VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kansliets noteringar Kod	Dnr
2011-2993-88525-26	

2011
Project Research Grant

Area of science
Natural and Engineering Sciences

Announced grants
Project research grant NT 13 April 2011

Total amount for which applied (kSEK)

2012	2013	2014	2015	2016
1046	1081	1114	1152	

APPLICANT

Name(Last name, First name)
Jansson, Patrik

Date of birth
720311-7515

Gender
Male

Email address
patrikj@chalmers.se

Academic title
Associate professor

Position
Docenttjänst

Phone
031-7725415

Doctoral degree awarded (yyyy-mm-dd)
2000-06-09

WORKING ADDRESS

University/corresponding, Department, Section/Unit, Address, etc.
Chalmers tekniska högskola
Department of Computer Science and Engineering
Programvaruteknik
Rännvägen 6b
41296 Göteborg, Sweden

ADMINISTERING ORGANISATION

Administering Organisation
Chalmers tekniska högskola

DESCRIPTIVE DATA

Project title, Swedish (max 200 char)
Starkt typade bibliotek för program och bevis

Project title, English (max 200 char)
Strongly Typed Libraries for Programs and Proofs

Abstract (max 1500 char)
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. In this research project, the main focus is on libraries. Strongly-typed programming languages allow to express functional specifications as types. Checking the types of a program then means checking it against its specification. Within such powerful programming languages, libraries are not only building blocks of programs, but also of proofs. We believe that such libraries will eventually become the main means of developing programs, and because they come with strong types, the programs built using the library will come with strong properties that will make the whole easy to prove correct. The production of such libraries will also inform the design of future strongly-typed programming languages. In the recent years, strongly-typed programming languages have started to become usable, but remain confined to a small niche. Our libraries will make them a viable solution for a broader range of applications, bringing higher guarantees of correctness to a wider user base. To check the applicability of our libraries, we will apply them to classical problems of computer programming, such as certain divide-and-conquer algorithms or optimisation problems, as well as to the construction of tools supporting dependently-typed programming themselves.

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2011-2993-88525-26
Name of Applicant
Jansson, Patrik
Date of birth
720311-7515

Abstract language
English
Keywords
Software Technology, Functional Programming, Dependent Types, Program Verification, Generic Programming
Research areas
Computer Science
Review panel
NT-S, NT-R
Classification codes (SCB) in order of priority
10201, 10205, 10103
Aspects

Continuation grant
Application concerns: New grant
Registration Number:
Application is also submitted to

similar to:	**identical to:**

ANIMAL STUDIES

Animal studies
No animal experiments

OTHER CO-WORKER

Name(Last name, First name)	**University/corresponding, Department, Section/Unit, Addressetc.**
,	
Date of birth	**Gender**
Academic title	**Doctoral degree awarded (yyyy-mm-dd)**

Name(Last name, First name)	**University/corresponding, Department, Section/Unit, Addressetc.**
,	
Date of birth	**Gender**
Academic title	**Doctoral degree awarded (yyyy-mm-dd)**

Name(Last name, First name)	**University/corresponding, Department, Section/Unit, Addressetc.**
,	
Date of birth	**Gender**
Academic title	**Doctoral degree awarded (yyyy-mm-dd)**

Name(Last name, First name)	**University/corresponding, Department, Section/Unit, Addressetc.**
,	
Date of birth	**Gender**
Academic title	**Doctoral degree awarded (yyyy-mm-dd)**

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2011-2993-88525-26

Name of Applicant
Jansson, Patrik

Date of birth
720311-7515

ENCLOSED APPENDICES

A, B, C, S

APPLIED FUNDING: THIS APPLICATION

Funding period (planned start and end date)
2012-01-01 -- 2015-12-31

Staff/ salaries (kSEK)

Main applicant	% of full time in the project	2012	2013	2014	2015	2016
Patrik Jansson	20	268	277	286	296	
Other staff						
Jean-Philippe Bernardy (PostDoc)	65	646	668	690	714	
Total, salaries (kSEK):		914	945	976	1010	

	2012	2013	2014	2015	2016
Travel, offices, IT	132	136	138	142	
Total, other costs (kSEK):	132	136	138	142	

Total amount for which applied (kSEK)

2012	2013	2014	2015	2016
1046	1081	1114	1152	

ALL FUNDING

Other VR-projects (granted and applied) by the applicant and co-workers, if applic. (kSEK)

