Chapter 1

Introduction

Programs are a unique engineering material as they can be constructed and reasoned about entirely abstractly and even mathematically: while solutions to engineering problems in general require more than mathematical reasoning, program correctness with respect to their formal specifications — a mathematical subject in itself — is a concern separable from others, like suitability of the specifications with respect to the actual engineering problems (see, e.g., Dijkstra [1982]). One possible approach to program correctness is by **verification**, in which a program is first written and then proved correct separately. For example, let a specification for imperative programs be a pair of predicates on machine states — called the precondition and the postcondition — expressed in first-order logic; a program meets the specification exactly when its execution transforms any state satisfying the precondition to one satisfying the postcondition, which can be proved with Hoare logic [Hoare, 1969], i.e., by annotating the program with Hoare triples. With this approach, however, the construction of a program is in general detached from the construction of its correctness proof; program development becomes strictly harder, since in addition to program construction, there is now the extra burden of proof construction. The approach was dismissed by Dijkstra as "putting the cart before the horse" [Dijkstra, 1974], who instead argued that correctness proofs should be developed hand in hand with — and even slightly ahead of — the programs, so correctness proofs can "act as an inspiring heuristic guidance" on the exploration of the program space,

thereby facilitating program construction rather than merely being an extra burden. This leads to the methodology of **program correctness by construction**. For example, instead of verifying the correctness of an imperative program by subsequent annotation, we can simultaneously derive the program and its correctness proof from the specification in the first place using the weakest precondition calculus [Dijkstra, 1976; Gries, 1981; Kaldewaij, 1990], with the whole development driven by the force of transforming the postcondition to the precondition.

On the other hand, Martin-Löf developed **intuitionistic type theory** [Martin-Löf, 1975, 1984; Nordström et al., 1990] to serve as a foundation of intuitionistic mathematics [Heyting, 1971; Dummett, 2000], like Bishop's renowned work on constructive analysis [Bishop and Bridges, 1985]. Central to the design of intuitionistic type theory (which we simply call "type theory" henceforth) is the **propositions-as-types principle**, which states that the notion of propositions and proofs is subsumed by the notion of types and programs (see **??**). Consequently, type theory can be regarded simultaneously as a computationally meaningful higher-order logic system and an expressively typed functional programming language (whose type system is commonly referred to as **dependent types**) — proof rules for constructing formal derivations in logic are typed primitive terms in the programming language, and valid derivations of propositions are typechecked programs. Compared with Hoare logic, in which an imperative program can be similarly regarded as a formal derivation of a transformation from one predicate to another, type theory is a more natural and powerful system for achieving program correctness: A specification for functional programs is simply a proposition — i.e., a type — rather than a predicate transformer (an apparently more complicated notion). Moreover, in type theory, the proof rules available for deriving propositions are the standard, structural ones (introduction and elimination rules for every type former), whereas in Hoare logic, the available primitive transformers are more contrived, catering for state-based computation. Most importantly, Hoare logic merely imposes a special-purpose proof system on imperative programs, whereas type theory is based on a fundamental unification of proofs and programs, allowing them to coexist in a uniform language and interact freely — in particular, we can state

and prove various properties about existing programs all in the same language, with the proving process being just additional programming.

The methodological distinction between program correctness by verification and by construction exists for type theory as well. With the former methodology, we first write a program under a simple type (just informative enough for sanity check) and then separately prove that the program satisfies some correctness property; with the latter methodology, the correctness property is encoded into a more sophisticated type such that any program having the type is automatically guaranteed to be correct by typechecking. The distinction is most significant when **inductive families** [Dybjer, 1994] — simultaneously inductively defined families of types, sometimes simply referred to as "indexed datatypes" — come into play. A type in general can be thought of as expressing a programming problem to be solved (which was Kolmogorov's interpretation of intuitionistic propositions; see, e.g., Martin-Löf [1987]); the use of an inductive family in a type suggests a particular decomposition of the problem into sub-problems (see **??**), and hence has a strong influence on the structure of programs having that type. When inductive families are used for defining predicates that state properties about existing programs, they can influence strategies for discharging proof obligations; more interestingly, when they are used as datatypes that are directly programmed with, they can effectively guide program construction by directly dictating the structure of conforming programs in unprecedented detail. The latter mode of programming, called **internalism** in this dissertation, is the most distinguishing feature of **dependently typed programming** [McBride, 2004; Altenkirch et al., 2005], while the former mode is called **externalism**.

Both internalism and externalism are indispensable in dependently typed program development (see **??**): key correctness properties can provide useful guidance on program construction and are best treated in the internalist way, but there are also some other properties that are separable concerns and thus should be dealt with separately with externalism. Whereas externalism is a well understood concept, being deeply rooted in the mathematical tradition, techniques and mechanisms for facilitating internalist programming are relatively lacking. To provide more support for internalism, this dissertation proposes that internalist programming can be facilitated by exploiting an **interconnection**

between internalism and externalism, expressed as isomorphisms for converting between internalist and externalist representations of data structures. Here an internalist representation refers to an inductive family used as a datatype with some built-in data structure invariant, while an externalist representation is a basic datatype paired with a separate predicate stating the invariant. The interconnection consists of two directions:

- one **analysing** any internalist datatype D into a basic datatype D' known to be related to D and a predicate on D' stating the invariant encoded in D, and
- the other **synthesising** new internalist datatypes from a basic datatype and a given class of predicates on the basic datatype.

