Type Theory with hidden monotonic Resources
— Case for Support —

Thorsten Altenkirch Bernhard Reus
University of Nottingham University of Sussex

Previous Research Track Record

Thorsten Altenkirch

Thorsten Altenkirch is a Reader at the University of Nottingham and co-chair of the Functional Programming Laboratory currently comprising 4 members of staff and 13 PhD students. His main research interests are Type Theory, Functional Programming, Categorical Methods and Quantum Computing and he has published over 50 papers in this areas which are frequently cited (h-index $\geq$ 24). He has been Principal Investigator on four EPSRC projects, including *Observational Equality For Dependently Typed Programming* (EPSRC grant EP/C512022/1, *$*242,198), and *Reusability and Dependent Types* (EP/G034109/1, *$*244,671) and co-investigator on two more. He has participated in a number of EU projects, in particular in the Coordination Action TYPES and its predecessors. He organized the annual TYPES meeting in 2007 and specialized workshops on Dependently Typed Programming, in 2004 (Dagstuhl seminar 04381), 2008 in Nottingham and 2010 in Edinburgh. Altenkirch is editor of Fundamentae Informatica since 2007 and was PC member of FLOPS12, MFPS12, MSFP12, QPL11, TLCA11, FICS10, ITP10, WGP10, TPHOLs09 and CIE08. He was twice (in 2007 and 1012) invited Professor at Université Denis Diderot for one month each and in Spring 2013 he is a visiting researcher at the Institute for Advanced Study In Princeton for the whole term to work on Homotopy Type Theory.
Especially relevant for the current proposal is the joint work of Altenkirch and Swierstra on *functional semantics of effects* [**?**, **?**, **?**] which showed how effects can be integrated in Type Theory without giving up the appealing simplicity of the type-theoretic approach. His recent work on relative monads [**?**] (which is inspired by earlier joint work with Reus [**?**]) demonstrates how categorical concepts can be successfully exploited to structure type-theoretic developments.

Bernhard Reus

Dr Bernhard Reus is a Senior Lecturer in Computer Science at the University of Sussex with special interest in program semantics and logics. Reus received a distinction ("summa-cum-laude") for his PhD in Computer Science from the Ludwig-Maximilians-Universität in Munich in 1996, got a Lectureship at Sussex University in 2000 and was promoted to Senior Lecturer in 2006. He is a member of the Foundations of Software Systems group in the Department of Informatics at Sussex. His recent research contributions have been in the area of program verification and semantics for programming languages and logics, in particular languages with stored procedures. He led a small team to build a verification tool for C-like programs that supports (semi-)automatic reasoning about stored procedures (function pointers). In co-operation with colleagues in Saarbrücken, Copenhagen, Oxford and Paris he developed new models for the elegant reasoning about recursive predicates indexed by recursively defined worlds. The models developed are presheaf-models with extra structure which is relevant to the proposed project.
Reus has been working in the area of *Applied Semantics* for most of his research career. He co-authored a frequently cited new denotational semantics for the call-by-name λ-calculus with control operators and a derivation of an interpreter for this functional language (Krivine's machine) [**?**] and was involved in the development of a structural operational semantics of multi-threaded Java [**?**] and a OCL like Hoare-logic for a sequential Java-like language [**?**].

