Dagstuhl Seminar 01341

Dependent Type Theory Meets Practical Programming

19.08.2001–24.08.2001

organized by

Gilles Barthe (INRIA, Sophia Antipolis),

Peter Dybjer (Chalmers, Göteborg), and

Peter Thiemann (Freiburg)

Modern programming languagees rely on advanced type systems that detect errors at compile-time. While the benefits of type systems have long been recognized, there are some areas where the standard systems in programming languages are not expressive enough. Language designers usually trade expressiveness for decidability of the type system. Some interesting programs will always be rejected (despite their semantical soundness) or be assigned uninformative types.

There are several remedies to this situation. We argue that dependent type systems, which allow the formation of types that explicitly depend on other types or values, are one of the most promising approaches. These systems are well-investigated from a theoretical point of view by logicians and type theorists. For example, dependent types are used in proof assistants to implement various logics and there are sophisticated proof editors for developing programs in a dependently typed language.

To the present day, the impact of these developments on practical programming has been small, partially because of the level of sophistication of these systems and of their type checkers. Only recently, there have been efforts to integrate dependent systems into intermediate languages in com-

pilers, for example, the TAL compiler (Morrisett and others[1]), and actual programming languages, for example, Cayenne (Augustsson[2]) and DML (Xi and Pfenning[3]). Additional uses have been identified in high-profile applications such as mobile code security. For example, proof carrying code (Necula and Lee[4]) relies on a dependently typed lambda calculus to encode proof terms.

Now the time is ripe to bring together researchers from the two communities (type theorists and programming experts), and to further cross-fertilization of ideas, techniques and formalisms developed independently in these communities. In particular, the seminar shall make researchers in programming languages aware of new developments and research directions on the theory side; point out to theorists practical uses of advanced type systems and urge them to address theoretical problems arising in emerging applications.

The need for such a seminar became clear during the first international Workshop on Dependent Types in Programming[5], held in Göteborg in March 1999. A second international Workshop on Dependent Types in Programming[6] has been held in Ponte de Lima in July 2000, but it is hard to discuss the problems pointed out above in a one-day workshop.

[1] `http://www.cs.cornell.edu/home/jgm/`
[2] `http://www.cs.chalmers.se/~augustss/`
[3] `http://www.ececs.uc.edu/~hwxi/`
[4] `http://www-nt.cs.berkeley.edu/home/necula/public_html/`
[5] `http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/dtp99.html`
[6] `http://www-sop.inria.fr/oasis/DTP00/index.html`

Contents

1 Principal Typings Demystified: What They Are, Why You Want Them, and Why Your Type System Doesn't Have Them[7]

Joe Wells, Heriot-Watt-University, Edinburgh (GB)

Let S be some type system. A typing in S for a typable term M is the collection of all of the information other than M which appears in the final judgement of a proof derivation showing that M is typable. For example, suppose there is a derivation in S ending with the judgement $A \vdash M : \tau$ meaning that M has result type τ when assuming the types of free variables are given by A. Then (A, τ) is a typing for M.

A principal typing in S for a term M is a typing for M which somehow represents all other possible typings in S for M. It is important not to confuse this notion with the weaker notion of principal type often mentioned in connection with the Hindley/Milner type system. Previous definitions of principal typings for specific type systems have involved various syntactic operations on typings such as substitution of types for type variables, expansion, lifting, etc.

This talk presents a new general definition of principal typings which does not depend on the details of any particular type system. This talk shows that the new general definition correctly generalizes previous system-dependent definitions. This talk explains why the new definition is the right one. Furthermore, the new definition is used to prove that certain polymorphic type systems using "for all" quantifiers, namely System F and the Hindley/Milner system, do not have principal typings.