Proj.no.(M) or reg.nr.	Funded 2011	Funded 2012	Applied 2012
2009-4303	1857	1857	

Project title
Software Design and Verification using Domain Specific Languages: Putting Functional Programming to Work

Applicant
John Hughes

Funds received by the applicant from other funding sources, incl ALF-grant (kSEK)

Funding source	Total	Proj.period	Applied 2012
EU, FP7	11700	2010-2012	

Project title
Global System Dynamics and Policy

Applicant
Carlo Jaeger

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2011-2993-88525-26

Name of Applicant
Jansson, Patrik

Date of birth
720311-7515

POPULAR SCIENCE DESCRIPTION

Popularscience heading and description (max 4500 char)

En viktig gren av forskningen inom datavetenskap handlar om att utveckla system (programspråk, verktyg, programbibliotek, teorier) som gör det enkelt att konstruera korrekt och återanvändbar programvara. Detta projekt siktar på att utnyttja funktionella programspråk med starka typsystem till att skapa bibliotek av komponenter som kan uttrycka både specifikationer och implementationer som uppfyller dessa. Vi kommer att utnyttja datorstödd interaktiv programutveckling där automatiska verktyg ger snabb återkoppling på vilka delar som inte uppfyller specifikationen.

Den teoretiska möjligheten att uttrycka program och bevis i samma programspråk är känd sedan många år, men det är bara nyligen som teknikutvecklingen har medgett att utveckla större programbibliotek på detta sätt. Detta innebär att det finns många spännande grundläggande frågor kvar att utforska och vi avser börja med enkla algoritmer för att sedan steg för steg utforska hur långt det går att komma. Vi arbetar iterativt i tre nivåer för att utveckla komponentbiblioteken. Första nivån är att implementera en lösning på ett visst problem (sökning, optimering eller liknande), nästa nivå är att abstrahera ut gemensamma mönster till programbibliotek och slutligen vill vi utvärdera vilka möjliga förändringar av den underliggande språket som skulle kunna förbättra resultaten. Inom projektet kommer vi att arbeta fram korrekta generiska bibliotek uttryckta i språket Agda. Agda är ett verktyg baserat på typteori och funktionell programmering som möjliggör utveckling av program och specifikationer i samma språk. Utvecklingen av språket har skett (och forskrider parallellt med biblioteksprojektet) i ett internationellt samarbete (med Japan, Tyskland och England) lett av Chalmers.

På lång sikt kan bevisbart korrekta programbibliotek användas och återanvändas som byggstenar vid all slags programvarukonstruktion. Detta ger allmänt sett mer pålitliga program, och färre buggar. Ett spännande applikationsområde är exekverbara, överblickbara högnivåmodeller för komplexa system. Vi har hittills mest fokuserat på att modellera komplexa system inom dataområdet (logiska ramverk, lingvistik, programspråk, hårdvara) men i samarbetet med Potsdams Institut för Klimatforskning (PIK) har vi börjat arbeta med komplexa system i interaktionen mellan klimat, ekonomi och samhälle. PIK har under flera år arbetat med simuleringar av komplexa system och har under senare år börjat använda funktionell programmering som ett verktyg för att experimentera med och kommunicera de högnivåmodeller som behövs för att överblicka komplexa system. Dessa högnivåmodeller översätts senare i flera steg till effektiv programkod som klarar att köra tunga simuleringar inom rimlig tid. (Dessa simuleringar ger underlag till politiska beslut inom klimatområdet.) PIK tog kontakt med Chalmers för att fördjupa sin kompetens inom högnivåmodellering med hjälp av moderna programspråk (som Haskell och C++) och vi har under åren som gått haft flera kontakter där starkt typade bibliotek för program och bevis har utkristalliserats som det forskningsområde där Chalmers bäst kan komplettera PIK.

På Chalmers leds projektet av Patrik Jansson (inom gruppen Funktionell Programmering). Jansson har forskat om generisk programmering sedan 1995 i olika konstellationer och det internationella kontaktnätet är mycket starkt. Den lokala forskningsmiljön inom D&IT-institutionen är världsledande även inom flera närliggande områden - automatisk testning (Hughes, Claessen), domänspecifika språk (Sheeran, Claessen), typteori (Coquand), språkteknologi (Ranta).

