Libraries You Can Trust:

High-level Specifications and Correct Implementations via Dependent Types

1 Purpose and aims

Our long term goal is to create systems (theories, programming languages, libraries and tools) which make it easy to develop software components and matching specifications. In this research project, we aim to leverage the power of languages with strong types to create libraries of components which can express functional specifications in a natural way, and, simultaneously, implementations which satisfy those specifications. The ideal we aim for is not merely correct programs, nor even proven correct programs; we want proof done against a specification that is naturally expressed for a domain expert.

Concretely, we aim to identify common patterns in the *specification* of programs, and capture those in libraries. At the same time, the patterns of *implementations* of these specifications will also be captured in the library, such that the development of software will go hand-in-hand with proofs of its functional correctness. As case-studies we will work in two areas: the Algebra of Parallel Programming (inspired by the Algebra of Programming [Bird and de Moor, 1997]) and Domain-specific modelling of global systems.

Case AoPP: We want to tackle one of the key problems confronting the computing world today: that of correctly implementing scalable parallel computations. At the core of this problem is divide&conquer: divide a workload (recursively) into parallel tasks and combine the results. We focus on the algebraic structure of this core and develop libraries for correct-by-construction parallel programming.

Case DSM: Climate impact researchers heavily depend on computer models to simulate the co-evolution of climate and society under different scenarios. There is currently a large gap between the mathematical models of the researchers and the codes they run and this makes it really hard to distinguish between an implementation error and a successful climate policy. We focus on formalising the key concepts in a domain specific language for correct-by-construction Global Systems Simulations (GSS).

Note that what we call "software" or "program" in this proposal is not just "*a sequence of instructions, written to perform a specified task with a computer*" [Wikipedia, Computer Program]. In our terminology a "program" is a more high-level description of what a computer should compute (perhaps closer to the meaning of "algorithm") or even, in the case of domain-specific modelling, a high-level description of a problem or a solution in a particular application domain (be it financial contracts, partial differential equations or formal proofs). This means that the iterative process of coming up with a "program" matching a "specification" is not limited to traditional programming, but it is also a way of exploring and understanding a domain. A resulting library of specifications can be seen as a collection of building-blocks for describing (problems in) the domain and a library of implementations can be seen as computer aided methodologies for computing with, or solving problems within, that domain.

2 Research area overview

Abstraction. The ability to name and reuse parts of algorithms is one of the cornerstones of computer science. Abstracting out common patterns enables separation of concerns, both in the small (variables, functions) and in the large (modules, libraries). Conversely, lack of abstraction may force the implementation to contain multiple instances of a single pattern. This process of replication is not only tedious, but error-prone, because the risk of software error is directly correlated with the size of the program. Hence, one important trend in the evolution of new programming languages is improved support for abstraction—making more and more of the language features programmable. Widely used languages such as Java, C++, Scheme and Haskell are actively gaining abstraction power with Java Generics, C++ Templates, Scheme's composable macros, and Haskell meta-programming, respectively. But power always comes at a price: in this case, without proper checking, more complex features can increase the risk of bugs and unintended behaviour. Thus, with new abstraction mechanisms we also need new, preferably computer-aided, mechanisms for checking the program code.

Types. Types are used in many parts of computer science to organise the different kinds of values and to prevent software from going wrong. In a nutshell, types enable the programmer to keep track of the structure of data and computation in a way that is checkable by the computer itself. Effectively, they act as contracts between the implementer of a program part and its users. If type-checking is performed statically, when the program is compiled, it then amounts to proving that properties hold for all possible executions of the program, independently of its input.

By the Curry–Howard correspondence, type systems are directly related to logics. Rich type systems, such as those for languages with higher-order abstraction, correspond to higher-order logics. A well-know example of a system based on this principle is the Coq proof assistant [The Coq development team, 2010].