More specifically, this dissertation shows that

- the analytic direction can be exploited for achieving a modular structure of libraries of internalist datatypes, and
- the synthetic direction for bridging internalist programming with the relational revision of Bird–Meertens formalism [Bird and de Moor, 1997].

After **??** covers some preliminary background, the above two applications and their supporting mechanisms are respectively presented in Chapters **??** and **??**. The supporting mechanisms are based on McBride's **ornaments** [2011]:

- modular library structure in **??** is achieved with **parallel composition** of ornaments, and
- the bridging of internalism with relational programming in **??** is done by (relational) **algebraic ornamentation**.

Some further theoretical characterisations of the two mechanisms using lightweight category theory are respectively presented in Chapters **??** and **??**. Finally **??** concludes.

Formalisation. Except for a major part of **??**, the constructions in this dissertation are completely formalised in the dependently typed programming language Agda [Norell, 2007, 2009; Bove and Dybjer, 2009]. The code base is available at `https://github.com/josh-hs-ko/Thesis`. □

Previous publications. **??** is based on the paper *Modularising inductive families* published in the *Progress in Informatics* journal [Ko and Gibbons, 2013a], and **??**

on the paper *Relational algebraic ornaments* presented at the *Workshop on Dependently Typed Programming* [Ko and Gibbons, 2013b]. All technical contributions of both papers are from the first author. □

Bibliography

Thorsten Altenkirch, Conor McBride, and James McKinna [2005]. Why dependent types matter. URL: http://www.cs.nott.ac.uk/~txa/publ/ydtm.pdf. ↰ page 3

Richard Bird and Oege de Moor [1997]. *Algebra of Programming*. Prentice-Hall. ISBN: 978-0135072455. ↰ page 4

Errett Bishop and Douglas Bridges [1985]. *Constructive Analysis*. Springer-Verlag. ISBN: 978-3642649059. ↰ page 2

Ana Bove and Peter Dybjer [2009]. Dependent types at work. In *Language Engineering and Rigorous Software Development*, volume 5520 of *Lecture Notes in Computer Science*, pages 57–99. Springer-Verlag. doi: 10.1007/978-3-642-03153-3_2. ↰ page 4

Edsger W. Dijkstra [1974]. Programming as a discipline of mathematical nature. *American Mathematical Monthly*, 81(6):608–612. doi: 10.2307/2319209. ↰ page 1

Edsger W. Dijkstra [1976]. *A Discipline of Programming*. Prentice-Hall. ISBN: 978-0132158718. ↰ page 2

Edsger W. Dijkstra [1982]. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag. doi: 10.1007/978-1-4612-5695-3_12. ↰ page 1

Michael Dummett [2000]. *Elements of Intuitionism*. Oxford University Press, second edition. ISBN: 978-0198505242. ↰ page 2

Peter Dybjer [1994]. Inductive families. *Formal Aspects of Computing*, 6(4):440–465. doi: 10.1007/BF01211308. ↰ page 3

David Gries [1981]. *The Science of Programming*. Springer-Verlag. ISBN: 978-0387964805. ↰ page 2

Arend Heyting [1971]. *Intuitionism: An Introduction*. Amsterdam: North-Holland Publishing, third revised edition. ISBN: 978-0720422399. ↰ page 2

C. A. R. Hoare [1969]. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580. doi: 10.1145/363235.363259. ↰ page 1

Anne Kaldewaij [1990]. *Programming: The Derivation of Algorithms*. Prentice-Hall. ISBN: 978-0132041089. ↰ page 2

Hsiang-Shang Ko and Jeremy Gibbons [2013a]. Modularising inductive families. *Progress in Informatics*, 10:65–88. doi: 10.2201/NiiPi.2013.10.5. ↰ page 5

Hsiang-Shang Ko and Jeremy Gibbons [2013b]. Relational algebraic ornaments. In *Dependently Typed Programming*, DTP'13. ACM. doi: 10.1145/2502409.2502413. ↰ page 5

Per Martin-Löf [1975]. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier B.V. doi: 10.1016/S0049-237X(08)71945-1. ↰ page 2

Per Martin-Löf [1984]. *Intuitionistic Type Theory*. Bibliopolis, Napoli. ↰ page 2

Per Martin-Löf [1987]. Truth of a proposition, evidence of a judgement, validity of a proof. *Synthese*, 73(3):407–420. doi: 10.1007/BF00484985. ↰ page 3

Conor McBride [2004]. Epigram: Practical programming with dependent types. In *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer-Verlag. doi: 10.1007/11546382_3. ↰ page 3

Conor McBride [2011]. Ornamental algebras, algebraic ornaments. URL: https://personal.cis.strath.ac.uk/conor.mcbride/pub/OAAO/LitOrn.pdf. ↰ page 4

Bengt Nordström, Kent Peterson, and Jan M. Smith [1990]. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford University Press. ISBN: 978-0198538141. ↰ page 2

Ulf Norell [2007]. *Towards a Practical Programming Language based on Dependent Type Theory*. Ph.D. thesis, Chalmers University of Technology. ↰ page 4

Ulf Norell [2009]. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag. doi: 10.1007/978-3-642-04652-0_5. ↰ page 4

Todo list