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2011-2993-88525-26

Name of applicant
Jansson, Patrik

Date of birth
720311-7515

Title of research programme
Strongly Typed Libraries for Programs and Proofs

Appendix A

Research programme

Vetenskapsrådet, Box 1035, SE-101 38 Stockholm, tel. +46 (0)8 546 44 000, vetenskapsradet@vr.se

Strongly Typed Libraries for Programs and Proofs

Patrik Jansson and Jean-Philippe Bernardy

1 Main objectives

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 three areas: simple divide-and-conquer algorithms, optimisation problems (inspired by the Algebra of Programming [Bird and de Moor, 1997]), and self-application: applying our results to parts of the implementation of the programming environment itself.

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 modern languages such as Java, C++, Scheme and Haskell are actively gaining abstraction power with Java Generics [Bracha et al., 1998], C++ Templates [Stepanov and Lee, 1995], Scheme's composable macros [Flatt, 2002] and Haskell meta-programming [Sheard and Peyton Jones, 2002]. But there is a danger lurking—more complex features can increase the risk of bugs and unintended behaviour. With new abstraction mechanisms we also need new computer-aided sanity checks of the program code.

Types. Types are used in many parts of computer science to keep track of different kinds of values and to keep 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 implementor 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 executions of the program, independently of its input.

By the Curry–Howard correspondence, type systems are isomorphic 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 recently gained popularity as a medium for programming. The flagship of dependently-typed programs is perhaps Compcert, a C compiler written and verified in Coq [Leroy, 2009]. Other applications are however rapidly appearing. Chlipala et al. [2009] show how to develop and verify imperative programs within Coq. Oury and Swierstra [2008] describe a library for database access which statically guarantees that queries are consistent with the schema of the underlying database. Morgenstern and Licata [2010] show how to do programming language based security in Agda.

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 as types, and proofs (for example using algebraic reasoning) in a single language (by taking advantage of the Curry–Howard isomorphism). Agda is currently emerging as a lingua-franca of programming with dependent types. Its canonical reference, Norell's Thesis [Norell, 2007], has been cited 50 times per year since its publication indicating strong academic interest. The focus of this project is on expressing libraries of correct programs and proofs in the dependently typed functional language of 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 is mostly applied to proofs rather than programs, because the Coq system is mostly intended as a proof assistant rather than a programming language. Even projects which aim to use Coq as a programming platform, such as [Chlipala et al., 2009, Leroy, 2009] retain this separation. The same observation applies to the libraries of most systems with dependent types. The Agda standard library (developed mainly by Danielsson), has evolved from common abstractions needed by Agda programmers. It has been applied to several domains, in particular parser combinators [Danielsson, 2010] and Algebra of Programming [Mu et al., 2009].

In the current Agda implementation, the portions of the library dedicated to programming are essentially decoupled from the portions dedicated to proofs. This can be a drawback: the structure of a proof often follows the same structure as the program it refers to, therefore keeping the two separated violates the principle of abstraction described above.

3 Project description

Our project will be organised in multiple iterations, each refining the libraries obtained during the previous one. (The first iteration will be based on our current experience with Agda, and its standard library.) 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 application domains. In this phase, we will assess the viability of our libraries by applying them to a particular application domain (see below for the chosen case studies on algebra of programming, optimisation and compilation).

2. **Extraction of common pattern 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 a pattern of implementation. We will then reimplement the application previously produced using the software components of 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 tool of the possible shortcomings we might identify, and participate in their remedy, if suitable.

We work iteratively towards the following milestones (case studies) ranging from classical problems of computer science to domain-specific applications:

AoP Develop libraries of programs and proofs for simple divide-and-conquer algorithms, like sorting and searching.

Optim Develop specifications and libraries for optimisation and dynamical systems (case study on economical and environmental models). In addition to domain knowledge (provided by our contacts in Potsdam), this requires specifications and proofs for higher order constructions like monads, functors and natural transformations.

Compile Work with the Agda team on implementation and specification of parts of Agda in Agda. This will both use the libraries developed in the project and improve the tool chain to the benefit of other users.

In the last section we show an example (chosen small enough to fit here) to illustrate how the three phases outlined above may be realized in practice. Note that it is done in the current Agda system, and that we expect to be able to improve on it.

4 Preliminary findings

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

4.1 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 are now ready to move from Haskell to Agda.

Patterns into libraries: We have developed, implemented and compared libraries of generic functions [Jansson and Jeuring, 1998a,b, Norell and Jansson, 2004b, 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.

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 have also contributed to the development of the "Concepts" feature of C++ by an extensive comparison to Haskell's type classes [Bernardy et al., 2010c].