Dependently typed programs. Even though type theory has been used as a logic for decades, it has only recently gained popularity as a medium for programming. The viability of dependent types in a substantial "real world" example was perhaps first demonstrated by CompCert, a C compiler written and verified in Coq [Leroy, 2009]. Other applications are rapidly appearing: verification of imperative programs within Coq [Chlipala et al., 2009], database access with static guarantees [Oury and Swierstra, 2008], distributed programming [Swamy et al., 2011] resource-safe programs [Brady and Hammond, 2012] A recent grant from the National Science Foundation is exploring "The Science of Deep Specification" using dependently typed programs and proofs ($10M, 2016–2021).

Agda. The programming language Agda is a system based on Martin-Löf type theory [Martin-Löf, 1984]. Within it, one can express programs, functional specifications, and proofs in a single language (by taking advantage of the Curry–Howard correspondence). Agda is currently emerging as a lingua-franca of programming with dependent types. Its canonical reference, Norell's thesis [2007], has been cited 60 times per year since its publication, indicating strong academic interest. Additionally, there are several high-quality video introductions to Agda available on the Internet, produced by scientists with no affiliation to Chalmers. These range from short demonstrations (e.g., L. Maydwell's 15 min. available on YouTube since 2012) to mini-courses consisting of several lectures (e.g., D. Licata's *Dependently-Typed Programming in Agda*, presented at the Oregon Prog. Lang. Summer School in 2013). Accordingly, the focus of this project is on expressing libraries of correct programs and proofs in Agda.

Libraries for dependent types. Strongly typed languages, such as Agda and Coq, come with standard libraries that contain useful building blocks to create programs, specifications, and proofs. The Coq library is part of a mature system which has been used in many projects (sometimes complemented by extensions such as Ssreflect [Gonthier, 2009]). However, it places a strong emphasis on proofs rather than programs, reflecting the fact that the Coq system is mostly intended as a proof assistant. Projects which aim to use Coq for program development, such as Ynot and CompCert [Chlipala et al., 2009; Leroy, 2009], do not result programs written in the dependently-typed language of Coq. This language is used to specify and prove the correctness of computations implemented in other (not dependently-typed, imperative) languages.

The same observation applies to the libraries of most systems with dependent types. In contrast, the Agda standard library has evolved from common abstractions needed by Agda programmers and thus emphasises programs rather than proofs. This is demonstrated by its use in programs across several fields, in particular parser combinators [Danielsson, 2010], Algebra of Programming [Mu et al., 2009] and Cryptography [Pouillard and Gustafsson, 2015].

3 Project description

Our project will be organised in multiple iterations, each refining the libraries developed in the previous one. The first iteration is based on our current experience with libraries built in the functional programming languages Haskell and Agda. Each iteration will have the following three phases.

1. **Development of a proven-correct application in a given domain.** We believe that the best way to develop libraries is by abstracting common patterns found in various applications. In this phase, we will assess the viability of our libraries by applying them to different applications ranging from classical problems of computer science to global systems science applications (details in the following subsections).

2. **Extraction of common patterns into libraries.** In this phase, we will identify common patterns found in the programs and specifications produced in the previous phase, and capture them in libraries. At the same time, we will tie each pattern of specification to one or more patterns of implementation. We will then reimplement the application previously produced using the library.

3. **Refinement of the programming language.** In this phase we will assess the strong and weak points of the underlying programming environment we use. We will inform the group in charge of the development of the language of the possible shortcomings we might identify and participate in their remedy.

The iterative refinement steps will be used in two major case studies described in the following subsections: **Algebra of Parallel Programming** and **Domain-Specific Modelling** of global systems.

3.1 Algebra of Parallel Programming (AoPP)

Dependent type theory is rich enough to express that a program satisfies a functional specification, but there is no *a-priori* method to derive a program once the specification-as-type is written. On the other hand, Bird and de Moor [1997] give a general methodology to derive Haskell programs from specifications, via algebraic reasoning. Despite the strong emphasis on correctness, their specifications and proofs are not expressed in a formally checkable way. In [Mu et al., 2009] we have shown how to encode Bird and de Moor–style program derivation in the dependently typed programming language Agda. A program is coupled with an algebraic derivation from a specification, whose correctness is guaranteed by the type system. We believe that this approach is useful in tackling one of the key problems confronting the computing world today: that of correctly implementing scalable parallel computations. This is our aim in the first case study.

