Strongly Typed Generic Libraries

Patrik Jansson

1 Main objectives

Our long term goal is to create systems (theories, programming languages, libraries and tools) which make it easy to develop reusable software components with matching specifications. The main goal of this research project is to improve the understanding, implementation and theory of strongly typed, generic programming. We work towards two sub-goals:

- **Efficient generic libraries**: benchmarking and efficient compilation
- **Correct generic libraries**: specification-driven development and parametricity

2 Research area overview

An important part of computer science research is about developing systems supporting the construction of correct and reusable software. This project is about systems which make it possible to develop programs which work uniformly over many different datastructures: generic programs. Since generic programs work for many different types, such as lists, database tables, and family trees, they can be used and reused as building blocks in all kinds of software development.

A simple example is a program which counts the number of elements in a datastructure. Without generic programming we have to use one version for each structure (tables, search trees, matrices, etc.) while a generic program directly works for all these.

Generic programming offers a number of benefits:

Reusability: Generics extend the power of polymorphism to allow classes of related algorithms to be described in one definition. Thus generic programs are very well suited for building program libraries — a very successful form of software reuse.

Adaptivity: Generic programs automatically adapt to changing datatypes. For example, even after adding or removing a constructor of a datatype, the same generic algorithm can still be applied. This adaptivity reduces the need for time consuming and boring rewrites of trivial methods and reduces the associated risk of making mistakes.

Applications: Some algorithms are naturally generic — to traverse a datastructure and perform the same transformation on all parts, to print, parse, compress or decompress datastructures, to compare two structures for equality, etc. Each of these can be described informally in a datatype-independent way, and with generic programming this informal description can be turned into a formal definition (program code).

Specifications: More general algorithms mean more general specifications. If we consider generic specifications, then each of the earlier benefits obtains an additional interpretation: we get reusable specifications, adaptive specifications, and if one can verify that a generic application conforms to its specification, then the resulting evidence holds for many instances.

Some international projects and research groups have worked or work on generic programming under different names:

- *Standard Template Library* (STL) is a generic software library for C++ containing containers, iterators, algorithms, and functors. Two influential projects building on STL are Boost [Boost] and Adobe Source Libraries [Parent and Marcus].

- *C++ Concepts* [Gregor et al., 2006] provide a way to express the syntactic and semantic behaviour of types and to constrain the type parameters in a C++ template. Language support for concepts have been recently voted out from the upcoming version of C++, but with a clear intention of reconsidering the future in only a few years. Our detailed comparison of concepts with type classes [J6, J14] is a useful tool in the future discussion.

- *Datatype-Generic Programming* (DGP) [Backhouse, 2006, Gibbons, 2007] in Oxford and in Nottingham. This work grew out of the earlier work on *Algebra of Programming* [Bird and de Moor, 1997]. We have collaborated closely with this group for many years [J2, J8, J17][1].

- The *Generic Haskell* project in Utrecht led by Johan Jeuring extended the purely functional language Haskell [Peyton Jones and Hughes, 1999] with *type indexed functions* [Hinze, 2000]. This project started from the applicant's (well cited) earlier work on *Polytypic Programming* [J1, J2, J3].

- The work on *Intensional Type Analysis* [Harper and Morrisett, 1995] applies generic programming to compilation of functional languages. They aim at efficient code by specialisation and later work has applied generic programming ideas to obtain type-safe run-time generic programming [Weirich, 2006].

[1] We use [Jnn] to cite papers in Jansson's publication list (appendix C) to avoid duplication.

3 Project description and preliminary findings

In the project we will work towards the goals of efficient and correct generic libraries. For each goal we present both the proposed work and our preliminary findings.

3.1 Efficient generic libraries

Currently, among main-stream languages, Haskell has the best support for generic programming [Garcia et al., 2007] while C++ has the most efficient generic libraries.

We will set up an open benchmark suite of generic algorithms with realistic test data and we will use this to investigate what can be improved to obtain better run-time behaviour. We will draw examples and benchmark code both from our generic programming expressivity benchmark suite [J7, J13] and from generic libraries for modelling complex systems [J12] (in collaboration with the Potsdam Institute for Climate Impact Research—PIK). An important aspect to explore is to what extent we can make use of concurrency in general and multi-cores in particular.

Our recent work on expressivity benchmarks for generic libraries in Haskell [J7, J13] includes a small performance analysis where the big run-time differences between the libraries indicate a large potential for improvement. No library is uniformly most efficient over the different tests and the overhead compared to hand-specialised code ranges from none to over 50 times! Preliminary investigation indicates that the main sources of inefficiency are unnecessary data conversion and the run-time passing of dictionaries (representing types the generic function should have been specialised to). Recent work by our colleagues in Utrecht [Magalhães et al., 2010] show promising results which we aim to build on.