4.2 The three application areas

AoP: We have worked actively on implementing programs and proofs in the Algebra of Programming tradition [Backhouse et al., 1999, Mu et al., 2008].

Optim: We also have recent experience in domain modelling in Haskell [Lincke et al., 2009] and optimisation modelling in Agda [Mu et al., 2009]. We are also working with PIK on Agda implementations of optimisation problems in economy & environment.

Compile: We have implemented a compiler for the language extension PolyP [Jansson, 2004b], experimented with embedding generic programming support in Haskell [Norell and Jansson, 2004a] and been involved in implementing Agda [Norell et al., 2008].

4.3 Other relevant experience

Parametricity theory and applications. Thanks to the Curry-Howard isomorphism, the type of each program correspond to a theorem. There is another relationship between types and propositions: each type-assignment gives rise to another theorem (the

parametricity condition) about the object being typed. Bernardy et al. [2010b] have investigated how to integrate the above result in dependently typed languages. In that context, the net effect is that for every type given by the programmer, an additional property becomes available (for free) for showing the correctness of the program. An interesting application of parametricity is in property based testing of polymorphic functions [Bernardy et al., 2010a]. We expect to find more uses of parametricity during this project.

Libraries for specifications and programs. 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 program derivation in the style of Bird and de Moor, in Agda. A program is coupled with an algebraic derivation from a specification, whose correctness is guaranteed by the type system. In this project we want to go further in this direction and develop useful libraries of programs and proofs with corresponding types and theorems.

5 Significance

The production of correct software is a problem which remains unsolved, and is of great economic significance. By leveraging the potential 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. A further advantage of this approach is that the skills required to construct programs are directly applicable to understanding the specifications.

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. We aim to go beyond state-of-the-art when it comes to expressivity of libraries for programming with dependent types, which is a relatively unexplored niche. By doing so, we hope to improve the software technology field in general, as these libraries should serve as examples of good design for other applications.

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/journal talks/papers (on the techniques used to create the libraries as well as on the amendments made to the languages with dependent types), and (post-)doctoral training.

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 implementors of dependently-typed languages, and downstream with end-users of frameworks for formal modeling and implementation. In fact, we believe that we are in the position to fill in the niche of producing libraries for dependently-typed languages, which are in demand from both sides, but currently lacking.

On the upstream side, we are in direct contact with the group currently in charge of the development of Agda: The main developers, Norell and Danielsson, were Jansson's students; and Agda Implementors' Meetings are held yearly at Chalmers. These meetings regularly attract participants from research groups in Nottingham Univ., TU Munich, and AIST (in Japan), among others. We have also close contacts with the programming-logic group at Univ. of Gothenburg, which deals with the fundamental aspects of type-theory.

Downstream, we have contacts with domain experts at the Potsdam Institute for Climate Impact Research (PIK), which are in demand of 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, matching the implementation with the models is important.

7 Organisation and budget

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 (20%), J-P Bernardy (PostDoc 65%)[1], a PhD student (not paid by the project) and several MSc thesis students (not paid by the project). We apply for 70% of the total project cost from VR, the rest is covered by Chalmers and other sources. We will benefit from work on generic libraries and high-level modelling done at (and funded by) PIK (Daniel Lincke, Cezar Ionescu).

The first year of project is devoted to library support for Algebra of Programming (milestone **AoP**), the second and third year focus is on **Optim** and the last year **Compile**. Jansson and Bernardy will together supervise the PhD student towards her PhD on "Strongly Typed Libraries for Programs and Proofs".

Jansson is 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, will provide more long-term basic research in the software technology of the future.

[1] J-P Bernardy is currently a PhD student in the FP group but will defend his thesis in June 2011.

8 An example illustrating the iterative process

Assume an ordering relation on values of type A. This may be expressed as follows.

```
postulate
  A : Set
  _⩽_ : Rel A
  ⩽−reflexive  : Reflexive _⩽_
  ⩽−transitive : Transitive _⩽_
```

Furthermore, using $a \leqslant? b$, we can decide which of a or b is bigger.

```
data Comp (a b : A) : Set where
  leq : a ⩽ b → Comp a b
  gt  : b ⩽ a → Comp a b
postulate
  _⩽?_ : (a b : A) → Comp a b
```

Using the above, it is possible to implement insertion into a sorted list.

```
insert : A → List A → List A
insert x [] = [x]
insert x (x′ :: xs) with x ⩽? x′
insert x (x′ :: xs) | leq _ = x :: x′ :: xs
insert x (x′ :: xs) | gt  _ = x′ :: insert x xs
```