At the core of scalable parallel programming is the ability to divide a workload into two independent tasks (which can be run in parallel) in such a way the solutions can be easily combined into a final result. This subdivision can be done recursively to the depth needed to effectively use the available hardware parallelism. We want the results of the parallel computation to be independent of the number of divisions of the workload, otherwise we would obtain different results on machines with different numbers of processors. Similarly, we want to obtain the same result irrespective of the order in which the individual tasks terminate.

These conditions are met by a large class of algorithms, namely those which are monoid homomorphisms. That is, a function $f : A \rightarrow B$ can be parallelised if it satisfies the following laws:

$$\begin{aligned} f\ \mathit{empty}_A &= \mathit{empty}_B \\ f\ (a \mathbin{+\!\!+}_A b) &= f\ a \mathbin{+\!\!+}_B f\ b \end{aligned}$$

where empty and $+\!\!+$ denote monoidal unit and composition (for the type in the subscript). Often, a function which is quite not a monoid homomorphism can be formulated in terms of an auxiliary function, which works on an extended type.

A simple example is word counting, which maps strings to natural numbers (the number of white-space separated words in the string). The monoid structures are concatenation with the empty string as unit for the domain, and addition with zero as unit for the co-domain. It is easy to see that word counting is not a homomorphism: if we cut a string containing a single word (no spaces) in two, each sub-task will count one word, and the addition of the two independent results will return the erroneous count of two.

The reason for that is that we have lost the information about the (lack of) spacing. To avoid this loss, we need to preserve the information about spacing on either side, in addition to counting the number of full words.

$$\begin{array}{lll} \mathit{wordCount} & : & \mathit{String} \rightarrow \mathbb{N} \\ \mathit{wordCount} & = & \mathit{extractCount} \circ \mathit{helper} \\ \mathit{helper} & : & \mathit{String} \rightarrow \mathit{CountAndSpacing} \quad \text{-- homomorphism} \\ \mathit{helper} & = & \mathit{divideAndConquer}\ \mathit{baseCase}\ \mathit{combine} \\ \mathit{extractCount} & : & \mathit{CountAndSpacing} \rightarrow \mathbb{N} \quad \text{-- simple projection} \\ \mathit{baseCase} & : & \mathit{String} \rightarrow \mathit{CountAndSpacing} \end{array}$$

$$
\begin{array}{lcl}
\mathit{combine} & : & \mathit{CountAndSpacing} \to \mathit{CountAndSpacing} \to \mathit{CountAndSpacing} \\
\ldots & &
\end{array}
$$

In this simple case, inventing the $\mathit{CountAndSpacing}$ type and the helper function does not require much ingenuity[1]. But when we move on to more realistic examples, such as that of parallel parsing, the type transformation is more challenging.

A parser is a program that analyses a piece of text to determine its logical structure. Parsing is, at least at first sight, an inherently sequential process, especially when it comes to formal texts, such as program code, since lines of code often require the surrounding context in order to make sense.

Nevertheless, inspired by sparse matrix algorithms and the work of Valiant [1975] on language recognition, we have been able to create a suitable analogue of the $\mathit{CountAndSpacing}$ type for parsing, leading to "*Efficient Parallel and Incremental Parsing of Practical Context-Free Languages*" [Bernardy and Claessen, 2013].

Perhaps more importantly, when formalising the proofs of correctness of the parallel parsers, we have been able to use the Bird and de Moor approach in order to *calculate* the parallelisation from the specification of parsing [Bernardy and Jansson, 2015]. This kind of program calculation is at the core of what we call the Algebra of Parallel Programming (AoPP). The resulting parsing algorithm is based on matrix algebra where the matrix elements are sets of non-terminals (often the empty set when there is no parse for a substring). The matrix representation is based on quad-trees to exploit the sparseness and the algebraic structure turns out to be related to semi-rings. (We intend to further explore this connection to linear algebra).