Applying partial evaluation, deforestation and specialisation techniques to the compilation of generic programs should result in optimised code as good as (or better than) hand-written instances. Combining the expertise in generic programming (Jansson) with Chalmers' strength in functional programming and partial evaluation we are confident that we can make rapid progress. Concretely, there is existing infrastructure for rewrite rules [Chakravarty and Keller, 2001] and specialisation in the Glasgow Haskell compiler which provides a good starting-point.

We will explore how type classes (concepts) with associated types (a technique used in many efficient generic C++ libraries) can be used in Haskell and Agda [Norell, 2007, Norell et al., 2008]. The dependently typed language Agda (developed at Chalmers) is the natural setting for generic programming, but the full expression of generic programming in Agda has only just started. We believe that the stronger type system guarantees of pure functional languages can make the library code both safer and more efficient. Positive examples include the tag-free interpreters made possible by dependent types (available as GADTs in Haskell and natively in Agda), and in general stronger types means more possibilities for powerful compiler optimisations [Brady, 2005, Nilsson, 2005, Xi and Pfenning, 1999]. We believe there is a close correspondence between a tag-free interpreter and a properly

specialised generic library, and we hope to use this as one way to obtain efficient generic libraries.

Jansson has been doing research on generic programming since 1995 with different collaborators in a strong international network. Based on the long experience with generic programming in Haskell [J1, J4, J10, J16, J19, J20, J21] and the more recent investigations of concepts in Haskell and C++ [J6, J12, J13, J14] we have the right background for practical and theoretical work on efficient generic libraries. We will also build on theoretical results about generic programs and proofs [J8, J9, J15, J17].

3.2 Correct generic libraries

For many practical applications software libraries need to be efficient, but speed does not help if the computed result is wrong! We will explore different techniques (QuickCheck and Agda) to obtain correct generic libraries. QuickCheck [Claessen and Hughes, 2000] is a successful concept (developed in the FP group at Chalmers) for specification and automatic random testing of program code, but it is not directly applicable to generic programs. The language Agda [J22] supports the Curry-Howard isomorphism between programs + types on the one hand, and specifications + proofs on the other hand. Agda thus works as a unifying language supporting specification-driven development of programs, properties and correctness proofs.

To use random testing for generic algorithms, we need generic executable specifications and generic generators of input data. We will develop executable specifications and generators for the algorithms in the benchmark suite mentioned above and we will use them to search for (and correct) bugs. Preliminary results on generic testing [J16] are promising, but have shown that there is much left to do. We have recently started to use parametricity results to pick the right types for testing generic functions [J11] and we will work on extending and implementing this method to improve the data generators (to obtain good test coverage from fewer tests).

Parametricity Wadler [1989] and in general Reynolds' abstraction theorem [Reynolds, 1983] shows how a typing judgement in System F can be translated into a theorem about functions of that type. This is a very powerful method of obtaining useful correctness guarantees for typed functional programs. But many generic programs need dependent types, which are not available in System F, so the parametricity results need to be strengthened. In this project we will strengthen parametricity and the abstraction theorem to work for languages with dependent types. We will then apply these results to generic libraries in Haskell and Agda. Two typical examples of generic functions that we tackle are:

catamorphism $\mathit{fold} : ((F, \mathit{map}) : \mathit{Functor}) \to (t : \star) \to (F\ t \to t) \to \mu\, F \to t$, which is a generic evaluator function defined within a dependently-typed language.

generic cast $\mathit{gcast} : (F : \star \to \star) \to (u\ t : U) \to \mathit{Maybe}\,(F\,(\mathit{El}\ u) \to F\,(\mathit{El}\ t))$, which comes from a modelling of representation types with universes.

In both cases, the derived parametricity condition yields useful properties to reason about the correctness of the function (or, in fact, any function of the same type).

Some generic algorithms are inherently difficult to test due to the rich structure of the parameters and a promising approach is to instead express the specification and the algorithm in the same logical framework and use computer-aided verification. We will investigate how the specification language (QuickCheck) can be adapted to allow for two back-ends: testing and proving. We will use Agda as the proof technology infrastructure keeping track of the top level correctness.

Jansson has worked on specification [Jansson and Jeuring, 1998] & [J4, J5, J18], testing [J14, J19] and correctness proofs [J10, J12, J13] of generic libraries. We will benefit from techniques and tools developed in the Cover project "Combining Verification Methods in Software Development" at Chalmers (funded by the Swedish Foundation for Strategic Research (SSF), 2002–2005). The Cover tool set is based on Haskell and consists of QuickCheck [Claessen and Hughes, 2000], Agda [Norell et al., 2008] and CoverTranslator, a prototype translator from Haskell to First Order Logic.