In phase 1 we might want to prove that it preserves the property *OrdAb lo* that the list is *Ord*ered and *Ab*ove a lower bound *lo*. The specification of this property and its proof are as follows:

```
OrdAb : A → List A → Set
OrdAb low [] = ⊤
OrdAb low (x :: xs) = (low ⩽ x) × OrdAb x xs
insert−ordered : ∀ {lo} x xs → lo ⩽ x → OrdAb lo xs → OrdAb lo (insert x xs)
insert−ordered x [] p q = (p, tt)
insert−ordered x (x′ :: xs) p q with x ⩽? x′
insert−ordered x (x′ :: xs) p q | leq y = (p, (y, proj₂ q))
insert−ordered x (x′ :: xs) p q | gt y  = (proj₁ q, insert−ordered x xs y (proj₂ q))
```

Independently, we might want to check that the insertion contains all the elements of the input list. We do this by defining $xs \backsimeq ys$ to express that xs is a permutation of ys:

```
data _≋_ : Rel (List A) where
  stop  : [] ≋ []
  skip  : ∀ x {xs xs′}   → xs ≋ xs′ → x :: xs ≋ x :: xs′
  swap  : ∀ x y xs       → y :: x :: xs ≋ x :: y :: xs
  trans : ∀ {xs xs′ xs″} → xs ≋ xs′ → xs′ ≋ xs″ → xs ≋ xs″
insert−keeps : ∀ x xs → insert x xs ≋ x :: xs
insert−keeps x [] = refl
insert−keeps x (x′ :: xs) with x ⩽? x′
insert−keeps x (x′ :: xs) | leq _ = refl
insert−keeps x (x′ :: xs) | gt _ = trans (skip x′ (insert−keeps x xs)) (swap x x′ xs)
```

In phase 2 we recognise that the induction pattern of both proofs about insertion is the same, and we capture it in a proof-pattern of the following type (we omit the definition for concision):

```
insert−ind : (P : A → List A → List A → Set) →
  (emp : ∀ x → P x [] [x]) →
  (les : ∀ x x′ xs → x ⩽ x′ → P x (x′ :: xs) (x :: x′ :: xs)) →
  (gre : ∀ x x′ xs res → x′ ⩽ x → P x xs res → P x (x′ :: xs) (x′ :: res)) →
  (∀ x xs → P x xs (insert x xs))
```

The proofs of preservation of ordering and permutation can then be easily constructed by application of the *insert−ind* pattern to the specific induction hypothesis (the rest of the arguments can be guessed by Agda's built-in proof search mechanism). For example:

```
insert−ordered : ∀ {lo} x xs → lo ⩽ x → OrdAb lo xs → OrdAb lo (insert x xs)
insert−ordered {lo0} x xs = insert−ind
  (λ x′ xs′ res → (lo : A) → lo ⩽ x′ → OrdAb lo xs′ → OrdAb lo res)
  (λ x′ lo x0 x1 → (x0, tt))  {-automatic -}
  (λ x′ x0 xs′ x1 lo x2 x3 → (x2, x1, proj₂ x3))  {-automatic -}
  (λ x′ x xs′ res xs rec lo x3 p → (proj₁ p, rec x xs (proj₂ p))) x xs lo0  {-automatic -}
```

In phase 3 we realise that the construction of the induction-pattern for proofs can be automatically deduced for any total function, and investigate how their generation can be automated, perhaps via a modification of the Agda system.

References

R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer, 1999.

J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In A. Gordon, editor, *Proc. of ESOP 2010*, volume 6012 of *LNCS*, pages 125–144. Springer, 2010a.

J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proc. of ICFP 2010*, pages 345–356. ACM, 2010b.

J.-P. Bernardy, P. Jansson, M. Zalewski, and S. Schupp. Generic programming with C++ concepts and Haskell type classes—a comparison. *J. Funct. Program.*, FirstView:1–32, 2010c. doi: 10.1017/S095679681000016X.

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

G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA '98*, pages 183–200. ACM, 1998. doi: 10.1145/286936.286957.

A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proc. of the 14th ACM SIGPLAN international conference on Funct. programming*, ICFP '09, pages 79–90. ACM, 2009.

N. A. Danielsson. Total parser combinators. In *Proc. of the 15th ACM SIGPLAN international conference on Funct. programming*, ICFP '10, pages 285–296. ACM, 2010.