He has also made contributions to foundations, in particular domain theory and type theory. One of the main results of the PI was the *logical* development of *Synthetic Domain Theory* which is a variation of Domain Theory, suggested by Dana Scott, in which domains can be treated simply as sets. His distinguished PhD thesis [?] laid out the formal development of a flavour of Synthetic Domain Theory in a type-theoretical setting based on few axioms only. Reus verified all theorems [?] using the proof-checker *Lego* which implements a specific type theory (ECC). *Lego* was somehow a precursor of the popular type theoretical systems *Coq* and *Agda* that will be used in the proposed project. In order to ensure the consistency of this formalization, a realizability model was defined [?]. Later this work has been generalised to allow for more models in [?] which in turn has been generalised further by other researchers.
The two PIs have collaborated once before in the area of type theory, developing an elegant definition of λ-terms using a generalized form of polymorphism [?]. At the time they also co-organised and co-chaired the TYPES workshop in Kloster Irsee near Munich.
Grants Reus was the sole PI on the recent EPSRC project *From Reasoning Principles for Function Pointers To Logics for Self-Configuring Programs* (EP/G003173/1, $390,871) which investigated models and program logics for higher-order store, i.e. theories and tools that allow one to prove correctness of programs that use code pointers. One application of the developed semantics is the full soundness proof of Pottier's *capability types* [?]. Another outcome is a (semi-)automatic theorem prover, Crowfoot [?], that allows one to write C-like programs with stored procedures (function pointers), specify them using pre- and postconditions and automatically prove that the procedures meet the specification using just some annotations.
Reus was also the sole PI on the EPSRC project *Programming Logics for Denotations of Recursive Objects* (GR/R65190/01), $62,360) on program logics for Abadi and Cardelli's object-calculus. The main results of this project include a correctness proof of the Abadi-Leino logic [?] using denotational semantics [?, ?] that can explain the inherent limitations of the Abadi-Leino logic and a new logic for a simple imperative while-language where procedures are first class data that can be stored away and updated at runtime, but not dynamically allocated [?].
Reus has also obtained numerous smaller grants from the DAAD, Nuffield Foundation, EPSRC, and the London Mathematical Society, respectively, that funded research visits and the organisation of events at Sussex.
Usual stuff? numbers of publications? other stuff? The PI has published numerous refereed papers and articles in international conferences (LICS, POPL, CSL, ESOP, ICALP, FOSSACS, VMCAI) and journals (TCS, MSCS, JFP, LMCS) of high standing. He also co-chaired and co-organised events in both areas, notably the now well established *Domains* and *Types* workshops (in 1997, 2008 and in 1998, respectively) and co-edited journal volumes (TCS vol 264(2), MSCS vol 20(2)) and an LNCS volume (1657) with selected papers, respectively. He has been the PC Chair for Domains IX (Sussex, 2008), and on the PC for Domains X (2011), Classical Logic and Computation (Brno, 2010). Reus has supervised two PhD students and externally examined two PhD students.

References

Proposed Research

up to 6 pages

Summary

Dependently typed programming languages like Agda exploit a powerful type system to integrate programming and reasoning allowing us to freely move between prototypical programs and certifed deliverables. Dependently typed programming (DTP) is becoming increasingly popular and is being used for real world software like web servers and communication protocols (*citation*). This raises an important issue: how to integrate effectful programming and dependent types in a way that supports the engineering of certified programs effectively? In particular we want to be able to program and reason about monotone resources like memory or threads. We are going to investigate how to use a semantic construction from category theory (presheaf models) to support the ubiquitous monotone resources building on previous work by by Swierstra and Altenkirch (*citation*) on functional specification of effects.

Introduction and Overview

Functional programming is referentially transparent as side effects are prohibited or strictly controlled by a type system. Modern languages like Haskell, Python, F# and object-oriented variants like OCaml and Scala have massively contributed to the popularity of functional programming (citations needed). These languages are frequently used outside academia (citation/examples needed, need some real life data). (Patterns of functional Map-reduce???).
One main feature of these languages is their strong type system which allows the programmer to discover certain errors at compile time. If the type system is rich enough such that it comprises dependent types, i.e. types that depend on program expressions, it can be used to encode program properties (propositions-as-types). This gives rise to what is called *Type theory*: a functional language with a type system strong enough to express (second-order) predicate logic. As a consequence, the verification of program properties can be done by the type checker which undoubtedly is of huge benefit to program developers. Of course the programs will now also contain proof terms that the programmer, now also a verifier, needs to provide.
Alas, functional programming is not sufficient for all applications. The so-called "*Awkward squad*" (citation) is needed in reality, ie input/output, state, references, exceptions, concurrent threads. These are examples of (impure) side effects, in functional programming modelled via monads (see Moggi/Wadler).
Our primary objective is to have a dependent type theory that "includes" such side effects (in a controlled way). We restrict to side effects caused by monotone resources like heap without deallocation or threads with possibility to spawn new threads but not kill any threads.
The major research question underlying this proposal is: **How can side effects on monotone resources be modelled inside type theory?** There are two approaches to fix this problem:

- use Hoare Type theory (references needed) where effects and operations are *added* to the type theory and *axioms are postulated* all of which need external justification. A Hoare triple type then is added that expresses the behaviour of the operations with effects. In this approach one does not build side effects into the type theory, one adds them (disadvantages?)
- implement effects and provide functional specifications for programs with effects along the lines of Wouter Swierstra's work (citation). Here the effect is simulated using concrete data types. However, in this approach dependency on resources needs to be modelled explicitly. Moreover, type theory is total and so some "smart constructors" (citation) are needed to encode everything nicely. The encodings are tedious and not re-usable. Swierstra says: *"Furthermore, the automatic weakening of references requires a decidable equality on our universe. This excludes references storing dependent types, such as dependent pairs or dependent functions. It would be interesting to investigate how to remove this restriction and better support the automatic weakening of functions that are polymorphic with respect to the shape of the heap, such as the inc function above."* (citation from his diss).

In this project we are looking to improve on the second approach. We will try to develop an elegant *“embedded’ solution*, an extension of core type theory with monotonic resources the semantics of which extends conservatively the standard semantics of type theory using an abstract data type that hides the implementation (ie the resource) but provides an interface to access the resource. We propose to do this first, by means of a representative example, for the local state monad. We will provide an adequate specification of the local state monad within dependent type theory and a presheaf model for it to ensure soundness. This kind of model is particularly apt as (a) presheaves are adequate abstractions of the kind of dependency on a “world” which in our case is the resource (eg. the local state) in a monotone way, and (b) there already exist standard categorical models for (dependent) type theory where the category of types can be suitable instantiated by pre sheaf categories. Categorical semantics delivers for free a rich toolbox of concepts and theorems for our models.
In a dependent type they with monotone resources reasoning about functional programs with resources (side effects) will be possible keeping all the useful and well-loved advantages of type theory (which are?). An additional benefit is that the development of such an extension via a “presheaf application” can be regarded and maybe established as methodology for potentially other extensions of type theory.

Background

Functional Programming

The **Functional Programming** paradigm [?] presents computation as the evaluation of mathematical *expressions* built by the application of functions, rather than the execution of *commands* which change the state of a machine. Functional languages thus abstract away from operational details, allowing programs to exhibit much more clearly their conceptual structure. Modern functional languages like Haskell [?] and OCaml [?] possess very powerful mechanisms for abstraction and re-use, facilitating brevity, clarity, reliability and productivity. At the same time, they possess production-quality compilers, most notably the Glasgow Haskell Compiler (GHC) and the Objective Caml system, with substantial libraries capable of sustaining a growing programmer base in practical application development. As computer systems become more complex and distributed, functional programming’s clarity and hygiene become ever more significant virtues, one of the reasons why the designers of mainstream languages such as Java, C# and Visual Basic are increasingly adopting ideas from the functional world, e.g. polymorphism to support the safe reuse of parametric software components.
Over the years, the rôle of *types* in functional programming has become increasingly significant. In the landmark Hindley-Milner system, types are fully inferred by machine [?] and they in no way determine the operational behaviour of programs—the assignment of a type merely confirms that a program will not ‘go wrong’ in an especially disorderly manner. More recently, languages like Haskell have supported type systems which drop full mechanical type inference, allowing explicit types to express more subtle design statements than a machine could determine for itself. Moreover, type-directed programming approaches [?, ?] depend crucially on type information to determine what programs mean.

Dependent Types

Dependent types are types which refer to (hence depend on) data. A typical example is the type of vectors: we write $\mathrm{Vec}\,A\,n$ for the type of length n sequences of type A items. Vectors refine the ubiquitous types of lists $[A]$. Conventionally, a program idM computing the identity matrix, represented as a sequence of sequences of real numbers, of a given finite dimension would have the type $\mathrm{idM} : \mathbb{N} \to [[\mathbb{R}]]$. In a dependently typed language the same program could be given the more informative type $\mathrm{idM} : (n : \mathbb{N}) \to \mathrm{Vec}\,(\mathrm{Vec}\,\mathbb{R}\,n)\,n$, clearly indicating the relationship between the value of the input and the structure of the output; and documenting the fact that idM always returns square matrices.
Dependent types have their origin in the constructive foundations of mathematics [?] developed by the Swedish philosopher and mathematician Per Martin-Löf, who showed that the *Curry-Howard* isomorphism between simple types and propositional logic naturally extends to an isomorphism

between dependent types and higher-order constructive logic. Exploiting this equivalence between logical propositions and types, dependent types have been used as the basis for proof assistants — the most advanced system developed in Europe, Coq [?], has been used to formalize Mathematics (e.g. the proof of the four colour theorem [?]) and to verify programs (e.g. correctness of a C compiler [?, ?, ?]).

