Appendix A

Research Program

Main objectives

Our long term goal is to create systems (theories, programming languages, libraries and tools) which make it easy to develop generic software together with proofs of its correctness.

The main goal of this research project is to improve the theory and implementation of strongly typed, generic functional programming.

- On the theoretical side we introduce dependent types to be able to express generic proofs as well as programs and to provide a solid semantic foundation for generic functional programming.
- On the implementation side we use staged compilation and partial evaluation techniques to generate more efficient code.

We will also compare the approaches to generic programming in the object oriented and the functional programming paradigms.

Research area overview

The ability to name and reuse common patterns of computation (as functions, objects, etc.) is a very important aspect of every programming language. In strongly typed languages this is tightly coupled to different kinds of polymorphism. The polymorphism in object oriented languages is based on subtyping and inheritance, while functional languages use parametric polymorphism. Both these kinds of polymorphism have been thoroughly studied and extensively used in the last few decades. Recently, with the increased use of user-defined datatypes, the need for a third kind of polymorphism (to type algorithms which work for different datatypes) has emerged. The project proposal forms an important part of the ongoing investigation and implementation of this third kind of polymorphism.

Many algorithms have to be implemented over and over again for different datatypes, either because datatypes change during the development of programs, or because the same algorithm is used for several datatypes. Examples of such algorithms are equality tests, traversals and pretty printers. Polytypic programming is a paradigm for expressing such algorithms.

In contrast to parametric polymorphic functions whose instances all do the same thing, the instances of a polytypic function are different, but share a common structure. Each polytypic instance can be obtained by instantiating a template algorithm with (the structure of) a datatype. In functional programming this structure is a datatype definition [5], while in object oriented programming this structure is a class graph [7].

Strategic relevance

The research on generic functional programming is an important part of the ongoing quest for higher level programming languages (shorter development times) and more reliable software (strong type and proof systems).

Compared with traditional generic programming (C++ templates), already normal functional programming provides much of that power through parametric polymorphism and higher order functions. Generic functional programming (polytypic programming) takes this even further and offers a number of benefits:

Reusability: Polytypism extends the power of polymorphic functions to allow classes of related algorithms to be described in one definition. Thus polytypic functions are very well suited for building program libraries.

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

Applications: Some algorithms are polytypic by nature: maps and traversals, pretty printing and parsing, data compression, term rewriting, etc. Each of these can be described informally in a datatype independent way, and with polytypic programming this informal description can be turned into a formal definition.

Provability: More general functions means more general proofs. If we consider polytypic proofs, then each of the earlier benefits obtains an additional interpretation: we get reusable proofs, adaptive proofs, and new proofs of properties of printing and parsing, packing, term rewriting etc.

Related work

Some international projects and research groups work on polytypic programming (without dependent types) under different names:

- *Generic functional programming* [2] in Oxford (Richard Bird *et al.*) and in Nottingham (Roland Backhouse *et al.*). They have done foundational theoretical work in a relational setting.
- The *Generic Haskell* project in Utrecht led by Johan Jeuring extends the purely functional language Haskell with *type indexed functions* [5].
- *Adaptive OOP* [7], developed by Karl Lieberherr, is the closest relative to polytypic programming in the object oriented programming world. The work on Adaptive OOP starts from a different paradigm compared with generic functional programming, but the general ideas are the same. Much can be learned on both sides by combining and comparing what has been found about polytypic programming in the two paradigms.
- The work on *Intensional Type Analysis* [4], pioneered by Harper and Morrisett, applies polytypic programming to compilation of functional languages. They aim at efficient code by specialization and later work (Cornell, Princeton, Yale) has applied polytypic ideas to obtain type-safe garbage collection [10, 8].
- Barry Jay *et al.* in Sydney work on a theory of polytypism called *shape polymorphism* [6]. They are implementing a language called FISh2 with support for polytypic definitions.

The connection between polytypic programming and dependent types has been investigated by Pfeifer & Rueß [9] (for some simple cases) and Dybjer & Setzer [3] (theoretical foundations).