In this project, we aim to develop and use AoPP in order to

- calculate parallel algorithms for optimisation algorithms (and other numerical methods required by the domain-specific modelling case study),
- calculate parallel versions of certain graph algorithms (like path finding),
- develop general principles for the construction of the intermediate datatypes and helper applications.

3.2 Domain-specific modelling (DSM)

The proof that an implementation satisfies a specification is not very valuable if the specification is expressed in an incomprehensible fashion. To be useful, specifications must be understandable to domain experts, and therefore it is important for computer scientists to work with the domain-specific concepts. We have done so in the past, in the domain of vulnerability for climate impact research [Lincke et al., 2009], grammars for language processing [Duregård and Jansson, 2011], and more recently, Walras equilibria and Pareto-efficiency for economics [Ionescu and Jansson, 2013a].

In the case of economics, the resulting specifications, while quite close to the mathematical formulations that the modellers are used to, are *non-constructive* in nature and can only be implemented in restricted settings (for example, when all the types involved are finite). To deal with more general cases will involve a great deal of innovation, both in

[1]Take $\mathit{CountAndSpacing} = \mathbb{B} \times \mathbb{N} \times \mathbb{B}$ for "space at start?", "word count", "space at end?".

the concepts to be specified (consider the difference between discrete and continuous mathematics) and in the numerical methods to be implemented. To bring about such innovations is our aim in this area of the project.

The chosen application area for the first phase of our iterative development is that of models of emissions trading and, more generally, international environmental agreements. We plan to build on the previous work by formalising, together with the domain experts of the Potsdam Institute for Climate Impact Research, the key concepts of such models. (*player, coalition, market, stability, free-riding*, etc.). This requires specifications and proofs for higher-order constructions such as functors, monads, and vulnerability measures [Ionescu, 2009].

In the second phase, we will develop a library of common patterns, creating a domain specific language for expressing models of emission trading and coalition formation.

And in the third phase we will suggest ways of improving language support to present the specifications in a way accessible to the domain experts. This includes better syntactic support for domain specific languages (using the Brady and Hammond [2012] as a starting point), and better support for eliding information, extending the mechanism for hidden arguments in Agda.

3.3 Organisation

The project is led by Patrik Jansson in the Functional Programming group at Chalmers. The work will be carried out by Jansson (20%), a PhD student (80%) and by C. Ionescu (Guest Researcher), and several MSc thesis students (not paid by the project). We will benefit from work on high-level modelling and scientific computing done at (and funded by) the Potsdam Institute for Climate Impact Research (N. Botta).

The first year of project is devoted to library support for Algebra of Programming (milestone **AoPP**) and the second year focus is on the **DSM** case study. From year two Irene Lobo Valbuena (currently a PhD student working on an EU-project, GRACeFUL, ending early 2018) will join.Irene will do her licentiate in the GRACeFUL project and finish her PhD in the LibTrust project working on the DSM case-study.

From year three onward we aim at merging the two tracks by applying the AoPP libraries to global systems models. We want the resulting software to scale up to enable running on HPC centres beyond the end of the project. Jansson and Ionescu will together supervise the new PhD student towards her PhD on “High-level Specifications and Correct Implementations via Dependent Types”.

To educate MSc and PhD students (in this and in other projects) we plan to organise a summer school in 2017 on the Algebra of Parallel Programming in Agda. We will also build up a github repository of library code (for specifications and implementations) produced in the project.

4 Significance

Effective production of correct software is a problem which remains unsolved, and is of great economic significance. By leveraging the capabilities of dependently-typed languages, this project aims to reduce the potential for errors by developing the specification of a system together with its implementation, and keeping them synchronised throughout the lifetime of the system.

Dependently-typed programming languages also allow us to formally encode and verify *program derivations*, which are important in minimising the number of "Eureka" steps needed to go from the specification to the implementation. This is even more so in the context of parallel programming, where the problems of sequential programming are compounded by the need to invent suitable operations and types for efficient parallelisation. The development of an algebra of parallel programming will be a significant advance in this area.

