A deep link between logic and programming

Matthías Páll Gissurarson
pallm@student.chalmers.se

2016-11-16

There is a deep link between logic and programming, that connects the world of abstract reasoning and computing in a fundamental and universal way, and that is the notion of propositions as types. The notion is that very fundamental logic such as propositional, predicate, and second-order logic, as well as logic systems such as natural deduction and sequent calculus can be expressed by types in a programming language (specifically the typed lambda calculus), and programs in that language are then proofs of those propositions.

Here programming languages refer to a set of instructions to be carried out by a computer, and types are a way of marking values in programming languages so that we can check whether the value can be used in some way. For example, if we have a numerical value in a programming language, we can mark it as such and then have the computer make sure that we don't mix up numbers and letters for example. Another example might be the way that physicists use different units of measurements for different things. They then commonly use these units to guide their calculations, and make sure that they end up with units that match the problem, such as getting $\frac{km}{s}$ for the speed of a car, and not $\frac{elephants}{km^2}$.

The notion of propositions as types in programming languages was discovered when a connection between the rules for lambda calculus, a set of rules and notation to describe a kind of very simple programming language, and the rules of natural deduction, a formal system for proving propositions, was observed.

Lambda calculus was initially presented by Alonzo Church as a way to define notations for logical formulas, like a macro language [1]. It is very compact, and contains only three constructs:

- Variables: Variables are a label represented by a letter (e.g. x), and used to denote some value that is not yet defined (such as true or false). A variable is in itself a term in the lambda calculus.
- Function abstraction: If t is a term and x is a variable, then the term $(\lambda x.t)$ is a function abstraction.
- Function application: If t and s are lambda terms, then $(t\ s)$ is a function application. An example would be that if t is $(\lambda x.\ x\ x)$ and s is the variable y, then $((\lambda x.\ x\ x)(y))$ is the application of t to s.

Computation in the lambda calculus is then done by normalizing lambda terms as follows: If t is a lambda term containing x and $(\lambda x.t)$ is a lambda abstraction, then $((\lambda x.t)y)$ is normalized to the term t', where all instances of the variable x has been replaced by the variable y.

The simple lambda calculus is however inconsistent [1], and as such not a good way to describe logical notations. This was however rectified by introducing types into the language, much as Russell and Whitehead solved their paradox by introducing types. The resulting calculus was then known as the typed lambda calculus.

The programming language we will talk about in this summary is the typed lambda calculus, since it is simple and easy to reason about, it forms the basis of all functional languages, and it presents an intuitive connection between natural deduction, the sequent calculus and programming.

Natural deduction is a formal logic framework to formally write down proofs based on strict rules, so as to not allow any hand-waving in the proofs themselves where not explicitly stated. It was developed by Gerhard Gentzen in 1935 [1], and he showed that there was a way to normalize proofs in it so that they were not round-about, i.e. that the proofs themselves did not contain anything not stated as preconditions for the proof. He had the insight that formal rules of logic for proofs should come in pairs, one rule to introduce a new variable, and another to eliminate a variable from the proof, i.e. one way to construct a value and another to use that value in some way.

Other approaches to programming such as imperative programming that are based on the computing ability of Turing machines [2] are also related to logic systems, like Turing machines are related to dynamic logic [3]. These logic systems are defined after the invention and acceptance of computers precisely to match these programming models, so the connection between intuitive logic and these systems is less clear.

This notion is often referred to as the Curry-Howard correspondence, and was discovered by Haskell Curry in 1934 [4] and refined by William Howard in 1969 [5]. The article we are summarizing is itself a kind of summary of the topic and its history by Philip Wadler, and describes the notion, its history, and inspirations in simple terms.

By viewing values in a program as axioms or already proven propositions, and functions as instructions on how we can prove a new proposition given some previous proposition, the connection is clear. We then say that the proposition has some type, and functions have a type from some proposition to another. In a simple example we could have a function that says "if your name is John, then Bob's your uncle", and we're provided with a proof of the proposition that your name is indeed John, then using the provided proof and the function, we can produce a proof that Bob's your uncle. Here the proposition has a type of "Your name is John", and the function has a type of "Your name is John" to "Bob's your uncle".

A more in-depth example would be that if we have a given function f of type $A \to B$ (we denote this with $f : A \to B$), then given a value of type A, we can apply the function to the value and get a value of type B. This is similar to natural deduction, where if we have an implication (incidentally also written as $A \to B$ in current standard notation) and a proposition of type A, we can use the implication elimination rule and get a proof of B. This similarity inspired Howard to make the following observations:

- Conjunction: A and B (written $A \wedge B$) in natural deduction corresponds to a Cartesian product (an ordered pair) of type $A \times B$. This follows from the observation that to prove

(called introduction in natural deduction) $A \wedge B$, we must prove A, and we must prove B. To construct a Cartesian product of type $A \times B$, we must provide a value of type A and a value of type B. If we have a value of type $A \times B$, we can extract from it a value of A and a value of B. Similarly for conjunction, if we have a proof of $A \wedge B$ in natural deduction, we can extract from it a proof of A and a proof of B.