Project description

The project consists of four parts:

- **Type systems and semantics of polytypic programming**

  Two systems for generic functional programming are PolyP, developed at Chalmers, and its successor Generic Haskell, developed in Utrecht. The two system implementations have different limitations and strengths. PolyP is good at type checking and type inference but accepts only a limited language. Generic Haskell covers the whole language but lacks type checking and type argument inference. Both systems can

express and instantiate generic (Haskell) functions but neither system can deal with proofs or dependent types.

This subproject aims at a new formulation of the type system and the semantics of generic functional programming based on (a restriction of) dependent types instead of (an extension of) Haskell types.

Using dependent types in the semantics also suggests that we should extend polytypic definitions to use dependent types as well (see the following subproject).

- **Type-indexed proofs** (in collaboration with Peter Dybjer and Marcin Benke.)

  Polytypic programming is about defining families of functions indexed by types. Traditionally the functions have been in sugared lambda calculus (Haskell) and the types have been recursive sum-of-product types. Dependent type theory gives a much more expressive language with types parametrized by values. The goal of this subproject is to investigate how the the expression language and the type index language of normal polytypic programming can be extended to a dependently typed language.

  Extending the expressions with dependently typed constructs seems relatively straightforward and increases expressibility significantly — we can express polytypic proofs of properties of polytypic functions. The challenge is to find a suitable extension of the dependent type system to be able to express these polytypic properties. Here we build on recent work by Dybjer and Setzer [3] on finite axiomatizations of a very general framework for dependent type theory, including so called inductive-recursive definitions. They use essentially a type of codes for datatypes definable in their system. Since they allow definition by recursion on this type of codes it is possible to define very general polytypic programs and proofs. We will study the practical implications of this work for polytypic programming with dependent types.

- **More efficient implementation of polytypism** (in collaboration with John Hughes and Johan Jeuring)

  The current implementations of generic functional programming are prototypes designed for experimentation rather than efficient code generation. The last year has seen a rapid development of the Generic Haskell implementation and the time for more demanding applications has come.

  Applying partial evaluation, deforestation and staged programming techniques to the instances generated from a polytypic definition should result in optimized code as good as (or better) than hand-written low level code. The goal of this subproject is to bring theory and tools from partial evaluation to good use in implementing polytypic programming. Combining the local expertise in partial evaluation (Hughes) and polytypic programming (Jansson) with the implementation experience of the Generic Haskell team (Utrecht) we are confident that we can make rapid progress.

- **Polytypic programming and OOP** in collaboration with Karl Lieberherr, Boston.

  This subproject will focus on how ideas from generic functional programming can be mapped to Adaptive OOP and vice versa.

  Qualitatively we know that when the type argument is made more general, the set of expressible polytypic functions becomes smaller (some functions do not make sense for the more complicated types). Conversely, by restricting the type argument to a certain class of types (for example, expression trees) we can express more polytypic functions (for example unification and rewriting). We will investigate these parts of the design space in more detail to find the boundaries of polytypic expressibility.

We plan to work on the four subprojects in parallel over a period of four years, with Benke focusing on "Type-indexed proofs". The division of work between the main applicant and the PhD student will naturally vary as the student will need some time to "get into" the project. Some parts are likely to be carried out by Masters Thesis students, and in fact, two projects in this area have already started.

Preliminary findings

We have developed a theory for functional polytypic programming, shown how to construct and prove properties of polytypic algorithms, presented the language extension PolyP for implementing polytypic algorithms in a type safe way, and implemented a number of applications in PolyP. The applications include a library of basic polytypic building blocks, PolyLib and two larger applications of polytypic programming: rewriting and data conversion. In both applications papers a number of polytypic properties are proved "by hand". In the proposed subproject "Type-indexed proofs" we will be able to express these (and other) proofs as constructive proof objects in a formal system.

PolyP extends a functional language (a subset of Haskell) with a construct for defining polytypic functions by induction on the structure of user-defined datatypes. The polytypic definitions are type checked and the generated instances are guaranteed to be type correct. We have implemented a compiler that translates PolyP programs to Haskell. The experience gained from experimenting with the PolyP implementation and the successor (Generic Haskell) forms the basis for the subproject "More efficient implementation of polytypism".