N. A. Danielsson and P. Jansson. 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, July 2004.

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

M. Flatt. Composable and compilable macros:: you want it when? In *ICFP '02*, pages 72–83. ACM, 2002. doi: 10.1145/581478.581486.

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

P. Jansson. The WWW home page for polytypic programming. `http://www.cse.chalmers.se/~patrikj/poly/`, 2004a.

P. Jansson. The PolyP 1.6 compiler. Available from the Polytypic prog. page [Jansson, 2004a], 2004b.

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

P. Jansson and J. Jeuring. PolyLib – a polytypic function library. Workshop on Generic Programming, Marstrand, June 1998a. Available from the Polytypic prog. page [Jansson, 2004a].

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

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

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

D. Lincke, P. Jansson, M. Zalewski, and C. Ionescu. Generic libraries in C++ with concepts from high-level domain descriptions in Haskell: A DSL for computational vulnerability assessment. In *IFIP Working Conf. on Domain Specific Languages*, volume 5658/2009 of *LNCS*, pages 236–261, 2009.

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

J. Morgenstern and D. R. Licata. Security-typed programming within dependently typed programming. In *ICFP '10*, pages 169–180. ACM, 2010. doi: 10.1145/1863543.1863569.

S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming using dependent types. In *Mathematics of Program Construction*, volume 5133/2008 of *LNCS*, pages 268–283. Springer, 2008.

S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in Agda: dependent types for relational program derivation. *J. Funct. Program.*, 19:545–579, 2009. doi: 10.1017/S0956796809007345.

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

U. Norell and P. Jansson. Prototyping generic programming in Template Haskell. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 314–333. Springer, 2004a.

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

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

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

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

T. Sheard and S. Peyton Jones. Template meta-programming for haskell. In *Proc. of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16. ACM, 2002.

A. A. Stepanov and M. Lee. The standard template library. Technical Report HPL-95-11(R.1), Hewlett Packard Laboratories, Palo Alto, CA, USA, Nov. 1995.

The Coq development team. The Coq proof assistant, 2010.

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2011-2993-88525-26

Name of applicant
Jansson, Patrik

Date of birth
720311-7515

Title of research programme
Strongly Typed Libraries for Programs and Proofs

Appendix B

Curriculum vitae

Vetenskapsrådet, Box 1035, SE-101 38 Stockholm, tel. +46 (0)8 546 44 000, vetenskapsradet@vr.se

Curriculum Vitæ: Patrik Jansson, 1972-03-11

1. Higher education degree:

1995: BSc+MSc degrees in Eng. Physics + Eng. Mathematics from Chalmers, Sweden. I graduated almost two years before schedule as the best student of my year.

2. Doctoral degree:

2000: Ph.D. degree in CS from Chalmers, Sweden, on *Functional Polytypic Programming*, Advisor: Johan Jeuring.

3. PostDoc and guest research:

1998, 1998, 2001: Research visits (2 + 2 + 3 months) to Northeastern University, Boston, USA; Oxford University Computing Lab, UK; Dept. of Computer Science, Yale, USA.

4. Qualification as Assoc. Professor:

2004: Docent (Associate Prof.) degree from Chalmers, Sweden.

5. Current Employment:

2004–now: Assoc. Prof., Chalmers. Research 50% (2011).

6. Prev. Employment and Education:

1991–1992: Military service: Crypto analyst at the National Defence Radio Establishment (FRA). Hand picked (as one out of five per year in Sweden) for 15 months of special education in mathematics, statistics and cryptanalysis.

2001–2004: Ass. Prof. in CS, Chalmers.

7. Interruptions in research:

Parental leave with Julia (1999) and Erik (2004) for a total of one full time year.

2002–2005: Director of studies of the CS dept. On average 35% / year for three years.

2005–2008: Vice head of the CSE dept. On average 50% of full time / year for four years.

8. Supervision experience:

I was PhD advisor of Ulf Norell (PhD 2007) and Nils Anders Danielsson (PhD 2007). With Norell I worked on generic programs and proofs and with Danielsson I worked on program correctness through types. Both Norell and Danielsson were immediately employed as PostDocs after their PhD (Norell here at Chalmers and Danielsson in Nottingham). I have supervised over 20 MSc and BSc project students.

I currently supervise the PhD students Jean-Philippe Bernardy (Lic. 2009, expected PhD in 2011), Cláudio Amaral (started March 2010) and Jonas Duregård (started August 2010). I also have partial responsibility for two other PhD students: Ramona Enache (I am her examiner) and Nick Smallbone (I am his co-supervisor).

