04381 Abstracts Collection
Dependently Typed Programming
— Dagstuhl Seminar —

Thorsten Altenkirch[1], Martin Hofmann[2] and John Hughes[3]

[1] University of Nottingham, GB
txa@cs.nott.ac.uk
[2] LMU München, DE
hofmann@ifi.lmu.de
[3] Chalmers TU, Göteborg, SE
rjmh@cs.chalmers.se

Abstract. From 12.09.04 to 17.09.04, the Dagstuhl Seminar 04381 "Dependently Typed Programming" was held in the International Conference and Research Center (IBFI), Schloss Dagstuhl. During the seminar, several participants presented their current research, and ongoing work and open problems were discussed. Abstracts of the presentations given during the seminar as well as abstracts of seminar results and ideas are put together in this paper. The first section describes the seminar topics and goals in general. Links to extended abstracts or full papers are provided, if available.

04381 Summary – Dependently Typed Programming

The Dagstuhl seminar (04381) on Dependently Typed Programming brought together researchers from all over the world who are interested in the use of dependent types in programming. An emerging topic was the interaction of the functional programming community and the Types community: an example is the use of GADTs in Haskell, which represent a restricted use of dependent types in Haskell while on the other hand proof systems like COQ in which allow the expression of many functional programming idioms. Emerging languages and systems, like Epigram, attempt to unify functional programming and Type Theory based proof development environments. Discussions during the seminar centred on the question how to integrate dependent types in real programming languages and on the pragmatic and theoretical questions raised by doing this.

Joint work of: Altenkirch, Thorsten; Hofmann, Martin; Hughes, John

Dagstuhl Seminar Proceedings 04381
Dependently Typed Programming
http://drops.dagstuhl.de/opus/volltexte/2005/186

Data vs. codata

Thorsten Altenkirch (University of Nottingham, GB)

In a total language we have to distinguish data and codata. I'll explain codata using a mirror: while data is defined by a producer contract, i.e. the producer promises only to use the agreed constructors to construct data, codata is defined by a consumer contract, i.e. the consumer promises only to use pattern matching to analyze the codata. These principles give rise to recursion - corecursion and induction - coinduction. I explain how coinductive reasoning can be explained using a coinductive definition of equality.

Filters on Co-Inductive streams; a study on the example of Eratosthene's sieve

Yves Bertot (INRIA - Sophia Antipolis, F)

Filters are functions from streams to streams that collect data satisfying a given predicate. Formalizing filters in Co-Inductive programming is difficult because they are not plain co-recursive function. We describe a solution that combines ideas from temporal logic, recursion on an ad-hoc predicate, uses of dependent types to describe partial functions, and advanced use of guardedness conditions. This solution was completed using Coq 8.0.

Keywords: Co-induction, dependent types to describe partial functions, general recursion.

Formalising Bitonic Sort using Dependt Types

Ana Bove (Chalmers UT - Göteborg, S)

Bitonic sort (Batcher, 1968) is one of the fastest sorting networks. A sorting network is a special kind of sorting algorithm, where the sequence of comparisons is not data-dependent. This makes sorting networks (and in particular bitonic sort) suitable for implementation in hardware or in parallel processor arrays.

Although the algorithm is short and computationally rather simple, we face two problems when we want to use type theory for its formalisation:

1) the algorithm is not structurally smaller;

2) its correctness proof is not very intuitive, hence its formalisation is not straightforward.

In this talk we present a formalisation of bitonic sort and its correctness proof in constructive type theory.

Here, the correctness of bitonic sort is developed on the basis of the 0-1-principle (Knuth, 1973). For bitonic sort, the 0-1-principle states that if bitonic sort sorts every sequence of 0's and 1's then it sorts every sequence of arbitrary values.

Computation by Judgement Rewriting

Venanzio Capretta (University of Ottawa, CDN)

I will shortly explain two methods to implement general recursion in type theory: (1) by a coinductive model of potentially diverging comutation; (2, with Ana Bove) by and inductive characterization of the domain of an algorithm.

The second method gives rise to some problem when interpreting higher order functionals. I will discuss a solution that exploit the power of impredicative type theory. Since the method relies on Dybjer's induction-recursion, I will also discuss a representation of this contruct in the Calculus of Constructions.

Both methods (1) and (2) represent functions, but do not allow direct computation by term reduction. However, in both cases, an effective computation model can be realized by a notion of judgement rewriting, consisting in a goal-directed refinement algorithm for sequents starting from the assumption of a variable as result of the computation.

Keywords: General Recursion, Impredicativity, Judgement Rewriting

Dependent Types for Update Programming

Martin Erwig (Oregon State University, USA)

Cells in spreadsheets that serve as headers can be interpreted as unit information for other cells to constrain the operations that are allowed to be applied. Units essentially refine types and thus are an example of dependent types.

