Types for Proofs and Programs
Third Annual Meeting

Abstracts of
Talks and Demos

June 12th - 16th

Lökeberg, Sweden

Table of Contents

Abstracts of Talks

A Predicative Analysis of Structural Recursion

Andreas Abel and Thorsten Altenkirch
Ludwig-Maximilians-University Munich
München, Germany
{abel,alti}@informatik.uni-muenchen.de

We are developing a system (MuTTI - Munich Type Theory Implementation) with dependent types which can be used for the development of provably correct programs in Type Theory. Inspired by Coquand's pattern matching for dependent types as implemented in the ALF system and its successors, we define a total language as a subset of a partial one. Hence, we are faced with the problem of verifying termination.

We restrict ourselves to structural recursion, where by structural recursion we mean that the only termination orderings we consider are lexical products of the natural structural ordering on a strictly positive datatype. We also allow mutual recursion. A further restriction is that smaller terms are only generated by primitive operators like case-analysis, projection and application.

Abel implemented a termination checker for a simply typed sublanguage of MuTTI (called foetus), this system allows mutual recursive definitions on general strictly positive datatypes and returns a lexical ordering on the arguments of the function if one exists.

In the work we want to present here, we show that from foetus output we can actually conclude that the function terminates. We define a semantic interpretation of each type and formally define the structural ordering on semantic values of possibly different types. A central result is that the structural ordering is wellfounded at each type. A general soundness theorem allows us to conclude that each term accepted by the foetus system actually terminates.

The original version of the above proof, which can be found in Abel's Master's thesis, defines the fixed-point types by Tarski's fixed-point theorem and thus is impredicative. Now by proving that the semantics of each type is a set-based operator, we the fixed-point types by strictly positive definitions. Thus we get a predicative proof, and our meta-theory is only slightly stronger than our object-theory.

Our next goal is to extend our termination checker to coinductively defined types and dependent types, including the definition of universes. We also hope to be able to answer the question whether the restriction of structural recursive function is actually necessary.

Logic in Dependent Type Theories

Peter Aczel
Departments of Mathematics and Computer Science
Manchester University, Britain
petera@cs.man.ac.uk

In my talk I will explore some of the ways that logic can be treated in a dependent type theory. My main focus will be a framework where the standard apparatus of intuitionistic predicate logic with equality and quantification over types of a specified kind can be built over a base type theory. The propositions-as-types idea aims at a translation of such a logic into the base type theory. This can be done in a variety of ways. Perhaps the most fundamental is to use the Curry-Howard correspondence.

But a popular alternative is the one available, when the base type theory includes a version of the calculus of constructions, with its fully impredicative type Prop. This uses an approach that defines all the logical operations just in terms of implication and universal quantification, including universal quantification over Prop. I believe that the approach originated in some neglected early sections of Bertrand Russell's Principles of Mathematics, that were rediscovered by Prawitz in his Ph. D. thesis. So it would seem natural to call these definitions the Russell-Prawitz definitions of the logical constants.

There is another alternative that gives a family of generalisations of the Curry-Howard correspondence, parametrised by a unary type operator T for which the following condition holds.

> For each type A there is a function $in_A : A \to TA$ and for all types A, B there is a function $lift_{A,B} : (A \to TB) \to (TA \to TB)$ such that $lift_{A,B}(f)(in(a)) = f(a) : B$ for $f : A \to TB$ and $a : A$.

The idea here is to modify the Curry-Howard correspondence by applying T to the result of applying each type operation that represents a logical operation to its arguments. For example the logical operation of disjunction is represented not by the binary operation $+$ on types but now by the operation $+^T$, where $A_1 +^T A_2 = T(A_1 + A_2)$.