Dependently Typed Programming

Currently, the development of high quality software is more an art than a science. Types are used to enforce basic sanity conditions but their scope is limited and not easily expandable without moving to a different language, or a typed domain specific language. Formal certification, if done at all, is performed independently of the development effort and is often limited in scope by relying on fully automatizable methods, while on the other side complete formal verification of an existing software system seems currently unfeasible.
Dependent types provide programmers with language to communicate the design of software to computers and expose its motivating structure. With a type system that doubles as a logic, programmers are free to negotiate their place in the spectrum of precision from basic memory safety to total correctness. This *pay as you go* approach allows us to raise hygiene standards in programming and improve guarantees (e.g., removing tags and tests on validated data) but to stop when requirements become too severe. Such expressive types can also contribute to software engineering methods and processes through interactive, type-directed development environments.
During the past decade we have progressed considerably towards the goal of exploiting dependent types in functional programming. Augustsson's Cayenne [?] showed how functional programmers could use dependent types to their advantage by giving static types to conventionally untypable programs line `printf` or `scanf`, or by implementing a tageless interpreter exploiting type information to avoid dynamic type checking at runtime. The torch was carried further by McBride's implementation of Epigram [?], with funding from EPSRC (*Epigram: Innovative Programming with Inductive Families* - GR/N72259), introducing both an implementation of Wadler's views [?] within a dependently typed framework, and an interactive IDE using types to guide the implementation process. On the subsequent EPSRC project (*Observational Equality For Dependently Typed Programming* - EP/C512022/1), Thorsten Altenkirch and Conor McBride made significant progress on one of the thorniest technical problems for dependently typed programming—how to represent equality [?]. More recently, Norell implemented Agda [?, ?], a more scalable implementation of dependently typed programming, strongly influenced by Cayenne and Epigram.
Recent developments in functional programming languages provide a more conservative approach to dependently typed programming, by insisting on a distinction between the language used at compile time and at run time. Examples for this development are Haskell's recently acquired Generalized Algebraic Datatypes, further rationalized in Sheard's Ωmega [?, ?] and Xi's ATS [?]. Other similar developments are also taking place in the US: Concoqtion [?, ?] uses Coq's Type Theory as a compile time language, Stump's system Guru [?] based on his proposal for Operational Type Theory which separates the notions of evaluation for proofs and for programs, and Morrisett's proposal for Ynot [?, ?], based on Hoare Type Theory [?] which uses ideas from separation logic to encapsulate effects.

Dependent types and effects

Functional programming languages like Haskell can now deal very well with the so-called "*Awkward squad*" (citation) is needed in reality, ie input/output, state, references, exceptions, concurrent threads. Indeed one can argue that pure functional programming is better suited to integrate new notions of effects such as the transactional memory model because it doesn't have a built-in notion of effect as conventional imperative langauges do.
However, when moving to dependent types where programs may appear in types it is not obvious how to extend Haskell's monadic approach to IO. Haskell's IO monad is opaque - it has no compile time behaviour. Indeed, we don't want effects to be executed at compile time.
Hoare Type Theory (references needed) bypasses this issue by adding types and operations and by postulating axioms. The notion of a monad is extended to include Hoare triples so that the behaviour of operations can be specified. The HTT approach fits very naturally in the Coq approach