2 Rules for Final Coalgebras in Dependent Type Theory

Anton Setzer, University of Wales, Swansea (GB)

with Peter Hancock, Edinburgh (GB)

We review some standard approaches for formulating interactive programs in functional programming languages, and introduce our approach. In our setting, we represent interactive programs as possibly non-wellfounded trees with nodes labeled by interactive commands and having branching degree over the response set corresponding to this command. We then raise the problem, namely the need for rules for final coalgebras in dependent type theory. Next, we show that elements of the final coalgebras corresponding to interactive programs can be introduced as graphs, labeled by commands

[7] A corresponding paper is available at `http://www.cee.hw.ac.uk/~jbw/papers/`.

and with arrows for every element of the corresponding response set into another node. On this basis we define rules corresponding to coiteration and corecursion. We indicate, why the successor for the co-natural numbers is difficult to compute using coiteration and results in high complexity, whereas with corecursion this is simple, similar to the fact that the predecessor is difficult to compute for the natural numbers using iteration, but easy using recursion. Finally we introduce constructions for defining elements of the final coalgebras: while- and repeat loops, a fixed-point construction and redirect, which allows to translate programs in a highlevel language into a low level language by replacing commands of the high level language by programs in the low level language. Redirect allows to refine programs in a top down approach and to build libraries.

3 A Type System for Certified Binaries

Zhong Shao, Yale University, New Haven (USA)

A certified binary is a value together with a proof that the value satisfies a given specification. Existing compilers that generate certified code have focused on simple memory and control-flow safety rather than more advanced properties. In this talk, we present a general framework for explicitly representing complex propositions and proofs in typed intermediate and assembly languages. The new framework allows us to reason about certified programs that involve effects while still maintaining decidable typechecking. We show how to integrate an entire proof system (the calculus of inductive constructions) into a compiler intermediate language and how the intermediate language can undergo complex transformations (CPS and closure conversion) while preserving proofs represented in the type system. Our work provides a foundation for the process of automatically generating certified binaries in a type-theoretic framework. This is joint work with Bratin Saha, Valery Trifonov, and Nikolaos Papaspyrou.

4 Generic Unification Generalized, or Datatypes Parameterized over Typerecords[8]

Magnus Carlsson, Oregon Graduate Institute, Beaverton (USA)

We discuss a programming exercise in which we have generalized Tim Sheard's generic unification algorithm (ICFP'01) to cope with mutually recursive term datatypes. The programming language of choice is Cayenne, where we

[8]The slides are available at `http://www.cse.ogi.edu/~magnus/dagstuhl2001/unify.ps.gz`.

use dependent types to encode records of types for the recursive parameter in the term datatypes.

5 Designing Reliable, High-Performance Networks with the Nuprl Proof Development System[9]

Christoph Kreitz, Cornell University, Ithaca (USA)

Formal methods tools have greatly influenced our ability to increase the reliability of software and hardware systems. Extended type checkers, model checkers and theorem provers have been used to detect subtle errors in prototype code and to clarify critical concepts in system design. Automated theorem proving now has the potential to support a formal development of reliable systems at the same pace as designs that are not formally assisted, provided it is engaged at the earliest stages of design and implementation.

An engagement of deductive methods at this stage depends on a formal language that is able to naturally express the ideas underlying the software systems, a knowledge base of formalized facts about systems concepts that a design team can use in its discussions, and a theorem prover capable of integrating a variety of different proof techniques while providing assurance for the correctness of the joint result.

The Nuprl Logical Programming Environment provides an expressive type theory and a theorem proving environment for the development of verified algorithmic knowledge and is capable of supporting the formal design and implementation of high-performance network systems.

The presentation briefly discusses aspects of Nuprl's type theory as well as the system's features and architecture. We also describe how the Nuprl LPE was used in the verification of protocols for the Ensemble group communication toolkit, in verifiably correct optimizations of Ensemble protocol stacks, and in the formal design and implementation of new adaptive network protocols.

6 PAL+: A Lambda-free Logical Framework[10]

Zhaohui Luo, University of Durham, Durham (GB)

A lambda-free logical framework takes parameterisation and definitions as the basic notions to provide schematic mechanisms for specification of type theories and their use in practice. The framework presented here, PAL+, is a logical framework for specification and implementation of type theories, such

[9] `http://www.cs.cornell.edu/home/kreitz/Abstracts/01dagstuhl-dependent.html`

[10] The full paper, to appear in J of Functional Programming, can be found at `http://www.dur.ac.uk/zhaohui.luo/PALPLUS.JOURNAL.ps.gz`

as Martin-Lof's type theory or UTT. As in Martin-Lof's logical framework, computational rules can be introduced and are used to give meanings to the declared constants. However, PAL+ only allows one to talk about the concepts that are intuitively in the object type theories: types and their objects, and families of types and families of objects of types. In particular, in PAL+, one cannot directly represent families of families of entities, which could be done in other logical frameworks by means of lambda abstraction.