I will give several examples of this generalisation of Curry-Howard. Of course the original uses the identity type operator. Another example is the double negation type operator, which of course gives classical logic. The translation that uses the Russell-Prawitz definitions can also be viewed as an example. This time the type operator T is defined as follows.

$$TA = (\Pi p : Prop)[(A \to p) \to p]$$

Reduction-free Normalisation for a λ-calculus with Coproducts

Thorsten Altenkirch
Department of Computer Science, Ludwig-Maximilians-University
Munich, Germany
alti@informatik.uni-muenchen.de

We show how the approach to reduction-free normalisation as presented in [AHS95] can be extended to coproducts. Where in loc.cit we have used presheaves, to deal with coproducts requires sheaves. We define a Grothendieck topology over the category of weakenings similar to the one presented in [FS99]. We can show that the presheaf of normal forms, which is defined as a quotient of an inductively defined set by a structural equivalence relation, is a sheaf. Furthermore the functions quote and unquote can be extended to coproducts. The correctness can be verified using a variation of *Grothendieck logical relations* as introduced in [FS99].

We are not yet able to verify important properties of the structural equivalence such as decidability. We will address this issue in the talk.

This presentation is based on joint work with Peter Dybjer and Martin Hofmann. However, I will present my particular view of the approach here.

[AHS95] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, LNCS 953, pages 182–199, 1995.

[FS99] Marcelo Fiore and Alex Simpson. Lambda definability with sums via grothendieck logical relations. In *Proceedings TLCA 99*, number 1581 in LNCS, pages 147–161. Springe, 1999.

A Logical Framework for mini-ML Programs Proofs and Specifications

Sylvain Baro
UFR d'Informatique, Université Paris VII
Paris, France
Sylvain.Baro@lip6.fr

There are mainly two ways of dealing with proofs and programs. The first one, known as *programming with proofs*, allows to automatically extract functional programs from a proof of the consistency of their specification. The second one, we could call *proving programs*, allows to check that existing programs actually meets their specification.

The source framework of our proposal is Krivine's and Leivant's AF2 in which the program specification can be reduced to its equational description and its intended type (in the usual programming languages meaning) The program itself is extracted from the proof that the intended type is derivable.
In our framework, we replace the equational data with (mini)ML programs. We keep from

AF2 the proof obligation of their intended type but we forget extraction while we already have the program. Doing so, we follow the so called proving programs approach.

Our language for programs and specifications consists of data types and functions definitions in mini-ML syntax and first order predicate calculus extended with function variables and restricted to bound quantification. For data types and programs, we support pattern matching and polymorphism. Quantifiers bindings are ML types.

We will show how this language can be founded on second order predicate calculus extended with a fixed point operator (namely, Parigot's TTR) in which mini ML programs and their natural semantics can be promoted as first class citizens. ML data types can be expressed as second order formulae with fixed point operator if they satisfy some positivity conditions hence ML typing can be embedded in the logical framework. Of course, typing becomes undecidable for the recursive functions ; we will discuss how to do with it.

Electricity, Constructive Logic, and 2-adic Numbers

Gerard Berry
Ecole des Mines de Paris
Sophia-Antipolis, France
`Gerard.Berry@sophia.inria.fr`

Boolean logic is the classical framework to talk about digital electronic circuits. It is indeed good enough for acyclic combinational circuits. We show why cyclic combinational circuits are necessary as well, and we show that constructive logic must replace classical logic for their analysis. Finally, we present Vuillemin's view that the theory of 2-adic numbers is the right way of talking about sequential circuits that operate over time

Extensionality of Simply Typed Logic Programs

Marc Bezem
Department of Philosophy, Utrecht University
Utrecht, The Netherlands
`bezem@phil.uu.nl`

We set up a framework for the study of extensionality in the context of higher-order logic programming. For simply typed logic programs we propose a novel declarative semantics, consisting of a model class with a semi-computable initial model, and a notion of extensionality. We show that the initial model of a simply typed logic program, in case the program is extensional, collapses into a simple, set-theoretic representation. Given the undecidability of extensionality in general, we develop a decidable, syntactic criterion which is sufficient for extensionality. Some typical examples of higher-order logic programs are shown to be extensional.

