System Q: Logical Soundness with Logically Unsound Types

Aaron Stump and Victor Taelin

August 8, 2025

1 What is a type theory?

A type theory is a statically typed programming language that can be understood as a logic. Programs are viewed as proofs, and the types of programs are viewed as the formulas they prove. This famous idea is called the Curry-Howard isomorphism. A very simple example is the program $\lambda\, x\,.\,\lambda\, y\,.\,x$, which takes in input x and then input y, and returns x. This program can be given the type $A \to B \to A$, for any types A and B. That type expresses that the program takes in an input of type A and then one of type B, and returns a result of type A. That indeed correctly describes the behavior of $\lambda\, x\,.\,\lambda\, y\,.\,x$. But it is also a valid logical formula, where the → operator is implication: A implies B implies A for the trivial reason that A implies A, and adding an extra assumption that B is true does not change that fact. To go beyond just propositional logic, type theories use more expressive types than just implications. We will see examples below.

To be interpreted as a logic, it is not enough to have a way to view the types of a programming language as formulas. We must ensure that it is not possible to prove false formulas. So the language must be logically sound. In type theory, an essential part of ensuring logical soundness is to guarantee that all programs terminate. The reason for this is that an infinite loop can be viewed, in most programming languages, as having any type one wants. So you can prove the formula False by writing a diverging program.

Much theoretical effort has been expended on techniques for proving logical soundness of type theories, by showing that all programs are guaranteed to terminate.

2 A new approach to logical soundness

There are two current traditions for devising type theories, that should be mentioned for comparison:

1. **Church-style** type theory builds up a notion of typed terms (programs), where the types are inherent to those terms. By a difficult argument, one shows that all well-typed programs terminate. So the type system is enforcing termination, in addition to other properties usually enforced by static typing. From termination, it is then easy to argue that the system is logically sound. This is because it is relatively easy to show that values, which are the final results of computation, cannot have type False.
2. **Curry-style** type theory starts with a notion of type-free program, and then adds types to describe properties of the behavior of programs. For example, the identity function can be described as having type $X \to X$ for any type X, as it is guaranteed to take an input of type X and return an output of type X (namely, the input it was given). A difficult argument is still required to show that typing enforces termination. But the language design is made quite a bit easier by not having types be inherent parts of programs. This is because in reasoning about programs, one does not then have to reason about types inside them. Programs are type-free, and typing comes second. In fact, the slogan I propose for this style of type theory is "Computation First" (because we first explain what type-free programs

$$\begin{array}{llll} \textit{Variables} & x, y, z, \ldots & & \\ \textit{Labels} & l & & \\ \textit{Terms} & s, r, t & ::= & x \mid l \mid \lambda\, x\,.\, s \mid t\ t' \mid \langle\rangle \mid [l, n, r] \mid r\ \bullet\ t \\ & & & \{l_1 \mapsto t_1\ ;\ \ldots\ ;\ l_k \mapsto t_k\} \end{array}$$

Figure 1: The syntax of Q's untyped language. In λ-abstractions, the variable x is allowed to occur at most once in the body s

are and how they execute, and only afterwards use types to describe their properties). Krivine puts it simply: "types can be thought of as properties of λ-terms" [1, page 43].

The philosophy we adopt here can be viewed as a strengthened form of Curry-style type theory, with the modified slogan: "Terminating Computation First". The idea is similar to Curry-style type theory, where one first defines type-free programs, and how they compute. But differently, these programs are designed so that they are guaranteed to terminate, without reference to any notion of typing. Just the structure of the programs and the rules for how they execute are sufficient to establish that all programs terminate. Giving a detailed proof of that fact is still not trivial, but expected to be much simpler than the approaches based on typing. And then one has a lot of freedom to design a type system on top of the terminating type-free language. The only requirement is that the language should have the usual type-safety property that one expects of any statically typed programming language. This is vastly easier to achieve than crafting a type system that enforces termination. It also opens the door to making use of exotic typing features that might not enforce termination. Since termination is enforced already from the structure of untyped terms, we are free to adopt such types without losing logical soundness.

