The History and Rationale of Typed Lisps

Casper Rutz
March 2, 2020, revised April 8, 2021

For good or for ill, Lisp has become legendary since its release in 1958. Lisp pioneered many features that have become nearly ubiquitous in programming, including macros, self-hosting compilers, garbage collection, dynamic typing, trees, conditionals, higher-order functions, REPLs, and recursion, and is also one of the most extensible and flexible programming language created. The creator of Clojure, Rich Hickey, noted that Lisp features are often either emulated or pillaged by other languages, but the core power of Lisp has never been duplicated [15]. This is true even within Lisps' own language family: the original Lisp can still be found as the core functionality of every modern Lisp.

In particular, dynamic typing has been a Lisp staple since its inception. Lisp code relies on the ability to create lists of very disparate types, a feature that dynamic typing helps enable. These multi-typed constructs are such a core language feature that many Lispers balk at the idea of trying to correctly type their code in a restrictive typing system and also refuse the idea of removing the ability to create these structures. Languages such as Haskell, with its extraordinary emphasis on its ML type system, seem alien and strange from a Lisp perspective. Because of this, the creation of typed Lisps is of particular interest to many Lispers, myself included, who sometimes view the effort with a certain level of skepticism.

In recent years typed alternatives to dynamically typed languages have appeared, such as TypeScript instead of JavaScript. Typed Racket is possibly the foremost example of this phenomenon in Lisp; Typed Racket introduces an optional Hindley-Milner typing system, cooperates with untyped Racket, and is very well implemented and documented. Having heard positive reviews of Typed Racket, I decided to research the rationale behind the addition of this typing to Lisp. What I was not expecting is to need to start back at the original 1958 version of Lisp.

The Effects of Dynamic Typing

Most contemporary programmers can list the cons of dynamic typing, however popular certain dynamically typed languages may be; however, in the years after Lisp was first released, the problems related to dynamic typing were just beginning to be known. Similar to modern dynamically typed languages, early Lisp often would not error if a function or program was

assigned the wrong type, but would instead just give nonsensical output. If the program did error, the errors were often unhelpful: a type error could crop up under `UNBOUND VARIABLE USED` if the compiler decided that the type mismatch was instead incorrect variable shadowing, or `FUNCTION OBJECT HAS NO DEFINITION` if it decided that the type mismatch was incorrect function polymorphism [2, 3]. Because there were no clear errors for type mismatches in early Lisp, it was often extremely difficult for early programmers to locate the exact point in their (often poorly-documented) code where the error first occurred.

Within a few years of Lisp's release, alternatives to its dynamic type system began to be considered. Fortran had debuted one year before Lisp with a static typing system replete with type declarations. However, there are multiple reasons the Lisp community realized a Fortran-like type system would not work in Lisp. One reason, mentioned above, is that Lisp code relies on flexible data structures a statically typed system would make impossible [4]; the second reason is that unlike Fortran, Lisp has almost no syntax. As a consequence of this, a normal amount of type declarations in Fortran can easily fade into the background, whereas the same number of type declarations in Lisp is much more obvious, as shown in Fig 1.

```
FORTRAN
  S = FLOAT (A + B + C) / 2.0
  AREA = SQRT( S * (S - FLOAT(A)) * (S - FLOAT(B)) *
              (S - FLOAT(C)))

LISP
(SET S (/ (FLOAT (+ A B C))
          2.0))

(SQRT (* S
         (- S (FLOAT A))
         (- S (FLOAT B))
         (- S (FLOAT C))))
```

Fig 1. Fortran and Lisp code, with type declarations highlighted in red. Syntax highlighting added for readability.

The problem of the sheer number of types obscuring Lisp code becomes more apparent when one considers more complex data structures. Due to Lisp's inherent polymorphism, constructing even a simple data structure like a list of integers would necessitate every third word being a type declaration, as shown in Fig 2. It is not hard to imagine that in such a system, the type for a complex data structure filled with other complex data structures of varying types

```
(CONS INT I (CONS INT J (CONS INT K (LIST INT NULL))))

(CONS I (CONS J (CONS K (LIST NULL))))
```

Fig 2. Integer list creation in Lisp using static vs dynamic typing.

may be more characters than the data structure itself. This is an unacceptable level of baggage to a language as syntactically simple as Lisp: because the structure of a Lisp program is also the logic of a Lisp program, the more levels of nesting and syntactic baggage a Lisp has, the more the logic of a program is obscured. A statically typed Lisp would certainly be more unreadable and harder to parse, even with the helpfulness of static type declarations.