Software libraries have long been recognised as vehicles for increased software productivity. First, they capture domain knowledge in terms of software solutions to the problems that a user wants to solve. Second, they add a layer of abstraction to the underlying computation, which allows developers to write software in terms closer to their problem domain and usually results in improved quality and robustness. We aim to go beyond state-of-the-art when it comes to expressivity of libraries for programming with dependent types, and set new standards for the design of libraries in general.

The scientific contributions to the computer science area will be in the form of software prototypes (the libraries and other associated code will be available under an open licence), conference and journal papers and talks (on the techniques used to create the libraries as well as on the amendments made to the languages with dependent types), and doctoral training. We also hope to help the wider research community by contributing libraries for increasingly correct scientific computing.

As the example of parallel parsing shows, there are computations that can be expressed as instances of (abstract) linear algebra algorithms. At the same time, the calculational proofs that we have given for the correctness of these algorithms improve on the classical informal proofs. The connection between computing science and linear algebra has been noticed several times in the literature, but it has until now been exploited mainly in one direction: finding new applications of linear algebra. We aim to explore the connection also in the other direction: using recursive types and calculational proofs to simplify presentations of classical linear algebra notions and results. This will be a natural part of our more general efforts (beyond this project) to specify and implement validated numerical methods.

5 Preliminary findings

We have published results showing relevant related experience in all the suggested iteration phases and application areas as indicated below.

5.1 The three phases of the iteration

Proven-correct applications: We have worked on correct applications in Haskell [Danielsson and Jansson, 2004; Jansson and Jeuring, 2002] and supporting theory [Danielsson et al., 2006]. We have also worked on applications to climate impact research and economic modelling directly in Agda: [Ionescu and Jansson, 2013a,b]. More recent work (in submission) includes a formalisation of Sequential Decision Problems [Botta et al., 2016b] and follow-up work on a computational theory of policy advice and avoidability [Botta et al., 2016a].

Patterns into libraries: We have developed, implemented and compared libraries of generic functions [Jansson and Jeuring, 1998a,b; Norell and Jansson, 2004; Rodriguez et al., 2008]. Most of this has been done in Haskell, but it has become clear that the natural setting for generic programming is dependent types [Benke et al., 2003]. We have also worked on libraries for parsing [Bernardy and Jansson, 2015; Duregård and Jansson, 2011], testing [Duregård et al., 2012; Jeuring et al., 2012] and several DSLs for applied mathematics and calculus [Ionescu and Jansson, 2013a,b, 2015]

Refinement of programming languages: We have designed a generic programming language extension (PolyP [Jansson and Jeuring, 1997]) for Haskell, and we have been involved in the design of the Agda language [Norell, 2007]. We are active in the development of Agda: from the development of parametricity theory [Bernardy et al., 2012], a new kind of generic programming, based on a generalisation of erasure, is being developed by our collaborators at Chalmers. We have also contributed to the development of the "Concepts" feature of C++ by an extensive comparison to Haskell's type classes [Bernardy et al., 2010].

5.2 The application areas

AoPP: In [Mu et al., 2009] we presented a library for Bird and de Moor–style program derivation in Agda. Based on this work, a similar library has been implemented in Idris by David Christiansen.

Our group has developed an efficient sparse matrix based algorithm for parallel parsing [Bernardy and Claessen, 2013]. As a follow-up, we have recently shown [Bernardy and Jansson, 2015] that it is possible to *calculate* the efficient parallel algorithm from a specification of parsing, thus illustrating the promise of an algebra of parallel programming.

DSM: We have used dependent types in order to express high-level specifications of software components used in computational assessments of vulnerability to climate change [Ionescu, 2016; Ionescu and Jansson, 2013b]. We have been able to prove the correctness of some of these components (and explain why others were incorrect), and, perhaps more importantly, we have been able to clarify some of the terminological confusion existing in the field.

In [Ionescu and Jansson, 2013a], we have also used type theory to specify the basic building blocks of economic theory, used in almost all economic models today, concepts such as Pareto efficiency, Walrasian equilibrium, Nash equilibrium, etc., together with the relations between them (for example, Walrasian equilibria are Pareto efficient). Recently, we have developed specifications and correct-by-construction implementations for a large class of sequential decision problems [Botta et al., 2013, 2016a,b].