This paper gives a particular example of this approach, in the form of a type theory called Q. The untyped substrate of the theory is based on affine lambda calculus plus an affine-compatible form of W-structures (the untyped components of W-types). There are two other design goals we pursue in this particular design: minimality and very expressive typing. We aim to have a core language with a small number of primitive operations, and similarly for the type system. Furthermore, to demonstrate the power of the approach, we include the principle known as "Type : Type", which renders the type system very expressive, but is usually avoided because it does not enforce termination. We emphasize that in our setting, the type system does not need to do that, because termination for the underlying language is ensured prior to typing.

3 The untyped language of Q

The syntax of Q's untyped programming language is shown in Figure 1. Atomic terms are either variables x or labels l. Labels will be used to distinguish between different kinds of data. Terms can also be anonymous functions $\lambda\, x\,.\, s$, with the restriction that x may be used at most once in s. Such λ-abstractions are called *affine*. Some restriction is needed, or else it is very easy to write diverging λ-terms. Traditionally, type theories have restricted anonymous functions using types. Requiring λ-abstractions to be affine is a well-known if seldom used alternative.

Returning to the syntax: we have applications $t\ t'$ of a term t being used as a function to term t' given as the argument to that function. We have a trivial piece of data $\langle\rangle$, which is useful as a placehold. We have a way to form structured data $[l, n, r]$, and a term $r\ \bullet\ t$ for recursing over such data. These constructs constitute the untyped components of a version of what is known as W-types, and they will be presented in detail below. Finally, there is a label-matching function $\{l_1 \mapsto t_1\ ;\ \ldots\ ;\ l_k \mapsto t_k\}$, which will return term t_i if applied to label l_i.

Label lists	L	$::=$	$l_1, \ldots, l_k$
Annotated terms	*all meta-variables but L*	$::=$	$\star \mid x \mid \Pi\, x : T\,.\,T' \mid \lambda\, x : T\,.\,M \mid M\ N \mid$ $\iota_L\ l \mid \{L\} \mid \{l_1 \mapsto M_1\ ;\ \ldots\ ;\ l_k \mapsto M_k\} \mid$ $\epsilon\ M \mid \rho\,[x\,.\,T]\ M - M' \mid \xi\, M \mid T\{M = M'\} \mid$ $\langle\rangle \mid [s, n, f] \mid r\ \bullet\ t \mid \langle L, N, S\rangle$

Figure 2: Annotated terms of Q

4 Typed Q

Having considered the untyped language for Q, we now consider typing. The approach we describe is a form of Curry-style typing, and so the essential matter is to describe types and which untyped terms have which types. But we present the typing relation on annotated terms, so that there is a clear path to implementing the theory. Without annotations, deciding if a given term has a given type is undecidable, as the type theory subsumes System F [2]. But the annotations can all be erased, resulting in an untyped term in the syntax of Figure 1. We adopt a single syntactic category of expressions, containing both term and type constructs. This avoids duplicating constructions like λ-abstractions at the term and type levels.

The syntax of expressions is in Figure 2, the typing rules are in Figure 3, and erasure is defined in Figure **??**. We write $l \in L$ to indicate that label l occurs in the list L of labels. The term constructs fall into several groups, some of which we will explain in more detail below:

- Constructs from pure dependent type theory:
  - $\star$, the type for types
  - variables x
  - dependent function types $\Pi\, x : T\,.\,T'$
  - anonymous functions $\lambda\, x : T\,.\,M$, and
  - function calls $M\ N$.
- constructs for labels:
  - $\iota_L\ l$ to show into which label type $\{L\}$ the label $l \in L$ should be injected
  - label types $\{L\}$
  - label matchings $\{l_1 \mapsto M_1\ ;\ \ldots\ ;\ l_k \mapsto M_k\}$
- constructs for typed extensional equality:
  - $\epsilon\ M$ a proof of reflexivity
  - $\rho\,[x\,.\,T]\ M - M'$ a substitution principle, allowing to replace a term by an equal one in a type
  - $\xi\, M$ an extensionality principle
  - $T\{M = M'\}$ expressing that M and M' are equal at type T
- constructs for W-structures:
  - $\langle\rangle$, a trivial value
  - $[s, n, f]$, a W-structure
  - $r\ \bullet\ t$, a recursion over a W-structure
  - $\langle L, N, S\rangle$, a W-type