It became clear to early Lisp programmers that a new type system needed to be invented in order to correctly type Lisp without essentially ruining the language. This dilemma is summed up succinctly by Henry Milner in *A Theory of Type Polymorphism in Programming* as an explanation for his research into alternate ways to type Lisp:

> 'The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures which work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists). Such flexibility is almost essential in this style of programming; unfortunately one often pays a price for it in the time taken to find rather inscrutable bugs…On the other hand a type discipline such as that of ALGOL 68…which precludes the flexibility mentioned above, also precludes the programming style which we are talking about.' [4].

The Introduction of Type Inference

Type inference is the process by which a compiler infers the type of an expression. Occasionally the compiler is helped along by type annotations provided by the programmer, as there are circumstances where the compiler cannot infer a type at all (such as in polymorphic recursion) or will infer a type that is less desirable (e.g., less space or time efficient) than another. Generally, the compiler can infer types by using type annotations to track the use of types through a program, or by "understanding" values enough to assign types to them. These options can also be employed in tandem, as they are in Ocaml, a modern dialect of ML [16].

On a very high level, Ocaml's type inference algorithm works by using context clues to determine the types in the first expression found in the program, and then uses those type definitions to determine the types of later expressions. For example, in the expression `let y = x + 1` the compiler can infer that because `1` is an integer, `+` must refer to integer addition, and therefore `x` and `y` must be integers as well. As `x` and `y` are passed around the program and interact with more values, the compiler gathers more and more information on the type constraints certain variables have. Eventually, the compiler can use enough of these type constraints to logically deduce what every—or almost every—type in a program must be [16].

The research being done into type inference in the 1960's and 1970's led to several versions of typed Lisp [12, 13], though none proved popular enough to become widely used. Instead, type inference ultimately culminated with the research of J. Roger Hindley and Robin

Milner. Hindley's 1969 paper introduced a type inference system for lambda calculus with parametric polymorphism, which perfectly describes Lisp [1]. Milner built on top of these concepts, and created a new type system using type inference now called the Hindley-Milner type system (abbreviated H-M henceforth) [4]. Milner introduced his typing system in *A Theory of Type Polymorphism in Programming*, but chose to concentrate the effects of H-M into a new language, ML, instead of injecting them back into Lisp. ML became a popular language in functional programming, and it inspired other languages such as Ocaml and Haskell [8]. ML also has been cited as an inspiration for Clojure, a modern Lisp [6]; however, Clojure is dynamically typed. It would be decades before Haskell's popularity spawned a useable, popular H-M typed Lisp.

Typed Racket

Typed Racket is the foremost example of typed Lisp and it also has a large influence over other typed Lisps, such as efforts to create a Typed Clojure. Most importantly, Typed Racket was implemented very carefully: it cleverly uses H-M while not limiting Lisp's flexibility, is well supported and virtually bug-free, and also recognizes that it is likely to be used by developers who are not used to typed Lisps. There is a lengthy guide on exactly how to annotate Typed Racket code correctly, and the guide lays out a method to allow users to incrementally add type annotations to traditional Racket code [9]. When compared to Haskell's reference documents, it is clear that Typed Racket kept 'not scaring off Lispers' in mind.

Despite Typed Racket's gentleness, it has a very robust typing system and also provides its users with powerful typing tools. Like many ML dialects, Typed Racket employs many various types with small differences between them. Fig 3 shows that there are seven broad categories of Number types, which can be further refined into seven different Integer types. When inferring types, Typed Racket will infer the least inclusive type, hence why in