6 International and national collaboration

With this project, we believe we are in an ideal situation for collaboration, as we have contacts both upstream with the implementers of dependently-typed languages, and downstream with end-users of frameworks for formal modelling and implementation.

On the upstream side, we are in direct contact with the group in charge of the development of Agda: Two of the main developers, Norell and Danielsson, were Jansson's students; and Agda Implementers' Meetings are held yearly at Chalmers. These meetings attract participants from research groups in Nottingham, Copenhagen, Munich, and Japan. We have also close contacts with the programming-logic group at Univ. of Gothenburg, which deals with the fundamental aspects of type theory (T. Coquand).

Downstream, we have contacts with domain experts (N. Botta, J. Heitzig) in Potsdam, who need tools to describe models of various dynamical systems (such as the atmosphere or the economy) in formal ways, as well as efficient implementations of these models. Since political decisions may depend on the outcome of their simulations, correctly implementing these models is important.

Based on our experience with functional programming and domain specific modelling we have acquired funding for the project GRACeFUL from the FETPROACT-1-2014 Global Systems Science call in Horizon 2020. The project started early 2015 and can complement the current application, especially the case study on Domain Specific Modelling. More recently we have started working in the "Centre of excellence for Global Systems Science (CoeGSS)", also with Horizon 2020 funding. Through CoeGSS we have access to two super-computing centres (HLRS in Stuttgart and PSNC in Poznań) and several sites with domain experts in Global Systems Science.

References

Benke, M., Dybjer, P., and Jansson, P. (2003). Universes for generic programs and proofs in dependent type theory. *Nordic J. of Computing*, 10(4):265–289. `http://dl.acm.org/citation.cfm?id=985801`.

Bernardy, J.-P. and Claessen, K. (2013). Efficient divide-and-conquer parsing of practical context-free languages. In *Proc. of ICFP 2013*, pages 111–122.

Bernardy, J.-P. and Jansson, P. (2015). Certified context-free parsing. *Logical Methods in Computer Science*. Accepted 2015-12-22, available at arxiv.org/abs/1601.07724.

Bernardy, J.-P., Jansson, P., and Paterson, R. (2012). Proofs for free — parametricity for dependent types. *J. of Functional Programming*, 22(02):107–152.

Bernardy, J.-P., Jansson, P., Zalewski, M., and Schupp, S. (2010). Generic programming with C++ concepts and Haskell type classes—a comparison. *J. Funct. Program.*, 20(3–4):271–302.

Bird, R. and de Moor, O. (1997). *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice-Hall International.

Botta, N., Ionescu, C., and Brady, E. (2013). Sequential decision problems, dependently typed solutions. In *Proc. Conf. on Intelligent Computer Mathematics (CICM 2013)*.

Botta, N., Jansson, P., and Ionescu, C. (2016a). Contributions to a computational theory of policy advice and avoidability. Code and pre-print available from github.

Botta, N., Jansson, P., and Ionescu, C. (2016b). Sequential decision problems, dependent types and generic solutions. Code and pre-print available on github.

Brady, E. and Hammond, K. (2012). Resource-safe systems programming with embedded domain specific languages. In *Practical Aspects of Declarative Languages*, pages 242–257. Springer.

Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., and Wisnesky, R. (2009). Effective interactive proofs for higher-order imperative programs. In *Proc. of ICFP 2009*, ICFP '09, pages 79–90. ACM.

Danielsson, N. A. (2010). Total parser combinators. In *Proc. of ICFP 2010*, ICFP '10, pages 285–296. ACM.

Danielsson, N. A., Hughes, J., Jansson, P., and Gibbons, J. (2006). Fast and loose reasoning is morally correct. In *POPL'06*, pages 206–217. ACM Press.

Danielsson, N. A. and Jansson, P. (2004). Chasing bottoms, a case study in program verification in the presence of partial and infinite values. In *MPC 2004*, volume 3125 of *LNCS*, pages 85–109. Springer.