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\frac{}{\Gamma \vdash \star : \star}$$

$$\frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash T' : \star}{\Gamma \vdash \Pi\, x : T \,.\, T' : \star}$$

$$\frac{\Gamma \vdash T : \star \quad \Gamma, x : T \vdash M : T'}{\Gamma \vdash \lambda\, x : T \,.\, M : \Pi\, x : T \,.\, T'}$$

$$\frac{\Gamma \vdash M : \Pi\, x : T' \,.\, T \quad \Gamma \vdash N : T'}{\Gamma \vdash M\ N : [N/x]T}$$

$$\frac{l \in L}{\Gamma \vdash \iota_L\ l : L}$$

$$\frac{}{\Gamma \vdash \{L\} : \star}$$

$$\frac{\forall\, i \in \{1, \ldots, k\} \,.\, \Gamma \vdash M_i : T\ l_i}{\Gamma \vdash \{l_1 \mapsto M_1\ ;\ \ldots\ ;\ l_k \mapsto M_k\} : \Pi\, x : \{l_1, \ldots, l_k\} \,.\, T\ x}$$

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash \epsilon\ M : T\{M = M\}}$$

$$\frac{\begin{array}{l} \Gamma \vdash M : T'\{M_1 = M_2\} \\ \Gamma, x : T' \vdash T : \hat{T} \\ \Gamma \vdash M' : [M_1/x]T \end{array}}{\Gamma \vdash \rho\, [x \,.\, T]\ M - M' : [M_2/x]T}$$

$$\frac{\Gamma \vdash M : \Pi\, x : T \,.\, (T'\{M_1\ x = M_2\ x\})}{\Gamma \vdash \xi\, M : \Pi\, x : T \,.\, T'\{M_1 = M_2\}}$$

$$\frac{\begin{array}{l} \Gamma \vdash M_1 : T\{M_a = M_b\} \\ \Gamma \vdash M_2 : T\{M_a = M_b\} \end{array}}{\Gamma \vdash \xi\, M : (T\{M_a = M_b\})\{M_1 = M_2\}}$$

$$\frac{}{\Gamma \vdash \langle\rangle : \langle\rangle}$$

$$\frac{\begin{array}{l} \Gamma \vdash S : \star \\ \Gamma \vdash N : S \to \star \\ \Gamma \vdash R : S \to \star \end{array}}{\Gamma \vdash \langle S, N, R\rangle : \star}$$

$$\frac{\begin{array}{l} \Gamma \vdash s : S \\ \Gamma \vdash n : N\ s \\ \Gamma \vdash f : R\ s \to \langle S, N, R\rangle \end{array}}{\Gamma \vdash [s, n, f] : \langle S, N, R\rangle}$$

$$\frac{\begin{array}{ll} \Gamma \vdash S : & \star \\ \Gamma \vdash N : & S \to \star \\ \Gamma \vdash R : & R \to \star \\ \Gamma \vdash r : & \Pi\, s : S \,. \\ & \Pi\, n : N\ s \,. \\ & \Pi\, f : R\ s \to \langle S, N, R\rangle \,. \\ & (\Pi\, x : R\ s \,.\, C\ (f\ x)) \to \\ & C\ [s, n, f] \\ \Gamma \vdash d : & \langle S, R, R\rangle \end{array}}{\Gamma \vdash r\ \bullet\ d : C\ d}$$

Figure 3: Typing rules of Q

References

[1] Jean-Louis Krivine. *Lambda-calculus, types and models.* Ellis Horwood series in computers and their applications. Masson, 1993.

[2] J. B. Wells. Typability and type checking in system F are equivalent and undecidable. *Ann. Pure Appl. Log.*, 98(1-3):111–156, 1999.