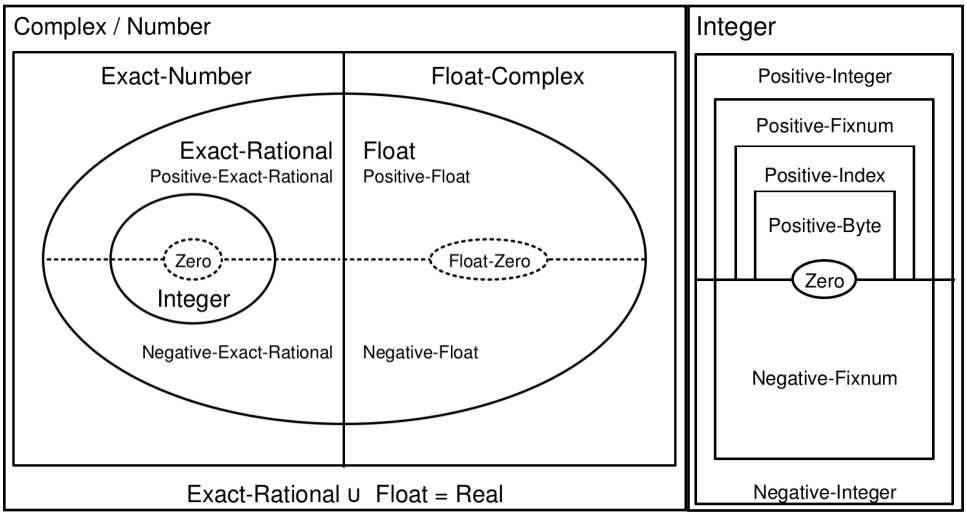

Fig 3. Examples of the Number and Integer subtypes in Typed Racket [10]

Fig 4, `1` is of type One, whereas `2` and `3` are of type Positive-Byte. When dealing of lists of multiple types, Typed Racket will create a new type that is the union of every type present in the list. Fig 5 shows that a list of false and null has type `(Listof (U False Null))`, or *type List of the union of type False and type Null*. This is how Typed Racket manages to keep Lisp's flexibility.

```
> (list 1 2 3)
- : (Listof Positive-Byte) [more precisely: (List One Positive-Byte Positive-Byte)]
'(1 2 3)
```

Fig 4. Examples of type inference in Typed Racket

```
> (list false null)
- : (Listof (U False Null)) [more precisely: (List False Null)]
'(#f ())
```

Fig 5. Examples of type inference in Typed Racket using type unions

Typed Racket also allows users to define their own types, as seen in Fig 6. User-defined types can be used to import functions from untyped Racket, refine existing types into new types such as in Fig 7, or in structs. To create a struct, the user first gives a name to the struct and then defines which atoms and associated types are included in the struct, whose type then becomes a union of each type listed. It is apparent that a struct is really a way to easily type annotate a complex data structure, similarly to how user-defined types are used in Haskell. As Haskell is descended from ML and Racket is not, we can deduce that this shared usage is due to the influence of the H-M typing system alone.

```
(struct turn
  ([board : (Listof r:cell)]
   [player : r:Player]
   [message: String]
   [first? : Boolean]
   [captured : (U r:cell False)]
   [src-coords : b:Position]
   [valid? : Boolean])
  #:transparent)
```

```
(define-type None False)

(define none: None #f)

(define-type Player (U String None))

(define-type Role String)
```

Fig 6. *Examples of user defined types in Typed Racket, including structs [17]*

```
(define-type Column
  (Refine [n : Integer]
          (and (> 8 n)
               (<= 0 n))))

(define-type Row
  (Refine [n : Integer]
          (and (> 4 n)
               (<= 0 n))))
```

Fig 7. *Examples of type refinements in Typed Racket [17]*

Do Types Make Racket Easier To Read?

I have remained objective up to this point, but ultimately, the question of whether types make Lisp easier to read is—and always will be—subjective. My opinion does not have much weight in this decades-long discussion, but I will offer it regardless.

I analyzed and altered a Racket program, *Graveyard*, to decide if I thought typed Racket lived up to the hype. *Graveyard* is a Banqi game written by Thea Leake [17]. Thea originally wrote her game in untyped Racket, and decided to convert it to typed Racket after the majority of the game was already written. As most Racket projects are either typed or untyped but not both, this provided me with a rare opportunity to get a side-by-side comparison of the same program in both typed and untyped Racket.

I read through the untyped code to gain a general understanding of the structure and flow of the program, and then I chose functions at random to see how difficult it was to determine the purpose of each function. It was slightly easier to track logic chains throughout the code base using typed Racket because I could immediately see type differences between complex data structures. For example, the Cannon game piece has a different type than the other game pieces. By following that type through the codebase, I realized that the Cannon has its own set of movement functions because it moves in a different manner than the other pieces do. I'd completely missed that when examining the untyped code.

Because this first experiment went so well, I decided to alter Thea's code (with her permission) as a project for the graduate level Internetworking Protocols course at PSU. My project was to design and add a network protocol into the game so that a game could be played between two remote players [18]. I spent a week analyzing her code and planning my approach

before I tried anything myself, as Thea's code relied heavily on concurrent programming which I was unfamiliar with. I then spent another week putting my plan into action.