Programming in Martin-Löf Type Theory: Unification - A non-trivial Example

Ana Bove
Chalmers University of Technology
Göteborg, Sweden
bove@cs.chalmers.se

Martin-Löf type theory is conceived as a formal language in which to carry out constructive mathematics. However, it can also be seen as a programming language where the specifications are types and the programs that satisfy the specification are objects of the type. In this way, when a theorem states the existence of an object with certain properties, the proof of the theorem computes such an object from any given proofs of the assumptions of the theorem.

Unification algorithms are widely used in resolution, term rewriting, type-checking, programming languages, etc. Although unification algorithms can be presented in different ways according to the application, all the presentations are in essence the same. The presentation we consider here is the following: given a list of pairs of terms, unification is the process of finding a substitution that, when applied, makes identical each pair of terms in the list; if such a substitution does not exist, the unification algorithm should produce an output that indicates so.

Unification algorithms are recursive. However, although one can reason about termination they are not recursive over structurally smaller arguments. This is not a problem if we work with imperative or functional languages but this kind of algorithms cannot be directly written in Martin-Löf type theory.

A natural way to overcome this problem is writing the algorithm with the help of the Accessibility predicate. Now, if we compare the Haskell-like and the type theory versions of the algorithm, the latter is longer and less clear, and contains parts that have no computational content because they are just proofs that the recursion is done over smaller elements.

In this work, we define the unification algorithm in type theory in such a way that the algorithm becomes simple, short and very similar to the Haskell-like version of it. Moreover, this presentation separates the algorithm from the proof that the algorithm terminates which contributes to its simplicity and shortness.

On top of the algorithm, we have constructed the proofs that the algorithm is correct. As these proofs follow the same structure as the algorithm itself, they have also benefitted from the method used to write the algorithm.

Using the same technique, we can write in type theory other non-structurally smaller recursive programs. Then, we would have programs that can be proven correct and are similar to their Haskell-like. We believe then, that this work is a step forward closing the existing gap between programming in functional languages and programming in type theory.

Church-Rosser and Strong Normalization of the Module Calculus MC

Judicael Courant
LRI, Orsay, France
`Judicael.Courant@lri.fr`

Several proof-assistants rely on the very formal basis of Pure Type Systems. However, some practical issues raised by the development of large proofs lead to the addition of other features to actual implementations for handling namespace management, for developing reusable proof libraries and for separate verification of distinct parts of large proofs. Unfortunately, few theoretical foundations are given for these features. We recently proposed an extension of Pure Type Systems with a module system, called $\mathcal{MC}$, inspired by SML-like module systems for programming languages, in order to address these issues from a (meta)theoretical side.

Our system gives a theoretical framework addressing the need for the features mentioned above. The expected metatheoretical results hold: Subject reduction, Church-Rosser property, existence of principal types, Strong Normalization and therefore conservativity over PTS and decidability of typing. Our system is therefore a true calculus.

In our talk, we will briefly recall what the module calculus is in general, and we will specifically focus on its reduction relation. We will show why it is highly unusual: it does not enjoy the substitution property (we may have m reducing to m' with $m[x \leftarrow q]$ not reducing to $m'[x \leftarrow q]$ — these module expressions are nevertheless convertible), nor the CR property on untyped module expressions. We will show that Geuvers's technique for proving CR of $\beta\eta$ is helpless since for any pair of module expressions m and m', there exists an untyped module expression m'' reducing to m and m' in a few steps. In fact, proving CR seems as complex as proving strong normalization. We will try to give hints to the audience about this intuition. Then we will introduce the technique we used: simultaneous proof of CR and SN in the style of Coquand-Gallier (using Kripke-style models). This technique is much inspired by Goguen's operational semantics proof for the Calculus of Construction.