to type-theoretic program development where program and proofs are separated. Using HTT already an impressive suit of examples of verified effectful programs has been verified using the HTT approach.
The HTT approach fits less well with DTP, where programs and proofs are not explicitely separated but tightly integrated. Postulates are generally rejected in this context because they lead to programs which do not compute. Altenkirch and Wouter have developed an alternative: *functional specification of effects*. Here the compile time semantic of effects is given by a functional program, e.g. state is modelled using the pruely functional state monad at compile time but is executed using real memory cells at run-time. This way it is possible to reason about stateful programs and run them. The beauty of this approach is that it doesn't require any extensions to Type Theory and it also comes with its own soundness proof. Sure, we still need to ensure that our compile-time speicfication agrees with the actual run-time behaviour but this issue arises in any approach.
The Swierstra-Altenkirch approach hits a serious obstacle when ddealing with ubiquitous monotone resources. By monotone resources we mean resources which can be explicitely alloacted but never need to be deallocated - this is usually doen by a garbage collector. In Haskell monotone resources can be dealt with easily becuase we know that resources can only be generated by the appriate allocation operation. However, we cannot model this fact in Type Theory because it relies on a closed world asssumption. Swierstra addressed this problem by introducing an explicitly modelling dependency on resources. Moreover, type theory is total and so some "smart constructors" (citation) are needed to encode everything nicely. The encodings are tedious and not re-usable. Swierstra says: *"Furthermore, the automatic weakening of references requires a decidable equality on our universe. This excludes references storing dependent types, such as dependent pairs or dependent functions. It would be interesting to investigate how to remove this restriction and better support the automatic weakening of functions that are polymorphic with respect to the shape of the heap, such as the inc function above."* (citation from his diss).

Presheaf models of Type Theory

In the past functional programming has benefitted by importing concepts from Category Theory, the ubiquitous use of monads is the only most famous example for this phenomen. The present issue - how to deal effectively with monotone resources in dependently typed programming - is yeat another example. Indeed we know how to deal with monotine resources in denotational semantics from the work of Levy and Plotkin/Power (citation): we model local state using a possible world semantics (or more technical a presheaf semantics) where the worlds correspond to the current allocation of resources. The monotonicity of resources correspods to the functoriality: and value in a given world is still valid in any extension which corresponds to additional resources which have been allocated in the mean-time.
Whileit is clear in principle that any dependently typed program can be given a presheaf semantics and that with this interpretation monotone resources can be given a functional specification, it is not yet clear how to use this idea in practice. What does it mean to interpret a given program or proof in a presehaf semantics? How can objects in this presheaf workd interact with ordinary objects? And how can we combine different notion of monotone effects which lead to different presehaf interpretations in one program?

National Importance

This research concerns the improvement of language support for the production of certified programs and thus fits perfectly in the EPSRC's "Verification and Correctness" theme. The development of languages and tools to support the production of correct software has numerous applications in safety critical systems, medical devices and even operating systems (*refer to Gerwin's verified OS? Msoft's verified device drivers?*). The EPSRC has earmarked this area for future growth and currently funds verification related projects to the extent of GBP 14 million. UK academics are world leaders in this field. Larger centres of excellence for DTP in the UK are Nottingham (Altenkirch, grant "Reusability and Dependent Types"), Oxford, and Strathclyde. Both investigators are involved in national and international collaborations with other leaders in DTP and Program Verification (*explain*) Further international leaders in the field are G. Morrisett (Harvard) and S. Weirich (UPenn) in the US (see Trellys project), W. Swierstra (Utrecht, Netherlands),

T. Streicher (Darmstadt) and M. Hofmann (LMU Munich) in Germany and the programming logic research group in Chalmers (Gothenburg, Sweden).
The success of the proposed research programme will have potential impact not only on the DTP community but also on the design of functional programming languages with rich type systems in general. The research outcomes are expected to enable programmers to enjoy the advantages of functional programming and effectful programming at the same time without losing the ability to prove programs correct. Long term it is expected that such languages will become more popular and more widely used in software engineering (world wide), potentially reducing the economic loss caused by malfunctioning software (estimated around 150$ in the US alone by by David Rice 2009, source: html).

Academic Impact

Describe how the research will benefit other researchers in the field and in related disciplines, both within the UK and elsewhere.
clear for the academic community but further afield?
What will be done to ensure that they can benefit? Explain any collaboration with other researchers and their role in the project.
For each Visiting Researcher, set out why they are the most appropriate person, and what they will contribute to the project.
visiting researchers? Who shall we nominate?
Wouter Swierstra, Greg Morrissett? Thomas Streicher, Jeremy Gibbons?
We should then say which work packages they will contribute for. Project partners should send letter of support as far as I understand.
Wed on;t have any industrial partners/visitors. is that a problem?