Far from harming readability, Thea's types proved very helpful during this whole process. I was able to instantly figure out what type something was without relying on context clues, which made the process of understanding the code and figuring out how to successfully add in changes both faster and easier. Debugging was also faster, as I was able to trace lost variables more easily through the program. I did not type my code however, so I cannot speak to the ease of that specifically.

In the end, my highly subjective series of experiments proved that typed Racket decreases the time it takes me to understand and debug unfamiliar code. I also appreciated typed Racket's quality of life features, such as the smooth interaction between typed and untyped code (as I did not type my code) and their stellar reference documents.

Conclusion

I went into this research having already programmed in both modern Lisps and Haskell and having strongly ended up favoring one over the other. In many ways Haskell and Lisp seem to be at the opposite sides of the functional programming spectrum, and I couldn't get over what I saw as an inherent logical divide between the languages. I never expected that frustrations with Lisp partly drove the invention of the Hindley-Milner type system.

This project gave me an appreciation of the Hindley-Milner type system that I did not have before. I can now say that I enjoy working in typed Racket, and will probably continue to use it in the future. I also decided to give languages in the ML family another chance, and spent ten weeks deep-diving into the inner workings of Curry, a functional logic programming language based on Haskell. My work in Racket and Curry has proven to me that the Hindley-Milner typing system is more nuanced than I originally thought, and is well-suited to providing types for flexible languages.

REFERENCES

[1] R. Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society* 146 (December 1969), 29-60. DOI:https://doi.org/10.2307/1995158

[2] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart and Michael I. Levin. 1985. *LISP 1.5 Programmer's Manual* (2nd. ed.). The M.I.T. Press, Cambridge, MA. DOI:https://doi.org/10.7551/mitpress/5619.001.0001

[3] J. McCarthy, R. Brayton, D. Edwards, P. Fox, L. Hodes, D. Luckham, K. Maling, D. Park and S. Russell. 1960. *Lisp I Programmer's Manual*. Computation Center and Research Laboratory of Electronics. Massachusetts Institute of Technology, Cambridge, MA.

[4] Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences* 17 (1978), 348-375. DOI:https://doi.org/10.1016/0022-0000(78)90014-4

[6] Rich Hickey. Clojure Bookshelf. *Listmania!*. Archived from the original October 3, 2017. Retrieved March 9, 2020 from https://web.archive.org/web/20171003001051/https://www.amazon.com/gp/richpub/listmania/fullview/R3LG3ZBZS4GCTH

[8] Mark P. Jones. 1994. *ML typing, explicit polymorphism and qualified types*. In TACS '94: Conference on theoretical aspects of computer software, April 1994, Sendai, Japan. Springer-Verlag Lecture Notes in Computer Science, 789.

[9] Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. The Typed Racket Guide (v.7.6). Retrieved March 9, 2020 from https://docs.racket-lang.org/ts-guide/index.html

[10] Sam Tobin-Hochstadt, Vincent St-Amour, Eric Dobson, and Asumu Takikawa. The Typed Racket Reference (v.7.6). Retrieved March 9, 2020 from https://docs.racket-lang.org/ts-reference/index.html

[12] Robert S. Cartwright. 1977. A practical formal semantic definition and verification system for typed lisp. Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Order Number: AAI7725755.

[13] Robert Cartwright. 1976. User-Defined Data Types as an Aid to Verifying LISP Programs. *International Colloquium on Automata, Languages and Programming* (1976), 228-256.

[14] Wikipedia. 2020. Fortran. Retrieved March 9, 2020 from https://en.wikipedia.org/wiki/Fortran

[15] Rich Hickey. 2019. Rationale. Retrieved March 9, 2020 from https://www.clojure.org/about/rationale

[16] 2016. Type Inference. Cornell University CS3110 lecture notes. Retrieved March 9, 2020 from https://www.cs.cornell.edu/courses/cs3110/2016fa/l/17-inference/notes.html

[17] Thea Leake. 2020. Graveyard: Game of Banqi. Retrieved March 9, 2020 from https://github.com/thea-leake/graveyard

[18] Casper Rutz. 2020. Graveyard: Game of Banqi. Retrieved April 8, 2021 from https://github.com/catspook/graveyard