We have already extended Reynolds abstraction theorem and parametricity results to work for pure type systems (paper in submission) and within this project we will explore, extend and apply this result to correctness of generic libraries.

4 Project plan and funding

The project is led by Patrik Jansson in the Functional Programming (FP) group of the Computer Science and Engineering (CSE) department at Chalmers. The work will be carried out by Jansson (30%), Jean-Philippe Bernardy (now a 3rd year PhD student, not paid by the project), one new PhD student (80%), and several MSc thesis students (not paid by the project). We apply for 80% of the total project cost from VR, the rest is covered by Chalmers. We will benefit from work on generic libraries and high-level modelling done at (and funded by) PIK (Daniel Lincke, Cezar Ionescu). Jansson is co-applicant on a multi-project grant about Programming with Dependent Types (PI is T. Coquand) which would complement this project well.

For the first two years of the project Jansson will start up and supervise the new PhD student towards the Efficient Generic Libraries goal, while Jansson & Bernardy will work on the Correct Generic Libraries goal (and in the end, Bernardy gets his PhD). In the last two years, the two goals converge in using strong types for efficiency and correctness, and Jansson supervises the new PhD student towards her PhD on "Strongly Typed Generic Libraries". We work towards the following milestones:

Y1–2 Efficient Setting up the efficiency benchmark suite (in collaboration with Utrech and PIK) and evaluating the efficiency of the compiled code in different scenarios.

Y1–2 Correct Develop the parametricity results for dependent types and implement a prototype tool based on Agda (in collaboration with the programming logic group).

Y3–4 Efficient Partial evaluation and optimisations based on strong types means selected generic libraries are now as efficient as hand-written code. Exploring parallellisation possibilities.

Y3–4 Correct Develop specifications and automatic random testing for generic algorithms and datastructures (case study on graph algorithms). Specifications and proofs for higher order constructions like monads, applicative functors and catamorphisms.

Jansson and Bernardy are partially funded (20%) by J. Hughes' "Software Design and Verification using Domain Specific Languages" (VR, multi-project grant in ICT, 2009–2012). Hughes' project applies functional programming techniques, especially DSLs embedded in Haskell and Erlang, to the design and verification of complex software, taking motivating examples from the telecom domain. The current project proposal, on the other hand, explores strongly typed generic libraries in general and compilation techniques and parametricity in particular.

5 Collaboration

The local research environment within the CSE department is excellent—we collaborate with several world class researchers in areas related to this project: Functional programming and automatic testing (J. Hughes, K. Claessen); Domain Specific Languages (J. Hughes, M. Sheeran); programming logic (T. Coquand, P. Dybjer); language technology (A. Ranta, B. Nordström); formal methods in software engineering (R. Hähnle). The proposed project will benefit from other already awarded related grants involving the CSE department: J. Hughes' ProTest: Property-based Testing (FP7-ICT-2007.1.2, started 2008, `www.protest-project.eu`), R. Hähnle's HATS: Highly Adaptable and Trustworthy Software using Formal Methods (FP7-ICT-2007-3, started 2009). Th. Coquand's Adv. Investigator's grant and his EU project "ForMath: Formalization of Mathematics".

The international contacts most relevant for this project are: J. Jeuring (Utrecht Univ., NL), S. Schupp (TU Hamburg, DE), R. Backhouse (Univ. of Nottingham, UK), J. Gibbons (Oxford Univ., UK), S.-C. Mu (Academia Sinica, Taiwan), C. Ionescu (Potsdam Institute for Climate Impact Research, DE).

We feel that even in this world of reliable and inexpensive means of long distance electronic communication, personal meetings between researchers are still very important for the advancement of the field. We have had very good experience from my research visits to different departments and from the visits of other researchers to Chalmers. We therefore apply for money so that we can travel to meet researchers at other sites *and* for inviting other researchers to visit us to do collaborative work.

6 Importance

In broad terms, our research centres around technologies that support the development of long-lasting software systems and their safe usage by end users. One particularly interesting question is how such support can be provided in the presence of changes, customisation, and software evolution, and what technologies and theories thus need to be devised. Analyses and tools that can continue to work robustly and efficiently in the presence of change typically require some collaboration on part of the software systems themselves. Many successful approaches are therefore naturally related to the design of generic software and software libraries, where extensibility is a major design force.

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 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. In the last fifteen years, generic libraries which introduce the aspects of reusability and strict performance guarantees to library design have gained attention [Musser and Stepanov, 1994].