I have been a member of the evaluation committee of three PhD defenses at Chalmers (T. Gedell, CSE (2008), M. Zalewski, CSE (2008), H. Johansson, Physics (2010)).

9. Awards, grants, etc.:

1991: Winner of the Swedish National Physics Olympiad

1991: Represented Sweden in the International Physics Olympiads.

1991: Represented Sweden in the International Mathematics Olympiad.

1996: Received the John Ericsson medal for outstanding scholarship, Chalmers

1997–2004: Obtained travel grants (277k SEK in total) from several private foundations

2003–2005: Co-applicant on *Cover — Combining Verification Methods in Software Development* funded with 8M SEK by the Swedish Foundation for Strategic Research.

2003–2005: Main applicant on the project *Generic Functional Programs and Proofs* funded with 1.8M SEK by VR.

2008–present: Elected member of IFIP (International Federation for Information Processing) Working Group 2.1 on "Algorithmic Languages and Calculi".

2009–present: Elected member of the faculty senate, Chalmers.

2009–2012: Co-applicant on "Software Design and Verification using Domain Specific Languages" funded with 11M SEK by the Swedish Science Council (VR, multi-project grant in strategic ICT).

2010–2012: Co-applicant and work-package leader in the Coordination Action "Global Systems Dynamics and Policy" (GSDP) funded with 1.3M EUR by the EU (ICT-2009.8.0 FET Open).

I have been reviewer for Journal of Functional Programming, Science of Computer Programming, Principles of Programming Languages and several other journals and conferences.

Leadership experience:

2002–2008: Member of the steering group of the department.

2002–2005: Director of Studies for the BSc and MSc education at the CS department

2005–2008: Vice head of the CSE dept. responsible for the BSc and MSc education.

2008–2010: Deputy project leader of the IMPACT project at Chalmers ("Development of Chalmers' New Master's Programmes", 30M SEK).

2009: Head of steering group of Chalmers eScience Initiative.

2011-: Head of the 5-year education programme in Computer Science and Engineering (Civilingenjör Datateknik, Chalmers).

Future goal:

In the long term I want to lead a strong academic environment developing theories, programming languages, libraries and tools for advanced software technology in general and generic programming in particular. To achieve this I have to attract funding for new PhD students and PostDocs and I have to continue my close collaboration with the researchers which make up the excellent local research environment.

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2011-2993-88525-26

Name of applicant
Jansson, Patrik

Date of birth
720311-7515

Title of research programme
Strongly Typed Libraries for Programs and Proofs

Appendix C

List of publications

Vetenskapsrådet, Box 1035, SE-101 38 Stockholm, tel. +46 (0)8 546 44 000, vetenskapsradet@vr.se

C Publication lists

Selected Publications: Patrik Jansson

Note to non computer scientists Conference articles in computer science are peer reviewed full articles — not 1–2 page abstracts, and are the normal form of refereed publication. The top conferences in each subfield (like *POPL* and *ICFP* below) typically have the highest impact factor within that field, higher even than any journal.

Most cited publications (Google Scholar, 2011-04-07)

Jansson's Hirsch-index is 16, his total citation count is over 1100 and the following papers are the five most cited.

P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *Proc. POPL'97: Principles of Programming Languages*, pages 470–482. ACM Press, 1997.
Number of citations: 294.

R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic programming: An introduction. In *Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer, 1999.
Number of citations: 166.

J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury et al., editors, *Advanced Functional Programming '96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.
Number of citations: 149.

P. Jansson and J. Jeuring. Polytypic data conversion programs. *Science of Computer Programming*, 43(1):35–75, 2002.
Number of citations: 57.

M. Benke, P. Dybjer, and P. Jansson. Universes for generic programs and proofs in dependent type theory. *Nordic Journal of Computing*, 10(4):265–289, 2003. ISSN 1236-6064.
Number of citations: 52.

Journal articles (last 8 years, excluding the above)

J.-P. Bernardy, P. Jansson, M. Zalewski, and S. Schupp. Generic programming with C++ concepts and Haskell type classes — a comparison. *J. Funct. Program.*, 20(3–4):271–302, 2010c. URL `http://dx.doi.org/10.1017/S095679681000016X`.
Number of citations: 2.