We can define a unit system for end-user spreadsheets that is based on the concrete notion of units instead of the abstract concept of types. The unit system contains concepts, such as dependent units, multiple units, and unit generalization, that allow the classification of spreadsheet contents on a more fine-grained level than types do. Also, because communication with the end user happens only in terms of objects that are contained in the spreadsheet, this system does not require end users to learn new abstract concepts of type systems.

Since the unit inference depends on information about headers in a spreadsheet, a realistic unit inference system requires a method for automatically determining headers. We sketch several spatial-analysis algorithms for header inference.

The combined header- and unit-inference system is fully integrated into Microsoft Excel and can be used to automatically identify various kinds of errors in spreadsheets. Test results show that the system works accurately and reliably.

Preservation of Typing for the Domain-Free Calculus of Inductive Constructions with Implicit Parameters

Benjamin Gregoire (INRIA - Sophia Antipolis, F)

In a proof system like Coq, checking the validity of a proof involves comparing types modulo beta-conversion, which is potentially a time-consuming task. To speed up this conversion test, Grégoire and Leroy showed how proof terms could be compiled towards a slightly modified Objective Caml bytecode. But compilation erases most of the type information carried by the proofs. It was an open problem to show that erasure does not change the underlying formalism. In this paper, we show the equivalence of the formalism implemented by the proof compiler and the official formalism of Coq.

The interest for such a result goes beyond the mere correctness proof of a compilation scheme since even non compiled implementations can be made more efficient by not comparing type annotations. It is also a significant generalisation and strengthening of a similar result (Preservation of Equational Theory) by Barthe and Sørensen on the class of Domain-free pure type systems.

Dependently Typed Programming and the Coq Proof Assistant

Nicolas Magaud (Univ. of New South Wales, AU)

Coq is a proof assistant based on the calculus of inductive constructions. Among other features such as polymorphism, higher-order and primitive inductive definitions, it provides dependent types. These dependent types can be used in a wide range of applications.

On the one hand, dependent types are used to achieve function definitions.

Such functions usually would not feature dependent types if written in a functional language like Caml or Haskell. In particular, they allow the user to transform partial functions into total ones by restricting the range of its inputs, e.g. from fact:int->int in a functional programming language we get fact:forall n:int, n>=0 -> int. In addition to that, dependent types are useful to define functions by well-founded induction. In this case, they are used to make sure one does only make recursive calls on smaller terms according to the considered relation.

On the other hand, we can view Coq as a functional programming language and actually use dependent types to specify the type of programs. Instead of defining a function with a simple type and later prove as lemmas some of its properties, we can refine the type of the function to make it more informative regarding the properties of the function. This leads to fully-specified functions whose type carries the function actual specification.

Finally, dependent data types such as vectors (dependent lists) can be defined. Coq provides means to write dependently typed programs such as reverse

rev:forall n:nat, (vect n) -> (vect n) and tools such as dependent equality to reason about them.

All this shows Coq is a suitable environment to write dependently typed programs. In addition, it is a framework in which you can even prove properties of the dependently typed programs one has defined and also extract these "then certified" programs to efficient functional programming languages like Haskell or Caml.

Keywords: Dependent types, Coq, well-founded recursion, formal proofs

Towards iteration for truly nested datatypes

Ralph Matthes (Universität München, D)

A case study is presented how one can predicatively justify nested datatypes with a nested recursive call in the datatype definition. The example is a representation of lambda terms including explicit substitutions by the equation

Lam A = A + Lam A x Lam A + Lam(1+A) + Lam(Lam A). (*)

Following an idea of Anton Setzer and a handwritten manuscript by Peter Aczel (Edinburgh, March 2003), Lam can be introduced by an inductive family of types, indexed over finite lists of booleans - a predicative construction. It can be shown (done by Peter Aczel) that Lam is a minimal pre-fixed point of the rank-2 operator underlying equation (*). The new insight is that this can even be used to program an iterator for Lam (in our case, within system Coq). However, its operational behaviour is hard to express in an implementation-independent fashion. A possible solution comes from a syntactic form of Kan extensions that reintroduces impredicativity.

The use of Lam is demonstrated by a representation of beta-developments of untyped lambda terms.

Wobbly types: type inference for generalised algebraic data types

Simon L. Peyton-Jones (Microsoft Research UK, GB)

Generalised algebraic data types (GADTs), sometimes known as "guarded recursive data types" or "first-class phantom types", are a simple but powerful generalisation of the data types of Haskell and ML.

Recent works have given compelling examples of the utility of GADTs, although type inference is known to be difficult.

It is time to pluck the fruit. Can GADTs be added to Haskell, without losing type inference, or requiring unacceptably heavy type annotations? Can this be done without completely rewriting the already-complex Haskell type-inference engine, and without complex interactions with (say) type classes? We answer

these questions in the affirmative, giving a type system that explains just what type annotations are required, and a prototype implementation that implements it. Our main technical innovation is "wobbly types", which express in a declarative way the uncertainty caused by the incremental nature of typical type-inference algorithms.