PAL+ is in the spirit of de Bruijn's PAL for Automath. Compared with PAL, PAL+ allows one to represent parametric concepts such as families of types and families of non-parametric objects, which can be used by themselves as totalities as well as when they are fully instantiated. Such parametric objects are represented by local definitions (let-expressions).

We claim that PAL+ is a correct meta-language for specifying type theories (e.g., dependent type theories), as it has the advantage of exactly capturing the intuitive concepts in object type theories, and that its implementation reflects the actual use of type theories in practice. We shall study the meta-theory of PAL+ by developing its typed operational semantics and showing that it has nice meta-theoretic properties.

7 MLF: Raising ML to the Power of System F

Didier Remy, INRIA Rocquencourt, Le Chesnay (F)

with Didier Le Bottlan

We propose a type system MLF that generalizes ML with first-class polymorphism as in system F. We perform partial type reconstruction. As ML and as opposed to system F, each typable expression admits a principal type, which can be inferred. Furthermore, all expressions of ML are well-typed, with a possibly more general type than in ML, without any need for type annotation. Only abstraction over values that are used polymorphically must be annotated, which allows to type all expressions of system F as well.

8 Hindley-Milner with Local Constraints: Extending Type Inference the Easy Way

Jacques Garrigue, Kyoto University, Kyoto (J)

We propose a new way to mix constrained types and type inference, where the interaction between the two is minimal. By using local constraints embedded in types, rather than the other way round, we obtain a system which keeps the usual structure of an Hindley-Milner type system. In practice, this means that it is easy to introduce local constraints in existing type inference algorithms.

Eventhough our system is notably weaker than general constraint-based type systems, making it unable to handle subtyping for instance, it is powerful enough to accomodate many features, from simple polymorphic records a la Ohori to Objective Caml's polymorphic variants, and accurate typing of pattern matching (i.e. polymorphic message dispatch), all these through tiny variations in the constraint part of the system.

9 Dependent Types in Type-Directed Partial Evaluation

Andrzej Filinski, University of Copenhagen, Copenhagen (DK)

Type-directed partial evaluation, a refinement of normalization by evaluation, is based on reconstructing the syntax of normal-form lambda-terms from their meanings in a special model. However, even though the term language is typed, the usual presentation of the syntax-extraction algorithm actually works with representations of potentially untypable terms, since it is somewhat awkward to keep explicit track of the type and scope of bound variables in a term, as it is being constructed.

In this talk, we consider a dependently-typed variant of the syntax extraction procedure, which now works with a type family of explicitly well-typed and well-scoped lambda-terms, in either de Bruijn index or de Bruijn level notation. The construction is based on a Kripke-style interpretation, where the worlds correspond to typing contexts and world passage is context extension.

The straightforward expression of the dependently-typed algorithm is conceptually simple, but computationally awkward, since every passage between worlds in the Kripke interpretation requires an explicit weakening transformation of the partially constructed term. Instead, we introduce a construction based on "delayed weakening", in which a term is represented as a function that can reconstruct the appropriate concrete term with respect to any sufficiently late world. With this representation, it becomes possible to limit all uses of weakening transformations to individual free variables only – a trivial operation for de Bruijn levels, and also very simple for de Bruijn indices.

Finally we show that, by uniformly suppressing the parts of the program that only maintain typing and scoping evidence, we can recover the standard Berger-Schwichtenberg algorithm for untyped representations. In this, partially constructed terms are expressed as functions from natural numbers (effectively representing context lengths) to concrete syntax. In other words, extracting the essential computational content of the dependently-typed algorithm gives a natural justification of the efficient, untyped variant.

10 Polymorphic Lemmas and Definitions in Lambda Prolog and Twelf

Amy Felty, University of Ottawa, Ottawa (CDN)

In applications of logic to software safety and security, such as proof-carrying code, the implementation of the proof checker for the core logic is inside the trusted code base (TCB). The tools for constructing proofs, however, need not be in the TCB, because the proofs constructed by them can be checked. We show that lemmas and definitions can be implemented with a great economy of expression in Twelf and Lambda Prolog, and thus the code needed to check them does not greatly complicate the TCB. Both Twelf and Lambda Prolog are known to be well-suited for expressing and implementing logics and inference systems; Twelf implements a dependently-typed meta-language, while Lambda Prolog does not. We illustrate by encoding a higher-order logic and a lemma and definition mechanism for it in both Twelf and Lambda Prolog. The encoding maps both terms and types of the object logic (higher-order logic) to terms of the meta-language (Twelf or Lambda Prolog). We compare the features of these two meta-languages for our purposes, and discuss the possibility of using a hybrid system to implement a proof-carrying code system.