Research Hypothesis and Objectives

Set out the research idea or hypothesis. Explain why the proposed project is of sufficient timeliness and novelty to warrant consideration for funding. Identify the overall aims of the project and the individual measurable objectives against which you would wish the outcome of the work to be assessed.
Our hypothesis: we can safely embed monotone resources in type theory and this leads to elegant programs that make side effects amenable to the functional programmer.
Objectives

- develop the syntactic language for the extension of dependent type theory with resources
- develop the pre sheaf models for soundness
- implement (what?) in Agda
- prove the usability doing examples like local state (and what else?)

Programme and Methodology

Detail the methodology to be used in pursuit of the research and justify this choice. Describe the programme of work, indicating the research to be undertaken and the milestones that can be used to measure its progress. The detail should be sufficient to indicate the programme of work for each member of the research team. Explain how the project will be managed.
We detail the work packages. Before we need to point out that we use the local state monad as guiding example, that we use pre sheaf semantics as we know it can deal with such monotone resource already.

Work Plan

one page, they like GANTT or PERT charts.

1. Define Presheaf semantics for core Type Theory
2. Model the monotone resources in the above
3. Agda implementation
4. Examples (like IORef)
5. Develop generic interface for Type Theory with monotone resources by reflection of semantics
6. Investigate combination of various resources and resource models

Task list

1.
2.
3.

1 Justification of Resources

up to 2 pages

1.1 Directly Incurred Costs

Staff We request two Research Assistants (scale 6 or 7(?)) to help with ? **need to make twos fit with the workplan.** Ideally, the first RA would have some experience in one or several of the following: (dependent) type theory, effect systems, category theory, sheaf semantics. The second RA would ideally have a good grasp of functional programming, ideally with Agda and or Coq and good knowledge about side effects and/or category theory.

Travel and Subsistence To present (partial) results and keep contact to other researchers in the same or related area support for attendance of 5 international workshops or conferences for both PIs and RFs e.g. ETAPS, POPL, LICS, CSL, ICALP, Types Workshops as well as national meetings such as BTCS and "Fun in the Afternoon". The list of conferences is provisional, precise plans cannot be made until the details of the conferences' locations, dates and programmes are known. Note that we calculate an average cost of 900*$*but some trips inside Europe will be cheaper but this is compensated by trips to Asia or the US which will be more expensive. (4 times 5 times *$*1,000 = *$*20,000).
To establish communication between the two sites involved trips between Sussex and Nottingham are requested as follows: 4 times 4 one day day visits by each member (*$*70 times 16 = *$*1,120) and 3 times 4 two day visits (6 times *$*150 = 900*$*).
Also included are short trips within the UK for an exchange with researchers working in the same area, e.g. Birmingham (Levy), London (?) (10 times 80*$*plus 10 times 18*$*= 980*$*).
For each Visiting Researcher we plan two one week (or one two week) stays at one of the project sites. Only travel and subsistence need to be budgeted for each of their stays (n times 600*$*= ?).
In order to maximise impact and for networking purposes we plan to organise a small specialist workshop in the topic. For this we request *$*2000 to to pay for travel and subsistence of speakers and *$*500 for coffee and meals.

Consumables The two RAs will require a PC. The University of Nottingham and Sussex, resp., do not provide machines for RAs so two such PCs (*$*1000 each) are requested.

1.2 Directly Allocated Costs

Investigators will work about 7 to 8 hours a week?
?

1.3 Exceptions

2 Pathways to Impact

up to 2 pages
Use this annex to the proposal to describe activities that can be undertaken during the project to accelerate the route to the identified benefits being realised; shortening the time between discovery and use of knowledge. Also identify the additional resources needed to undertake these activities.
In summary, the document should describe the kinds of impact envisaged, how the proposed research project will be managed to engage users and beneficiaries and increase the likelihood of impacts, including (wherever appropriate): Methods for communications and engagement
Collaboration and exploitation in the most effective and appropriate manner
The project teams track record in this area
The resources required for these activities.
Please ensure these are also captured in the financial summary and the Justification of Resources.