A PolyP function can be applied to values of a large class of datatypes, but some restrictions apply. The implementation requires that a polytypic function is applied to values of *regular* datatypes only. A one-parameter datatype $D\ a$ is regular if it is not mutually recursive, contains no function spaces, and if the argument of the datatype constructor on the left- and right-hand side in its definition are the same. (The collection of regular datatypes still contains many conventional recursive datatypes, such as lists and different kinds of trees.) The reason for these restrictions is to stay close to the Haskell type system and ensure

polytypic functions can be type checked. To remove some of these restrictions is the aim of the subproject "Type systems and semantics of polytypic programming".

Patrik Jansson, two masters students and Marcin Benke (Chalmers) have already started exploring the subproject "Type-indexed proofs". The preliminary results have shown that generic functional programming can be modeled using dependent types. A joint project application (Chalmers, Durham, Nottingham and Utrecht) about programming with dependent types is being drafted for the European Sixth Framework Programme.

Initial work on "Polytypic programming and OOP" has been carried out during Patrik Jansson's two month visit to Boston, 2001. I have also been in contact with Johan Larsson (a PhD student at KTH, Sweden) who has a strong background in OOP and functional programming and who is working on polytypic programming from this combined perspective.

International collaboration

I 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. I have had very good experience from my research visits to different departments and from the visits of other researchers to Chalmers. I therefore apply for money so that I and my PhD student can travel to meet researchers at other sites *and* for inviting other researchers to visit us to do collaborative work.

The series of Workshops on Generic Programming (WGP) was started by Johan Jeuring and myself with the first meeting in Marstrand in 1998. The second workshop was held in Portugal, 2000 and the third is held the Netherlands, 2002. I would like to organize a WGP in Sweden, 2004, as part of this research program.

International contacts relevant for this application:

- I am involved in the Generic Haskell project in Utrecht (led by Johan Jeuring) in which we combine the experiences from the PolyP system with Hinze's new ideas about *type indexed functions* (extensions to handle multiple type arguments, mutually recursive datatypes). I have visited Utrecht for shorter times at least six times the last few years and Johan Jeuring has also visited me at Chalmers about that often.

- I visited Karl Lieberherr, Northeastern, Boston for ten weeks in the spring of 2001. Karl is working on the closest relative to polytypic programming in the object oriented world: *Adaptive OOP*. This visit was very interesting and gave a lot of ideas for cross-fertilizing the research on functional and object oriented polytypism.

- I have had recent discussions with Altenkirch (Nottingham), McBride (Durham), Jeuring (Utrecht), Benke (Chalmers) and Dybjer (Chalmers) about generic programming and dependent types [1]. We are now working on writing a joint project

application (Chalmers, Durham, Nottingham and Utrecht) for the European Sixth Framework Programme.

- I visited Paul Hudak and his functional programming group at Yale for three months in 1998. There I got in contact with Zhong Shao and the FLINT project (an intermediate language for compilation of strongly typed languages). The FLINT team are experimenting with polytypic programming and dependent types.
- I have good contacts with the *generic functional programming* groups in Nottingham (Roland Backhouse *et al.*) and Oxford (Richard Bird *et al.*). I visited them for two months in 1998.

Equipment

The project uses standard computing equipment at the computing science department at Chalmers and a laptop to be purchased for the project (to simplify travel, seminars and work from home).

Research group members and other funding

Both the main applicant (Patrik Jansson) and a PhD student (Ulf Norell, starting 1st of September 2002) would be paid 80% from the proposed project and 20% from the department of computing science, Chalmers/GU. (These 20% correspond to time for departmental activities — mainly pedagogical work.) Marcin Benke would be paid 25% for working mainly on the subproject "Type-indexed proofs".

Patrik Jansson and Marcin Benke are currently employed as assistant professors (forskarassistenter) at Chalmers.

