'two) ;;  ⇒  "144"
```

We can incorporate type specfiers such as unions!

```
(declare-type f : (or integer string) string)
(cl-defun f (x) (format "%s" x))

(f 144)     ;;  ⇒  "144"
(f "neato") ;;  ⇒  "neato"

(f 'shaka-when-the-walls-fell)
;;  ⇒  f: Type mismatch! Expected (or integer string) for argument 0
;;       Given symbol shaka-when-the-walls-fell.
```

No positional arguments but a complex optional argument!

```
(declare-type f (:z (satisfies (lambda (it) (and (integerp it) (= 0 (mod it 5))))))
                character)
(cl-defun f (&key z) ?A)

(f 'hi)     ;;  ⇒   Keyword argument 144 not one of (:z)
(f)         ;;  ⇒  65; i.e., the character 'A'
(f :z 6)
;;  ⇒   f: Type mismatch!
;;    Expected (satisfies (lambda (it) (and (integerp it) (= 0 (mod it 5))))) :z
;;     Given integer 6.

(f :z 10) ;;  ⇒  65; i.e., the expected output since 10 mod 5  ≈  0 & so 10 is valid input.
```

Preconditions! The previous example had a complex type on a keyword, but that was essentially a precondition; we can do the same on positional arguments.

```
(declare-type f : (satisfies (lambda (it) (= it 5)))
                  integer)
(cl-defun f (n) n)
;; The identity on 5 function; and undefined otherwise.

(f 4)
;;  ⇒  f: Type mismatch! Expected (satisfies (lambda (it) (= it 5))) for argument 0
;;       Given integer 4.

(f 5) ;;  ⇒  5
```

Postconditions! Given an integer greater than 5, we present an integer greater than 2; i.e., this is a constructive proof that $nn > 5 \Rightarrow n > 2$.

```
(declare-type f : (satisfies (lambda (in)  (> in 5)))
                  (satisfies (lambda (out) (> out 2))))
(cl-defun f (n) n)
```

```
;; The identity on 5 function; and undefined otherwise.

(f 4)
;;  ⇒   f: Type mismatch! Expected (satisfies (lambda (in) (> in 5))) for argument 0
;;         Given integer 4.

(f 72) ;;  ⇒  72; since indeed 72 > 5 for the input, and clearly 72 > 2 for the output.
```

As it currently stands we cannot make any explicit references between the inputs and the output, but that's an easy fix: Simply add a local function `old` to the `declare-type` macro which is intentionally exposed so that it can be used in the type declarations to refer to the 'old', or initial, values provided to the function. Additionally, one could also add keyword arguments `:requires` and `:ensures` for a more sophisticated pre- and post-condition framework. Something along these lines is implemented for Common Lisp.

Here's a fun exercise: Recast the Liquid Haskell examples in Lisp using this `declare-type` form.

8 Closing

> *I have heard more than one LISP advocate state such subjective comments as, "LISP is the most powerful and elegant programming language in the world" and expect such comments to be taken as objective truth. I have never heard a Java, C++, C, Perl, or Python advocate make the same claim about their own language of choice.*
>
> —A guy on slashdot

I learned a lot of stuff, hope you did too ^_^

9 References

Neato web articles:

- What to know before debating type systems
  - Debunks a number of fallacies such as "dynamic typing provides no way to find bugs" and "static types need type declarations".
- Dynamic Languages Strike Back
  - Everything you might wanna know about dynamically checked languages.
- The Association of Lisp Users
  - Abundant resource relating to Lisp.
- Untyped Programs Don't Exist
  - It's not a matter of typing but of pragmatics.
- What is Gradual Typing:
  - Discusses how static and dynamic typing can be used together hamroniously.
- CLiki — The Common Lisp Wiki
  - Contains resources for learning about and using the programming language Common Lisp.
  - The humour section is delightful.
- Strong Static Type Checking for Functional Common Lisp

  - PhD thesis regarding strong static type checking in an applicative subset of CL.

- Beating the Averages
  - Paul Graham discusses "the most powerful language available" —Lisp.
  - Other articles he's written about Lisp can be found here.

- The Bipolar Lisp Programmer
  - "Lisp is, like life, what you make of it." Lisps attract a certain kind of personality.

- A bunch of papers on polymorphic (modal) type systems for Lisp-like multi-staged languages: This is generic, this is ML + Scheme, this for compile-time typing, and this one "allows the programmer to declaratively express the types of heterogeneous sequences in a way which is natural in the Common Lisp language."

- Type Systems as Macros
  - After defining `declare-type` I thought the slogan "types by macros" sounded nifty; Googling it led me to this paper where the Racket is endowed with types.

    Lisp is great lol.

- How Lisp Became God's Own Programming Language
  - History and venerance of Lisp.

- Common Lisp HyperSpec – Type Specifiers