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, such as trying to multiply a customer's telephone number and address. 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 application of formal methods.*"

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 make programs inherently 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 more generally.

Static types cannot...

There is a deep connection between types and mathematical logic known as the *Curry-Howard correspondence*.Simply stated, this correspondence states that every type system may be viewed as a logic. Every type in written in a type 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 language constructs.

A common complaint about statically typed languages is *the lack of expressivity*. While static type systems are certainly no silver bullet, the properties that may be enforced by most type systems of mainstream languages is very limited. 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 express both proof and computation. To illustrate the importance of quantification, consider the following three type signatures:

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

Judging from its type, the 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.Crucially, the universal quantifier is restricted to abstract over *types* in most languages. In contrast, the last type signature uses *dependent types* to specify that for every input list *xs*, the function must compute a new list, *ys*, that must be the sorted permutation of *xs*.

Not all code has as clear a specification as sorting algorithms. Nonetheless, *all* code has some (partial) specification or invariant that programmers track in their head. Rather

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

than abandoning static typing altogether, wouldn't it be better to explore languages where these properties can be described and enforced? Doing so offers programmers a spectrum of correctness: from basic code hygiene 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.

I would like to offer a slightly different perspective. Developers do not produce software by writing random strings and subsequently checking which ones happen to correspond to meaningful programs. Instead, software design is a creative intellectual activity: a program is constructed methodologically,breaking a large problem into smaller pieces. Once the pieces are small enough, we can proceed to figure out the inputs and outputs of each piece. Programmers should write example inputs and outputs—turning these into tests—before implementing the functions that solve the individual problem pieces.

See how types fit naturally in this philosophy? Types form a partial specification; type checking the signatures of the individual pieces show how each individual function—once it has been implemented—may be composed to solve the original problem.

But types have much more to offer than such simple correctness properties. By starting to write down the types, the code often follows naturally. Integrated Development Environments (IDEs) such as Visual Studio use static type information to help a programmer navigate a complex codebase. Programmers who have worked with dependently typed systems such as Agda and Idris, will know how most of the thought goes into the careful design of the *types* of a function; once the types are fixed, the *program* almost writes itself. The programmer only need provide hints about which argument to pattern match on or when to make a recursive call; the IDE is often happy to use the static type information of the values in scope to find any missing values automatically.

Going even further, research on *dataype generic programming*has shown how we can generate new functions from the structure of our data types, or even refactor functions automatically exploiting structured changes to the types involved.

The purpose of a static type system is not *only* to rule out bad programs; a type provides information about the values that inhabit it. *Knowing* the types provide a valuable clue to a program's construction. To cite Conor McBride it: "*is a type a lifebuoy or a lamp*"?

Just-in-time static typing

How can static types help if the *types* of the data my program manipulates are not all known before a program is run? 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 equal if they *evaluate* to the same value. This simple rule has profound consequences: by mixing evaluation and type checking, it allows us to *compute* new types on the fly. For example, a function like *printf* is hard to assign a *single* static type, but the format string passed as its first argument may be used to *compute* the types of any remaining arguments that it expects.

There are many similar situations that simple statically typed languages cannot handle well. When interfacing with a database in a dependently typed language, for example, we might ask for a description of certain tables, parse the server's response, and compute the corresponding types. Even if these types are not know statically—we still have the guarantee that the pieces of our program will behave well when composed. *Type providers*, such as those implemented by F#, show how computing new types provide a welcome foothold when interfacing with foreign data. Dependently typed languages, that freely mix evaluation and type checking, take these ideas one step further.

Looking ahead

Despite their promise, there is still much research and development necessary before dependently typed languages can expect more widespread adoption. This research ranges from fundamental questions—such as finding a suitable definition of equality—to more mundane engineering challenges.

Many beginners still experience a high barrier to entry when learning to program with dependent types. Better training material, more informative type error messages, and more robust compiler implementations would all facilitate their more widespread adoption. With the richer design space that dependent types offer, beginners often find themselves making the wrong design choice early on in the development process. Categorising the design principles of experienced users and facilitating the automatic refactoring of typed programs would certainly help explore this richer design space—and prevent beginners from painting themselves into a corner.

Static types cannot possibly solve all the problems in the construction of modern software. But the field of programming with dependent types, while based on fundamental research almost a century old, addresses many of the limitations of traditional static type systems, offering new perspectives on the future of high-assurance program development.