The following list describes the current funding situation:

- Assistant professor Patrik Jansson: 100% Chalmers (research time 65%)
- PhD student Ulf Norell: starting Sept. 2002 (research time 80%)
- Assistant professor Marcin Benke: 100% Chalmers (research time 65%)

A recent grant application to SSF (decision due in June 2002) is related to the current application:

> *Combining Verification Methods in Software Development*, main applicants: Thierry Coquand, Peter Dybjer, John Hughes, and Mary Sheeran

Jansson and Benke would be paid 50% from SSF (2003–2004) if that project should be fully funded.

Bibliography

[1] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In Jeremy Gibbons and Johan Jeuring, editors, *Proceedings of the IFIP WG2.1 Working Conference on Generic Programming 2002*. Kluwer, July 2002. Accepted; to appear. `http://www.dur.ac.uk/c.t.mcbride/generic/`.

[2] Richard Bird, Oege de Moor, and Paul Hoogendijk. Generic functional programming with types and relations. *Journal of Functional Programming*, 6(1):1–28, 1996.

[3] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive and inductive-recursive definitions. In Jean-Yves Girard, editor, *Proceedings 4th Int. Conf. on Typed Lambda Calculi and Applications, TLCA'99, L'Aquila, Italy, 7–9 Apr 1999*, volume 1581 of *LNCS*, pages 129–146, Berlin, April 1999. Springer-Verlag.

[4] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *22nd Symposium on Principles of Programming Languages, POPL '95*, pages 130–141, 1995.

[5] Ralf Hinze. A new approach to generic functional programming. In *POPL'00*, pages 119–132. ACM Press, 2000.

[6] C. Barry Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[7] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

[8] Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proc. ACM SIGPLAN '01 Conf. on Prog. Lang. Design and Implementation*, pages 81–91, New York, 2001. ACM Press.

[9] Holger Pfeifer and Harald Rueß. Polytypic proof construction. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proc. 12th Intl. Conf. on Theorem Proving in Higher Order Logics*, number 1690 in LNCS, pages 55–72. Springer-Verlag, 1999.

[10] Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *POPL 2001: The 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–178, 2001.

Appendix B

Curricula Vitae

B.1 Curriculum Vitae: Patrik Jansson

Assistant Professor in Computing Science
Chalmers University of Technology
patrikj@cs.chalmers.se
Born: March 11, 1972

Education and Other Experience

July 2001 – present: Assistant Professor in Computing Science, Chalmers.

Aug 2000 – June 2001: PostDoc ("Doktorstjänst") in Computing Science, Chalmers.

1995–2000: Successful graduate studies at the Computing Science Department, Chalmers resulting in the Ph.D. thesis, *Functional Polytypic Programming*, 2000. Advisor: Johan Jeuring, Utrecht.

1992–1995: Undergraduate studies in Engineering Physics, Chalmers. I completed a M. Sc. in Engineering Physics in March 1995, graduating almost two years before schedule as the best student of my year and was awarded the University Medal (John Ericsson Medal).

1991–1992: Military service: hand picked (as one out of five per year in Sweden) for 15 months of special education in mathematics, statistics and cryptographic analysis.

Pedagogical Experience

Pedagogical courses: "Teaching and Learning in Higher Education" (2001–2002), "Teaching, Learning and Presentation" (1997), father of Julia Jansson (1999 – present).

Graduate teaching: "Functional Polytypic Programming" (2000).

Undergraduate teaching Masters project student supervision (2001, 2002), Databases (2000, 2001), Programming Languages (2000, 2001), Imperative Programming in Ada (1998, 1997), Introductory Programming in Haskell and Ada (1996, 1995), Programming in Modula 2 (1995), Numerical Analysis (1994), Programming in C (1990).

Research Collaboration and Committee Work

Spring 2001: Research visit (2 months) to Northeastern University, Boston.

2000–2001: On the program committee of *Haskell Workshop, 2001*.

Spring 2000: On the program committee of *Workshop on Generic Programming, 2000*.

1995–2000: Several shorter research visits to Utrecht, the Netherlands.

1997–2000: Hosted several visits to Chalmers by Johan Jeuring (Utrecht), Barry Jay (Sydney), Doaitse Swierstra (Utrecht) and Mark Jones (OGI).