S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming in Agda: dependent types for relational program derivation. *J. Funct. Program.*, 19:545–579, 2009. doi: 10.1017/S0956796809007345.
Number of citations: 16.

Articles in refereed collections and conf. proceedings (last 8 years)

J.-P. Bernardy, P. Jansson, and R. Paterson. Parametricity and dependent types. In *Proc. of ICFP 2010*, pages 345–356. ACM, 2010b.
Number of citations: 16.

J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In A. Gordon, editor, *Proc. of ESOP 2010*, volume 6012 of *LNCS*, pages 125–144. Springer, 2010a.
Number of citations: 7.

D. Lincke, P. Jansson, M. Zalewski, and C. Ionescu. Generic libraries in C++ with concepts from high-level domain descriptions in Haskell: A DSL for computational vulnerability assessment. In *IFIP Working Conf. on Domain Specific Languages*, volume 5658/2009 of *LNCS*, pages 236–261, 2009.
Number of citations: 12.

A. Rodriguez, J. Jeuring, P. Jansson, A. Gerdes, O. Kiselyov, and B. C. d. S. Oliveira. Comparing libraries for generic programming in Haskell. In *Haskell'08*, pages 111–122. ACM, 2008.
Number of citations: 41.

J.-P. Bernardy, P. Jansson, M. Zalewski, S. Schupp, and A. Priesnitz. A comparison of C++ concepts and Haskell type classes. In *Proc. ACM SIGPLAN Workshop on Generic Programming (WGP)*, pages 37–48. ACM, 2008.
Number of citations: 22.

S.-C. Mu, H.-S. Ko, and P. Jansson. Algebra of programming using dependent types. In *Mathematics of Program Construction*, volume 5133/2008 of *LNCS*, pages 268–283. Springer, 2008.
Number of citations: 9.

P. Jansson, J. Jeuring, and students of the Utrecht University Generic Programming class. Testing properties of generic functions. In Z. Horvath, editor, *Proceedings of IFL 2006*, volume 4449 of *LNCS*, pages 217–234. Springer-Verlag, 2007.
Number of citations: 4.

N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *POPL'06*, pages 206–217. ACM Press, 2006.
Number of citations: 41.

N. A. Danielsson and P. Jansson. 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, July 2004.
Number of citations: 27.

U. Norell and P. Jansson. Prototyping generic programming in Template Haskell. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 314–333. Springer, 2004a.
Number of citations: 10.

U. Norell and P. Jansson. Polytypic programming in Haskell. In *Implementation of Functional Languages 2003*, volume 3145 of *LNCS*, pages 168–184. Springer, 2004b.
Number of citations: 35.

Publicly available implementations (last 8 years)

I have designed and implemented a compiler for the polytypic language PolyP:

P. Jansson. The PolyP 1.6 compiler. Available from the Polytypic prog. page [Jansson, 2004a], 2004b.

I have also participated in the development of the Agda proof engine,

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

The first description of Agda was in the PhD thesis of Ulf Norell (2007) and it has been cited $\simeq 50$ times / year since then, indicating a quick spread in academia.

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2011-2993-88525-26

Dnr
2011 -

Name of applicant
Jansson, Patrik

Date of birth
720311-7515

Reg date
2011-04-13 15:17:29

Project title
Strongly Typed Libraries for Programs and Proofs

NT

NT-S, NT-R

Project research grant NT 13 April 2011
Project Research Grant
Computer Science
Chalmers tekniska högskola

Appendix S

Signatures

2012	2013	2014	2015	2016	
1046	1081	1114	1152		2012-01-01 -- 2015-12-31

Applicant | **Date**

Head of department at host University | **Clarifi cation of signature** | **Telephone**

A signature on the application is required not only from the applicant but also from the authorised representative of the host university/institution or equivalent (normally the head of the department or establishment where the research is to be conducted). The signature confirms that the department can accommodate the proposed research, position or equipment; that the costing in the application is approved for the department's part, that any proposed experimentation on human or animal subjects has been reported, and that the applicant has reported any secondary occupations and commercial ties (s)he may have, and nothing inconsistent with good research practice has thereby emerged. The applicant must have discussed these conditions with the representative of the host university/institution

International Postdoctoral Fellowships are administered by the Swedish Research Council. The only signature required on the application is that of the applicant.

Vetenskapsrådets noteringar
Kod

2011-2993-88525-26

Vetenskapsrådet, Box 1035, SE-101 38 Stockholm, tel. +46 (0)8 546 44 000, vetenskapsradet@vr.se