Then, if we have enough time, we will show that we are not completely satisfied with the current proof: current definition of module equality prevents us from extending $\mathcal{MC}$ with some useful constructs. We will give hints on what a possible solution could be, still based on Goguen's ideas.

Useless-code Detection and Elimination for Typed Functional Programs

Ferruccio Damiani
Dipartimento di Informatica, Università di Torino
Torino, Italy
`damiani@di.unito.it`

We present a non-standard type assignment system and simplifications mappings for detecting and removing *useless-code* in higher-order typed functional programs with algebraic datatypes and recursive functions. We characterize two classes of useless-code: the *dead-code*, that is code that is never executed under the lazy-call-by-name evaluation, and the *minimum-information-code*, that is code that contributes to the computation only with a minimum amount of constant information.

Useless-code elimination can be performed by optimizing compillers to improve the quality of the code produced. This kind of optimization is particularly useful when dealing with programs obtained by automatic translation from constructive proofs.

An Approach to Real Numbers in Coq

A. Ciaffaglione, P. Di Gianantonio and F. Honsell
Dipartimento di Matematica e Informatica
Università di Udine, Italy
`pietro@dimi.uniud.it`

The approach we present is based on a digit representation for the real numbers, and it capitilzes substantially on the co-inductive features (definition mechanisms and tactics for coindutctive types) offered by Coq.

This approach has the advantage of being connected to exisisting implementations of exact real number computation. In this way it is feasible to use Coq to prove the correctness of fundamental algorithms on real numbers or to synthesise algorithms from proofs.

We discuss the issue of adequacy for our representation of reals and prove some basic properties for real numbers and functions on reals numbers.

If time permits we discuss the connections of our approach with other approaches to real numbers in Coq.

HOL-lambda-sigma: An Intentional First-order Expression of Higher-order Logic

Gilles Dowek, Thérèse Hardin and Claude Kirchner
INRIA, Le Chesnay, France
`Gilles.Dowek@inria.f`

We give a first-order presentation of higher-order logic based on explicit substitutions. This presentation is intentionally equivalent to the usual presentation of higher-order logic based on λ-calculus, i.e. a proposition can be proved without the extensionality axioms in one theory if and only if it can in the other. We show that we can apply the *Extended Narrowing and Resolution* first-order proof-search method to this theory. We get this way a step by step simulation of higher-order resolution. Hence expressing higher-order logic as a first-order theory and applying a first-order proof search method is at least as efficient as a direct implementation. Furthermore, the well studied improvements of proof search for first-order logic could be reused at no cost for higher-order automated deduction. Moreover as we stay in a first-order setting, extensions, such as equational higher-order resolution, are easier to handle.

Proof of Correctness of a Distributed Reference Counting Algorithm

Jean Duprat
Ecole Normale Superieure de Lyon
Lyon, France
`Jean.duprat@ens-lyon.fr`

Correctness is a crucial point of distributed algorithms. Each processing element works locally and cannot handle the global behavior of the computation. In this domain, certified proofs are justified, even if establishing them needs lots of efforts. We show how Luc Moreau and I have proved the correctness of a distributed reference counting algorithm using the proof assistant Coq.

The machine is a finite set of sites connected by a communication network. Each site is a processor and a local memory; it executes imperative programs, it sends and it receives messages. A reference counting program is an operating system service; for example, a garbage collector uses it to know if an object is still referenced somewhere. Moreau's algorithm maintains counter tables in every site by means of messages. Every action of the program is modelized by a transition over the machine status.

The correctness has two facets. The first one is a safety property. If, at any time, there exists somewhere a reference to an object, the counter of the owner cannot be null. The second one is the liveness. If after any time, no new activity begins, the system evolves in a finite sequence of transitions to a null state. The safety proof uses invariant properties (numerical equalities, acyclicity of a graph) through the transitions. The liveness proof looks like a well founded order over the transitions.