Duregård, J. and Jansson, P. (2011). Embedded parser generators. In *Haskell '11*, pages 107–117, New York, NY, USA. ACM.

Duregård, J., Jansson, P., and Wang, M. (2012). Feat: Functional enumeration of algebraic types. In *Haskell'12*, pages 61–72. ACM.

Gonthier, G. (2009). Ssreflect: Structured scripting for higher-order theorem proving. In *PLMMS'09*, page 1. ACM.

Ionescu, C. (2009). *Vulnerability modelling and monadic dynamical systems*. PhD thesis, Freie Universität Berlin.

Ionescu, C. (2016). Vulnerability modelling with functional programming and dependent types. *Mathematical Structures in Computer Science*, 26(01):114–128.

Ionescu, C. and Jansson, P. (2013a). Dependently-typed programming in scientific computing: Examples from economic modelling. In Hinze, R., editor, *24th Symposium on Implementation and Application of Functional Languages (IFL 2012)*, volume 8241 of *LNCS*, pages 140–156. Springer-Verlag.

Ionescu, C. and Jansson, P. (2013b). Testing versus proving in climate impact research. In *Proc. TYPES 2011*, volume 19 of *Leibniz Int. Proc. in Informatics (LIPIcs)*, pages 41–54, Dagstuhl, Germany.

Ionescu, C. and Jansson, P. (2015). Domain-specific languages of mathematics: Presenting mathematical analysis using functional programming. In *Proc. 4th Int. Workshop on Trends in Functional Programming in Education*, EPTCS. Open Publishing Association.

Jansson, P. and Jeuring, J. (1997). PolyP — a polytypic programming language extension. In *Proc. POPL'97: Principles of Programming Languages*, pages 470–482. ACM Press.

Jansson, P. and Jeuring, J. (1998a). Functional pearl: Polytypic unification. *J. Funct. Program.*, 8(5):527–536.

Jansson, P. and Jeuring, J. (1998b). PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand. Available from `www.cse.chalmers.se/~patrikj/poly/polylib/`.

Jansson, P. and Jeuring, J. (2002). Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75.

Jeuring, J., Jansson, P., and Amaral, C. (2012). Testing type class laws. In *Haskell'12*, pages 49–60. ACM.

Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115.

Lincke, D., Jansson, P., Zalewski, M., and Ionescu, C. (2009). Generic libraries in C++ with concepts from high-level domain descriptions in Haskell. In *IFIP Working Conference on Domain Specific Languages*, volume 5658/2009 of *LNCS*, pages 236–261.

Martin-Löf, P. (1984). *Intuitionistic type theory*. Bibliopolis.

Mu, S.-C., Ko, H.-S., and Jansson, P. (2009). Algebra of programming in Agda: dependent types for relational program derivation. *J. Funct. Program.*, 19:545–579.

Norell, U. (2007). *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers Tekniska Högskola.

Norell, U. and Jansson, P. (2004). Polytypic programming in Haskell. In *Implementation of Functional Languages 2003*, volume 3145 of *LNCS*, pages 168–184. Springer.

Oury, N. and Swierstra, W. (2008). The power of Pi. In *Proc. of ICFP 2008*, pages 39–50. ACM.

Pouillard, N. and Gustafsson, D. (2015). crypto-agda: Cryptographic constructions in the type theory of agda. Available on github: `https://github.com/crypto-agda`. Ongoing work by `www.DemTech.dk`.

Rodriguez, A., Jeuring, J., Jansson, P., Gerdes, A., Kiselyov, O., and d. S. Oliveira, B. C. (2008). Comparing libraries for generic programming in Haskell. In *Haskell'08*, pages 111–122. ACM.

Swamy, N., Chen, J., Fournet, C., Strub, P.-Y., Bhargavan, K., and Yang, J. (2011). Secure distributed programming with value-dependent types. In *Proc. of ICFP 2011*, pages 266–278.

The Coq development team (2010). The Coq proof assistant.

Valiant, L. (1975). General context-free recognition in less than cubic time. *J. of computer and system sciences*, 10(2):308–314.