11 Inductive and Inductive-Recursive Definitions in Dependent Type Theory

Peter Dybjer, Chalmers University of Technology, Göteborg (S)

We give an introduction to the theory of inductive and inductive-recursive definitions in type theory, by showing some examples. Firstly, we show two examples of inductive definitions essentially using dependent types: (i) the type of well-orderings, that is, well-founded trees with varying branching factor, and (ii) the type of n-tuples (lists of a certain length). We also point out how the latter can be defined either inductively, by introduction rules, or by recursion on n.

We then give an example of an inductive-recursive definition, the simultaneous inductive definition of the type of lists where all elements are different, and the recursive definition of the freshness relation. The new feature here is that an introduction rule (the type of the constructor cons) refers to the freshness relation, although the latter is defined by recursion on the structure of the elements which are being generated. This feature is not present in usual inductive definitions, where structural recursion can be only be performed on an already given inductively generated structure. We also show an application due to Bove and Capretta, where inductive-recursive definitions arise in the analysis of the termination of functions defined by

nested recursion. Finally, we show how universes a la Tarski arise as special kinds of inductive-recursive definitions. More generally, as shown by Dybjer and Setzer, inductive-recursive definitions subsume a variety of powerful universe constructions, and greatly increase the proof-theoretic strength of constructive type theory.

12 Randomized Algorithms in Type Theory

Christine Paulin-Mohring, Université Paris Sud, Orsay (F)

with Philippe Audebaud, ENS Lyon

and Richard Lassaigne, Université Paris 7

We study how to specify and prove randomized algorithms in type theory. Our goal is to build an environment for reasoning on randomized algorithms on top of the Coq proof assistant. This work presents a shallow embedding of a functional language with randomized constructions into type theory and a set of rules to analyse the probabilistic behavior of programs expressed in this language.

It is known that there is a natural representation of randomized programs constructions as operations on distributions on the state. Instead to interpret an imperative program as a function from state to state, it becomes a function from distribution on state to distribution on state. Pre and post conditions are generalised to real-valued functions on state, with values in the segment [0,1]. The fact that a program satisfies a certain pre and post-condition is generalised to the fact that for any input distribution, the measure of the precondition is less than the measure of the post-condition for the output distribution of the program. The purpose of this talk is to show how to apply this technics for doing an interpretation of functional randomized algorithms inside a pure functional type theory. We show that the interpretation of programs corresponds to a monadic transformation $(\tau \to (\tau \to [0,1]) \to [0,1])$. We discuss a possible representation of [0,1] in type theory which enjoys the expected properties. We finally propose a system for reasoning on randomized programs.

13 First-Class Polyvariant Functions, Co-Arity Raising, and Threaded Specialization

Peter Thiemann, Universität Freiburg, Freiburg (D)

Tag removal is a transformation that eliminates unnecessary type tags at runtime. Tag removal is useful by itself to improve performance but it also plays a central role as a post-pass for partial evaluation of typed languages.

It is an important stepping stone to achieve (so-called) Jones-optimal specialization.

John Hughes has achieved Jones-optimal specialization using type specialization, which "somehow" removes the tags. We examine the ingredients of type specialization and identify first-class polyvariant functions, co-arity raising, and threaded specialization as the essential requirements to achieve Jones-optimal specialization. We extend a standard partial evaluator with first-class polyvariant functions, co-arity raising, and threaded specialization. We demonstrate its ability to achieve Jones-optimal specialization by specializing a typed interpreter for a simply-typed applied lambda calculus so that no run-time tagging operations remain in the specialized program. Hence, we claim that first-class polyvariant functions and co-arity raising along with threaded specialization perform a task similar to tag removal.

The main technical contributions are the specification of a generalized binding-time analysis for polyvariant functions and co-arity raising, a structural operational semantics for the corresponding specializer, and a type soundness proof of the analysis with respect to the specializer. The latter proof establishes the correctness of our implementation with respect to the specification of specialization in the form of a type system. As a new implementation technique, our partial evaluator makes essential use of threaded specialization where multiple specializations are started concurrently. Hence, our work provides an operational specification of a large and essential part of Hughes's type specialization.