Autumn 1998: Research visit (2 months) to Oxford University Computing Laboratory, UK.

Summer 1998: On the organizing committee of *Mathematics for Program Construction 1998*.

Spring 1998: Research visit (3 months) to the Department of Computer Science, Yale, USA.

Peer Review Duty

Referee on a number of occasion for journals, conferences and workshops, including *Journal of Functional Programming*, *ACM Transactions on Programming Languages and Systems*, *Mathematics of Program Construction*, *International Conference on Functional Programming*, *Typed Lambda Calculi and Applications*, *Workshop on Generic Programming* and *Workshop on Advanced Separation of Concerns*.

Awards Received

Represented Sweden in the International Mathematics Olympiad (1991) and the International Physics Olympiads (1991). Winner of the Swedish National Physics Olympiad (1991). Received the John Ericsson medal for outstanding scholarship, Chalmers, 1996.

Obtained travel grants from "Magnus Bergvalls Stiftelse" (2001), "Wenner-Gren Foundation" (2000), "Knut och Alice Wallenbergs Stiftelse" (2000, 1999) and "Adlerbertska Research Foundation" (1999, 1998, 1997).

References

Roland Backhouse, University of Nottingham, UK, `Roland.Backhouse@Nottingham.ac.uk`
Richard Bird, Oxford University, UK, `Richard.Bird@comlab.ox.ac.uk`
Barry Jay, University of Technology, Sydney, Australia, `cbj@socs.uts.edu.au`
Johan Jeuring, Utrecht University, the Netherlands, `johanj@cs.uu.nl`
Paul Hudak, Yale University, USA, `paul.hudak@yale.edu`
Karl Lieberherr, Northeastern University, USA, `lieber@ccs.neu.edu`

B.2 Curriculum Vitae: Marcin Benke

Family name:	Benke
First name:	Marcin
Born:	March 2, 1969, Warsaw, Poland
Present position:	Assistant Professor
	Department of Computer Science
	Chalmers Univeristy of Technology
	e-mail: `marcin@cs.chalmers.se`

Education — scientific degrees:

- **M.Sc.:** (with honours) Faculty of Mathematics, Informatics and Mechanics, Warsaw University, 1992, thesis *A compiler and programming environment for a lazy functional language* (in Polish).
- **Ph.D.** (with honours) Faculty of Mathematics, Informatics and Mechanics, Warsaw University, 1998, thesis *Complexity of type reconstruction in programming languages with subtyping*, advisor: Prof. Jerzy Tiuryn.

Employment:

- Since Oct 2001 — Assistant Professor, Chalmers University of Technology, Gothenburg, Sweden
- Oct 1998–Sep 2000 — Assistant Professor, Institute of Informatics, Warsaw University, Poland
- Oct 1992–Sep 1998 — Researcher, Institute of Informatics, Warsaw University, Poland

Longer scientific visits

- Cambridge University Computer Laboratory, Cambridge, UK, May–Aug 1996 — EUROFOCS Research Fellow.
- Department of Computing Sciences, Chalmers University of Technology, Gothenburg, Sweden, Oct 2000–Sep 2001 - Postdoctoral Fellow.

Scientific activities

My main research interests lie in the area of functional and object-oriented languages. In the years 1992-1997 my research concentrated on various forms of subtyping and constucting computationally feasible systems of type reconstruction with subtyping. The results from this period are collected in my PhD thesis. After my PhD I extended my interests towards polymorphic subtyping; some results about predicative polymorphic subtypes are outlined in my 1998 MFCS paper.

Currently I am working in the area of dependent types and proof-editors, focusing on methods for program verification.

One of the areas of my interest involves automatical proof generation. Preliminary results in this domain have been presented at the "Strategies 2001" workshop (part of IJCAR'2001) and the TPHOL'2001 conference.

Another area of my interest is generic programming, in particular, applications of dependent type systems to create generic functions and generic proofs.

Appendix C

Selected Publications

C.1 Selected Publications: Patrik Jansson

Journal articles

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

Patrik Jansson and Johan Jeuring. Functional pearl: Polytypic unification. *Journal of Functional Programming*, 8(5):527–536, September 1998.