- Disjunction: A or B (written $A \vee B$) in natural deduction corresponds to a sum type, i.e. a type with two alternatives. A sum type is a type denoted by $A \mid B$, and is a type whose value is either of type A or of type B, with a mark to distinguish between them. To introduce (prove) $A \vee B$, we must provide a proof of either A or of type B, and indicate which one it is that we proved. To construct a value of type $A \mid B$, we must provide a value of either type A or B and denote which one of them it is. If we have a proof of a disjunction, we can extract a proof of A or a proof of B by using the mark to distinguish which one of them the proof is of. Similarly, to extract a value of type A or type B from a provided value of type $A \mid B$, we can use the supplied mark to know which type of value it is and extract the value of type A or type B accordingly.
- Implication: As mentioned before, the proposition that A implies B (written as $A \to B$) in natural deduction corresponds to functions and function application. This follows from the observation that to prove that $A \to B$ in natural deduction, we must assume that we have a proposition A, and prove that B follows from A. To create a function of type $A \to B$, we must provide a function that, given a value of type A produces a value of type B. As previously described, function application corresponds to the application of implication in natural deduction.

By using the notion of propositions as types to connect the worlds of logic and programming, we can use programs to prove theorems that may otherwise have been intractable by hand, while still being able to convince ourselves of the correctness of the proofs that the program provides. The most famous example of this is perhaps the formal computer-generated proof of the four color theorem in the Coq programming language [6], a language that uses propositions as types as well as a rich type system to express those types as a base for creating programs that are verified, and programs that represent proofs of mathematical theorems. The notion of propositions as types is used to prove that four colors are sufficient to color any planar graph, without human intervention (modulo the initial programming of course). The four color theorem is notorious for being the first long outstanding theorem to be proven with the assistance of a computer [7], but the initial proof was in a language which does not include this notion as a paradigm. This left a lot of mathematician unsatisfied, since it only demonstrated the fact correct, but did not provide any insight into the problem itself or the patterns at work. This cast doubt on the correctness of the proof as well, since the program had to be verified by hand, with few built-in logical guarantees to assist with the verification. When the notion of types as propositions was later used to prove the theorem, it left little doubt of the proofs correctness. This was mainly due to the strong formal foundations that the notion provided to the proof.

Other concrete uses of this notion include CompCert [8], an optimizing compiler for all of the C programming language, a proof of correctness of the disjoint-set data structure [9], and a proof of the Feit-Thompson theorem [10]. The increased use of proof assistants shows the power of this notion in helping mathematicians go beyond what was previously possible with the help of computers, but on solid foundations by using propositions as types.

The notion also makes other benefits from logic available in programming, since we can use methods and principles from logic to assure us that programs that we write in languages

that embed the notion are correct, since the compiler can check that the types are correct, which ensures (and provides a proof of!) that no logical structure embedded at the type level is misused in the program. This can give the programmer some peace of mind since a large source of errors can be eliminated right from the beginning. This also benefits businesses, since by employing a programming language embedding this notion, they can get some guarantees about the reliability and correctness of the program that would otherwise have to be checked by hand (in an often error prone way).

How well we can utilize this connection between propositions and types depends on how expressive the type system in a given language is. This is analogous to how expressiveness affects what we can propose, and thus what we can prove. There is also danger inherent in having too much expressiveness, as Gödel [11] proved with his incompleteness theorems, that too much expressiveness allows us to propose either unprovable or untrue propositions.

Some languages such as Coq and Agda [12] implement type systems where the type definition can depend on a value, called dependent types [13]. In a fully dependent type system, there is no distinction between terms and types, and as such most of what is possible to do in the language itself is possible in the type system. As an example, this allows for there to be a type for each of the natural numbers, allowing us to express lists of a given length or arrays of a given size in the type system. That then allows us to prove that a function always returns a list of a given length, and allows us to avoid out-of-bounds errors, since we know precisely the allowed indices of the list.

In conclusion, the notion of propositions as types forms a deep connection between logic and programming and is a point in favor of the view that concepts in mathematics and computer science are discovered, not invented. The structure of certain programming languages (namely functional programming languages) is closely related to the structure of logic, and it is hard to conceive of logic having a fundamentally different structure than we have seen before. This can inform our views of programming languages and suggests that functional languages may be somehow more fundamental than others, and at the very least more universal than other paradigms [1].

References

[1] P. Wadler, "Propositions as types," *Communications of the ACM*, vol. 58, no. 12, pp. 75–84, 2015.

[2] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[3] M. J. Fischer and R. E. Ladner, "Propositional dynamic logic of regular programs," *Journal of computer and system sciences*, vol. 18, no. 2, pp. 194–211, 1979.

[4] H. B. Curry, "Functionality in combinatory logic," *Proceedings of the National Academy of Sciences*, vol. 20, no. 11, pp. 584–590, 1934.

[5] W. A. Howard, "The formulae-as-types notion of construction," *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pp. 479–491, 1980. The original version was circulated privately in 1969.

[6] G. Gonthier, "Formal proof–the four-color theorem," *Notices of the AMS*, vol. 55, no. 11, pp. 1382–1393, 2008.

[7] R. Thomas, "An update on the four-color theorem," *Notices of the AMS*, vol. 45, no. 7, pp. 848–859, 1998.

[8] X. Leroy, "The compcert c verified compiler," *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[9] S. Conchon and J.-C. Filliâtre, "A persistent union-find data structure," in *Proceedings of the 2007 workshop on Workshop on ML*, pp. 37–46, ACM, 2007.

[10] G. Gonthier, "Engineering mathematics: the odd order theorem proof," in *ACM SIGPLAN Notices*, vol. 48, pp. 1–2, ACM, 2013.

[11] K. Gödel, "Über formal unentscheidbare sätze der principia mathematica und verwandter systeme i," *Monatshefte für mathematik und physik*, vol. 38, no. 1, pp. 173–198, 1931.

[12] U. Norell, "Dependently typed programming in agda," in *Advanced Functional Programming*, pp. 230–266, Springer, 2009.

[13] R. Eisenberg, "Dependent types in Haskell: Theory and practice." PhD thesis draft, University of Pennsylvania, 2016.