Programming with dependent types

Wouter Swierstra
Universiteit Utrecht
The Netherlands
w.s.swierstra@uu.nl

Computer programs manipulate data. This data comes in many different shapes and sizes: the telephone numbers stored in our smartphone; the salary calculations done in Excel spreadsheets; or the customer addresses stored in a database. Many modern programming languages use a *type system* to classify data and rule out nonsensical calculations. For instance, multiplying a customer's telephone number and address is a nonsensical calculation. A (static) type system rules out such calculations before a program is run. Indeed, Simon Peyton Jones famously described static type systems as "*the world's most successful formal method.*"

Yet not all developers are enamored of statically typed languages. Some of the most common points of critique include:

- Simply typed languages can only prevent simple errors, such as mixing up the order of arguments passed to a method. In reality, programs are full of rich properties that cannot be enforced effectively by a type system.
- Requiring a program to be type correct before it can be executed bogs down the development process. Programmers should not spend their time writing code, rather than fixing type errors. Static types may help ensure safety, but makes programs harder to write.
- Not all type information may be known when a program is first written. Code that needs to interface with a database or other external data source may not know the type of all data involved before its execution. Similarly, some methods—such as `printf` or those using C's `varargs`—are notoriously difficult to type statically.

This note aims to illustrate how some of these limitations may be tackled by embracing *programming languages with dependent types*. In particular, it aims to show how current research on such languages provides a novel perspective on the benefits of statically typed programming and formal methods in general.

Types are not expressive

There is a deep connection between types and mathematical logic known as the *Curry-Howard correspondence* [Wadler 2015]. Simply stated, this correspondence states that every type system may be viewed as a logic. Every type in this system corresponds to a unique logical proposition; every program corresponds to a unique proof. To illustrate this point, consider the rules for *modus ponens* in logic and the typing rule for function application:

$$\frac{p \to q \qquad p}{q} \qquad\qquad \frac{f : a \to b \qquad x : a}{f\ (x) : b}$$

These rules are strikingly similar! The Curry-Howard correspondence shows how this is no coincidence—this similarity extends to richer logics and languages.

A common complaint about statically typed languages is *the lack of expressivity*, typically phrased in a sentence starting with "static types cannot…"—and such criticism is often correct for the type systems used by many mainstream languages. The 'logic' underlying such type systems is an (embarrassingly unsound) propositional logic in which you cannot express any interesting properties. To say anything meaningful about the behaviour of programs, we need *quantifiers* in our logic.

Viewed through the Curry-Howard lens, adding quantifiers our logic amounts to shifting from a simply typed programming language to one with *dependent types*. Per Martin-Löf was one of the first to propose a single dependently typed language to model proof and computation [Martin-Löf 1982]. Dependent types are the foundations on which modern interactive proof assistants, such as Coq and Lean, are built. To illustrate the importance of quantification, consider the following three type signatures:

$$f_1 : List\ Int \to List\ Int$$
$$f_2 : \forall\, a.Order\ a \to List\ a \to List\ a$$
$$f_3 : \forall\, a.Order\ a \to (xs : List\ a) \to \exists\ ys : List\ a, Sorted\ xs\ ys$$

Judging from its type, first function could do anything from reversing the list to incrementing every element. The type associated with f_2 is much more restrictive: the parametric polymorphism—induced by the universal quantifier—ensures that any elements in the output list must also occur in the input [Wadler 1989]. Crucially, such quantification is restricted to *types* in most languages. In contrast, the last type signature uses a *dependent type* to specify exactly that the list returned must be the sorted permutation of the input list *xs*.

Of course, *sorting* has a clear specification, that may be lacking in real world**TODO: don't say real world** code. Nonetheless, such code is typically full of invariants, many of which cannot be expressed in a simple type system. Instead of abandoning static typing altogether, it might be better to explore languages where these invariants can be described.

A Research Agenda for Formal Methods in The Netherlands, September 3–4, Lorentz Center, Leiden, The Netherlands
2018.

This offer programmers a spectrum of correctness: from basic code hygine to full blown mechanised proofs of functional correctness.

"Computer says know"

A common perspective on static types states that the type-correct programs are only a subset of all meaningful programs. A static type system polices the development process, slapping the wrist of any careless developer that dares adventure outside the subset of programs considered valid.

Personally, I prefer a slightly different perspective. Humans do not produce software by writing random strings and subsequently checking which ones happen to correspond to well-typed programs. Instead, software design is an intellectual activity. Consider the work by Felleisen How To Design Programs, one of the few methodological approaches to program development. We study a problem and break it into subproblems. We figure out the inputs and outputs of our functions. We write example inputs and outputs, and turn these into tests. Only then we start to code.

Types fit naturally in this philosophy. Viewing types as a partial specification; showing that the types of a large system fit together is a proof that the system satisfies its specification.

By writing types, the code follows naturally. IDEs with Intellisense use static type information to suggest how to complete a definition. Programmers who have worked with systems such as Agda and Idris, will acknowledge that most of the thought is in the design of the types; once these are known, the program writes itself (with a little help from the IDE).

Going even further, dataype generic programming shows how we can generate new functions from the structure of our data types.

The purpose of a static type system is not *only* to rule out bad programs ('computer says no'); a type provides information about the values that inhabit it ('computer says know'). There is more to types than *no-ing* bad programs; *knowing* the types provide a valuable clue to a programs development. As Conor McBride put it: "*is a type a lifebuoy or a lamp*?"

Just-in-time static typing

What if not all my types are known statically? One of the key rules in a dependently typed programming language is the *conversion rule*:

$$\frac{\Gamma \vdash t : \sigma \qquad \sigma \equiv_\beta \tau}{\Gamma \vdash t : \tau}$$

This rule states that types are interchangeable up to evaluation.

As a result, we can *compute* new types once we know *some* arguments. A common example is typing *printf* in this style:

But this is not the only such situation. For example, when interfacing with an unknown database, we might ask for a description of certain tables, parse the response, and compute the corresponding types.

Languages like F# have shown how type providers—facilitating the computation of new types from data—are very useful. Dependently typed languages take this one step further.

Looking ahead

Despite these advantages, there is still much research and development necessary before dependently typed languages can expect widespread adoption.

High barrier to entry – better type error messages, training material, etc.

Implementation quirks, unexplored design space, bells and whistles of Coq

Robust implementations, great IDEs, not always appreciated as novel research contributions; efficient compilation; integration with existing languages.

Not a silver bullet; still need testing; still immature in many respects; but an interesting idea that deserves further research.

References

Per Martin-Löf. 1982. Constructive mathematics and computer programming. In *Studies in Logic and the Foundations of Mathematics*. Vol. 104. Elsevier, 153–175.

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. ACM, 347–359.

Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84.