14 Programming with Dependent Types: Perspectives and Problems

Randy Pollack, University of Edinburgh, Edinburgh (GB)

I like programming with dependent types; it allows precise discrimination in typing. However, there are some problems. Pattern matching, the main tool for eliminating datatypes in functional programming, causes problems with type dependency. In the talk I give an example where pattern matching is intuitive and much more convenient than the usual elimination rule for a datatype: programming the 'head' function for vectors. Then I give a series of examples of programming with heterogenous association lists that get more and more difficult. Finally the 'assoc' function for heterogenous association lists with no duplicate labels is so difficult that a new approach is needed. I show a technique of Conor McBride that works in this example, and many others, but which lacks a general formulation as yet.

Moral: Equality in intentional type theory is a problem. (Lennart Augustsson can be seen nodding his head in agreement.) Question: Is extensional type theory the answer?

15 Pure: A Functional Programming Language Based on Pure Type Systems

Johan Jeuring, Utrecht University, Utrecht (NL)

with Jan-Willem Roorda

We present a functional programming language based on Pure Type Systems (PTSs). We show how we can define such a language by extending the PTS framework with algebraic data types, case expressions and definitions. To be able to experiment with our language we present an implementation of a type checker and an interpreter for our language.

PTSs are well suited as a basis for a functional programming language because they are at the top of a hierarchy of increasingly stronger type systems. The concepts of 'existential types', 'rank-n polymorphism' and 'dependent types' arise naturally in functional programming languages based on the systems in this hierarchy. There is no need for ad-hoc extensions to incorporate these features.

The type system of our language is more powerful than the Hindley-Milner system. We illustrate this fact by giving a number of meaningful programs that cannot be typed in Haskell but are typable in our language. A 'real world' example of such a program is the mapping of a specialisation of a Generic Haskell function to a Haskell function.

Unlike the description of the Henk language by Simon Peyton Jones and Erik Meijer we give a complete formal definition of the type system and the operational semantics of our language. Another difference between Henk and our language is that our language is defined for a large class of Pure Type Systems instead of only for the systems of the lambda-cube.

16 Explicit Subtyping in Dependently Typed Programming

Thorsten Altenkirch, University of Nottingham, Nottingham (GB)

I introduce a notion of subtyping which is a specialisation of the Pi-type, the inhabitants of such types are called coercions. Essentially coercions are eta-long forms of the identity or may refer to other or hypothetical coercions. Subtypings are propositional, i.e. there is at most one coercion, hence coherence holds automatically. I also discussed how subtyping for product types can be derived from subtyping for coproduct types.

This is related to Zhaohui Luo's coercive subtyping. However, one important difference is that we restriuct subtypings such that extensionally there is at most one coercion between any two types.

17 Programmable Pattern Analysis (or 'Greening the Left')

Conor McBride, University of Durham, Durham (GB)

My doctoral work equips the dependently typed functional programmer with a translation from structurally recursive programs presented in a pattern-matching style to programs expressed in terms of the basic elimination operators with which inductive datatypes are traditionally equipped. This translation exploits the fact that dependently typed elimination operators resemble induction principles, explicitly giving the constructor pattern being analysed in each case. By contrast, fold operators in Hindley-Milner languages make no obvious connection between the types of the functions passed in and the patterns to which they correspond.

In this talk, joint work with James McKinna, I propose to take the pattern-specific types of elimination operators as specifications of the components from which allowable systems of patterns may be constructed, with the semantics of the resulting programs being given by whatever implements those operators. As well as supporting the analysis by constructors (which I colour red), available 'for free' with each datatype, we may now specify and implement admissible notions of pattern, putting defined (green) symbols on the left-hand sides of programs. The intuitive first-order style of programming with patterns thus acquires a new compositionality, generalising and giving a direct semantics to Wadler's proposed notion of 'view'.

Of course, in order to use a derived notion of matching, we must write the (usually recursive) higher-order function which implements it. By exploiting the same translation from pattern matching programs to operator applications, these operators themselves may also be presented in a first-order style.

On a technical level, this work shows that dependent types give us the leverage we need to extend the analytical power of a programming language simply by programming in it. More widely, it suggests that we should be prepared to learn afresh how to write programs, and how to choose which programs to write.