Joint work of: Simon Peyton Jones, Stephanie Weirich, Geoffrey Washburn

Certified Typechecking in Foundational Certified Code Systems

Susmit Sarkar (CMU - Pittsburgh, USA)

Certified code enable untrusted programs to be proven safe to execute in a machine-checkable manner. This is done by packaging a proof of code safety together with the code. Recent work has focused on building foundational certified code systems, where safety is defined relative to a concrete machine architecture. To be practical, such systems factor the proof of program safety into two parts. The first part is a generic part, in which a class of programs is proven safe. In the second, program-specific part, the program of interest is shown to belong to the class. If the class of programs is defined as a type system, the first problem corresponds to proving type safety, and the second problem to type checking.

Previous work, including ours, have focussed on the first problem. The second problem is less well studied. A typechecker for the untrusted program has to be provided by untrusted sources. It may not implement the type system correctly. We need a certifying typechecker, which can be statically checked. We propose a dependently-typed version of SML to write such typecheckers. Dependent types help us check partial correctness statically.

Keywords: Foundational certified code

Delphin: Functional Programming with Dependent Types

Carsten Schürmann (Yale University, USA)

Logical frameworks are meta languages designed to represent deductive systems, such as derivation in logic and type theories, or traces of operational semantics and translation procedures. Functional programming languages are designed to help programmers express computations. Delphin aims at combing the two and provides a programming paradigm for writing programs with higher-order, dependently typed encodings. in a natural way as if they were natural numbers lists or trees.

Delphin is a two level programming language that distinguishes between a universe for representation and a universe for computation.

As a novelty it would be useful to mentation the injection type embedding data into the computation level and a modal type constructor that permits evaluation under lambda binders and ensures scoping. A version for the simply typed logical framework is implemented and accessible thorough our homepage http://www.cs.yale.edu/Carsten/delphin.

Keywords: Delphin, Logical Frameworks, Higher-order Abstract Syntax, Functional Programming, Dependent Types.

Joint work of: Schürmann, Carsten; Poswolsky, Adam; Sarnat, Jeff

Interactive Programs and Weakly Final Coalgebras in Dependent Type Theory (Extended Version)

Anton Setzer (University of Wales - Swansea, GB)

We reconsider the representation of interactive programs in dependent type theory that the authors proposed in earlier papers. Whereas in previous versions the type of interactive programs was introduced in an ad hoc way, it is here defined as a weakly final coalgebra for a general form of polynomial functor. The are two versions: in the first the interface with the real world is fixed, while in the second the potential interactions can depend on the history of previous interactions. The second version may be appropriate for working with specifications of interactive programs.

We focus on command-response interfaces, and consider both client and server programs, that run on opposite sides such an interface. We give formation/introduction/elimination/equality rules for these coalgebras. These are explored in two dimensions: coiterative versus corecursive, and monadic versus non-monadic. We also comment upon the relationship of the corresponding rules with guarded induction. It turns out that the introduction rules are nothing but a slightly restricted form of guarded induction. However, the form in which we write guarded induction is not recursive equations (which would break normalisation – we show that type checking becomes undecidable), but instead involves an elimination operator in a crucial way.

Keywords: Dependently types programming, interactive programs, coalgebras, weakly final coalgebras, coiteration, corecursion, monad

Joint work of: Setzer, Anton; Hancock, Peter

Full Paper: http://drops.dagstuhl.de/opus/volltexte/2005/176

Partiality is an effect

Tarmo Uustalu (Tallinn Technical University, EE)

Partial functions are usually considered as something basic or "purely functional" in functional languages, hence semantics starts from CPO-like categories with fixpoint operators. We maintain that partiality due to non-termination can alternatively be treated as a monadic effect. The appropriate monad is the free completely iterative monad on the identity functor which captures timed, possibly non-terminating computation; one can also consider the quotient that identifies all terminating computations yielding the same value (constructively, this is not the same as the error monad, which can only capture partiality due to finite failure). Looping constructions are supported immediately; we discuss general recursion and combination of the monad with monads for other effects. Time permitting, I will also show how the dual comonad can be used to represent causal stream functions. (Work in progress jointly with T. Altenkirch and V. Capretta.)

A Core Language for Generalized Algebraic Datatypes

Stephanie Weirich (University of Pennsylvania, USA)

Generalised algebraic data types (GADTs), sometimes known as "guarded recursive data types" or "first-class phantom types", are a simple but powerful generalisation of the data types of Haskell and ML. Recent works have given compelling examples of the utility of GADTs, although type inference is known to be difficult.

Simon Peyton Jones will be speaking about our work in combining GADTs with Haskell type inference. However, before we could even think about doing that, we needed to define the semantics of GADTs in Haskell's explicitly-typed core language. Although the semantics of GADTs in explicitly-typed languages is not new, we did encounter a few technical difficulties in designing a version that worked well with Haskell's existing core language. I'll speak about these issues as well as provide material background for Simon's talk.