Articles in refereed collections and conference proceedings

P. Jansson and J. Jeuring. Polytypic compact printing and parsing. In Doaitse Swierstra, editor, *ESOP'99*, volume 1576 of *LNCS*, pages 273–287. Springer-Verlag, 1999.

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

Invited tutorials

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-Verlag, 1999.

J. Jeuring and P. Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming '96*, volume 1129 of *LNCS*, pages 68–114. Springer-Verlag, 1996.

Articles in workshop proceedings

Patrik Jansson and Johan Jeuring. A framework for polytypic programming on terms, with an application to rewriting. In *Workshop on Generic Programming*. Utrecht University, 2000. UU-CS-2000-19.

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

PhD thesis

Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Computing Science, Chalmers University of Technology and Göteborg University, Sweden, May 2000.

Publicly available implementation

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

Patrik Jansson. The PolyP 1.6 compiler. Available from the Polytypic programming WWW page `http://www.cs.chalmers.se/~patrikj/poly/`

C.2 Selected Publications: Marcin Benke

Journal articles

(*) Marcin Benke. Some complexity bounds for subtype inequalities. *Theoretical Computer Science*, 212(1–2):3–27, February 1999.

Articles in refereed collections and conference proceedings

(*) Marcin Benke. Efficient type reconstruction in the presence of inheritance. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium*, volume 711 of LNCS, pages 272–280, Springer Verlag 1993.

Marcin Benke. A logical approach to complexity bounds for subtype inequalities. In Petr Hajek, editor, *Gödel'96; proceedings*, volume 6 of *Lecture Notes in Logic*. Springer Verlag, 1996.

Marcin Benke. Predicative polymorphic subtyping. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1998.

Marcin Benke. An algebraic characterization of typability in ML with subtyping. In Thomas, W., editor, *Foundations of Software Science and Computation Structures* Springer LNCS, 1578:104–119, 1999.

(*) Marcin Benke. Some tools for computer-assisted theorem proving in Martin-Löf type theory. in: R. J. Boulton, P. B. Jackson (Eds.) *Theorem Proving in Higher Order Logics — Supplemental Proceedings* University of Edinburgh, 2001.

Articles in refereed workshop proceedings

(*) Marcin Benke. Strategies for interactive proof and program development in Martin-Löf type theory. in: B. Gramlich, M. P. Bonacina (editors) *Proceedings of 4th International Workshop on Strategies in Automated Deduction*, Universitá degli studi di Siena 2001.

PhD thesis

(*) Marcin Benke. *Complexity of type reconstruction in programming languages with subtyping.* PhD thesis, Faculty of Mathematics, Informatics and Mechanics, Warsaw University, January 1998.

Appendix I

Prioritised Areas

Informationsteknik

Det moderna samhället blir för vart år som går allt mer beroende av informationsteknik på alla nivåer. Informationssamhället bygger i allt högre grad på varierad programvara och standardiserad hårdvara. Programsystem blir mer och mer komplicerade samtidigt som kraven på pålitlighet och korrekthet ökar — datorkrascher kan leda till stora ekonomiska skador och kan även ta människoliv. För att kunna möta korrekthetskraven på dagens och framtidens komplexa system behövs nya tekniker för utveckling av korrekt programvara. Vi behöver nya programmeringsspråk för att mera koncist och överblickbart kunna lösa stora programmeringsproblem och vi behöver nya metoder för verifiering av program och konstruktion av korrekta programsystem. För att kunna avgöra om ett program är korrekt behövs dels en specifikation (de krav som ställs på programmet), dels en semantik (en stringent definition av vad programspråkets konstruktioner betyder).

Modern typteori har visat att alla specifikationer kan uttryckas som typer i ett avancerat typsystem (beroende typer). Funktionella programspråk har under det senaste årtiondet visat sin styrka genom sina koncisa program och sin enkla semantik. Detta projekt handlar om grundforskning i gränsområdet mellan typteori och funktionell programmering och genom att kombinera och utveckla dessa båda områden kan vi komma ett steg på vägen mot mer pålitliga system inom informationstekniken.