Generic programming has become most used in the programming language C++, where now most cutting-edge development takes place in the form of libraries and where the influential Boost organisation for library standardisation controls overall software quality [Boost]. Key for the success of libraries in C++ are the compile-time features of the language ("templates"), which allow, at least in theory, eliminating the overhead that otherwise comes along with high-level abstractions [Gregor et al., 2005]. Templates are powerful, but unrestricted use often results in very complex error messages or (worse) silently accepted incorrect behaviour. Language support for concepts in C++ has been proposed to resolve these problems and we have been active in evaluating this proposal.

Generic programming today will shape "normal programming" in the future. We can already see this in modern functional programming languages like Haskell, OCaml and Erlang. What C++ calls concepts are basically the same [J6, J14] as Haskell's type classes, which have been used for generic programming for at least 15 years. An increasing number of companies, large and small, are using functional languages to gain competitive advantage. Many of these companies now need to expand, creating opportunities for skilled functional programmers to work with this exciting technology. Here in Gothenburg, both Chalmers and the IT faculty have taught functional languages for many years, and there is a pool of available talent that is attractive to commercial users. The FP group arranged the very successful first "Jobs in Functional Programming" event in 2007 and we have good contacts with industrial partners interested in technology transfer.

Both C++ and Haskell are in the process of working out the next language revision (C++0x and Haskell-prime) and we hope to affect relevant design decisions in the generic programming area. We are also in a position to actively take part in the evolution of Agda towards a next-generation language for efficient and correct generic libraries.

To summarise—the impact of "Strongly Typed Generic Libraries" outside the academic world will be through companies and researchers benefiting from efficient generic libraries, users and programmers obtaining correct generic libraries, and students and software engineers learning to use new language features in main-stream languages.

The scientific contributions to the computer science area will be in the form of software prototypes (the benchmark suite and associated code), conference talks/papers (on compilation techniques and library correctness), journal papers and doctoral training. We aim to go beyond state-of-the-art when it comes to expressivity, efficiency and correctness of generic programming and we hope to improve the software technology field in general.

References

R. Backhouse. Datatype-generic reasoning. In A. Beckmann et al., editors, *Logical Approaches to Computational Barriers*, volume 3988 of *LNCS*, pages 21–34. Springer-Verlag, 2006.

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

Boost. The Boost initiative for free peer-reviewed portable C++ source libraries. `http://www.boost.org`, 2009.

E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, University of Durham, 2005.

M. M. T. Chakravarty and G. Keller. Functional array fusion. In *ICFP'01: Int. Conf. on Functional Programming*, pages 205–216, 2001.

K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP'00: Int. Conf. on Functional Programming*, pages 268–279. ACM, 2000.

R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. An extended comparative study of language support for generic programming. *J. Funct. Program.*, 17(2):145–205, 2007.

J. Gibbons. Datatype-generic programming. In R. Backhouse et al., editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *LNCS*. Springer-Verlag, 2007.

D. Gregor, J. Järvi, M. Kulkarni, A. Lumsdaine, D. Musser, and S. Schupp. Generic programming and high-performance libraries. *International J. of Parallel Programming*, 33(2):145–164, 2005.

D. Gregor, J. Järvi, J. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: Linguistic support for generic programming in C++. In *OOPSLA: Object-Oriented Programming Systems, Languages, and Applications*, pages 291–310. ACM Press, 2006.

R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *POPL'95: Principles of Programming Languages*, pages 130–141, 1995.

R. Hinze. A new approach to generic functional programming. In *POPL'00: Principles of Programming Languages*, pages 119–132. ACM Press, 2000.

P. Jansson. The WWW home page for polytypic programming. Available from `http://www.cse.chalmers.se/~patrikj/poly/`, 2003.

P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998. Available from the Polytypic prog. page [Jansson, 2003].

J. P. Magalhães, S. Holdermans, J. Jeuring, and A. Löh. Optimizing generics is easy! In *PEPM '10: Proc. ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 33–42. ACM, 2010.

D. Musser and A. Stepanov. Algorithm-oriented generic libraries. *Software—Practice and Experience*, 27(7):623–642, Jul 1994.

H. Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *ICFP'05*, pages 54–65. ACM Press, 2005.

U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, 2007.

U. Norell et al. Agda — a dependently typed programming language. Implementation available from Google Code: `http://code.google.com/p/agda/`, 2008.

S. Parent and M. Marcus. Adobe source libraries (ASL). `http://stlab.adobe.com/`.

S. Peyton Jones and J. Hughes, editors. *Haskell 98 — A Non-strict, Purely Functional Language.* Available from `http://www.haskell.org/definition/`, Feb. 1999.

J. C. Reynolds. Types, abstraction and parametric polymorphism. *Information processing*, 83(1): 513–523, 1983.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, FPCA '89*, pages 347–359. ACM Press, 1989.

S. Weirich. Type-safe run-time polytypic programming. *J. Funct. Program.*, 16(6):681–710, 2006.

H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99*, pages 214–227. ACM, 1999.