Transitions over the machine status are easily translated into inductive definitions in Coq. The asynchronous behavior of the program is modelized by non deterministic choice of legal transitions. Invariance proofs follow the same scheme, an induction followed by a case study. Writing the proof leads us to specifya synchronization constraint upon the network. For some kinds of messages, for any couple of sender-receiver, the order in which the messages are received has to be the same that the order in which they have been sent.

An other interesting point of discussion can be a relaxed notion of liveness, concerning only one object : if at any time, no reference exists upon an object and in the future, no reference to this object will be created, then, after a finite sequence of transitions, the counter of the owner will become null. Such a property needs a fairness constraint in the choice of legal transition.

Ordinals as Universes

Peter Dybjer
Chalmers University of Technology
Göteborg, Sweden
`peterd@cs.chalmers.se`

I will show a way to represent countable von Neumann ordinals in intuitionistic type theory. The key idea is that ordinals are degenerate universes. This idea can also be used for constructing ordinals in Aczel's type-theoretic cumulative hierarchy V.

Formalizing the Fundamental Theorem of Algebra

Herman Geuvers
University of Nijmegen & Eindhoven University of Technology
Eindhoven, The Netherlands
herman@cs.kun.nl

Joint work with Freek Wiedeijk and Randy Pollack

In Nijmegen we are working on formalizing a constructive proof of the Fundamental Theorem of Algebra (FTA) in Coq. The goal of this enterprise is manyfold. First we want to show that a large theory development containing some real mathematics can be done. Second we want to gain experience in setting up and performing such a large project. Third, we want to investigate how type theory or the specific implementation (Coq) can be used in an optimal way (or how they should be improved) to complete our proof development.

In the project we distinguish three phases: 1. Writing the mathematical theory (incl. proofs) on paper, 2. Theory development in Coq: setting up the definitions and stating the theorems to be proved, 3. Proof development in Coq: proving the theorems. At this moment we are in the process of theory development: setting up the definitions and the theorems to be proved. Little proof development has been done.

Formalization of a Concurrent Object Calculus up to Alpha-conversion

Guillaume Gillard
INRIA, Nice, France
Guillaume.Gillard@sophia.inria.fr

We present a formalization of a concurrent object calculus in the Calculus of Inductive Constructions. We use de Bruijn technique in an intermediate syntax, but de Bruijn indices do not appear in the final formalization of the terms of the calculus, which are still defined up to α-conversion. Moreover, we derive substitution rewriting rules and an inductive principle on the final terms. Once a certain amount of preliminary work has been done on the intermediate syntax this induction theorem makes possible natural proofs which do not deal with de Bruijn number.

Using Functor Categories to Explain and Justify Axiomatizations of Variables and Schemata in HOAS

M. Hofmann (LFCS, University of Edinburgh) and F. Honsell and M. Miculan and I. Scagnetto
Dipartimento di Matematica e Informatica, Università di Udine
Udine, Italy
honsell@dimi.uniud.it

We present a type-theoretic axiomatization/account of the notions of *variable* and *scheme* as they appear in Higher Order Abstract Syntax, and we prove its consistency using a functor category model as outlined by Hofmann in his seminal paper [1]. This theory, based on the one in [2] for dealing with Milner's π-calculus, is suggestive of a framework system (metatheory) for reasoning on nominal calculi, and other systems which manipulate variables, names or locations, when represented in HOAS.

This paper has a *pedagogical* purpose. It illustrates on a simple example the novelty of Hofmann's approach to reasoning on systems in HOAS using presheaves to model types, natural transformations to model terms, and Pitt's notion of triposes for interpreting predicates.

This paper has also the *technical* purpose of working out the details of the category-theoretic ideas and constructions outlined in [1, Section 7]. One of the crucial tools that we introduce to this end is a notion of *forcing* which allows us to streamline the computation of the truth value of a proposition.

Finally, this paper has also the *speculative* purpose of discussing, besides consistency, also independence and completeness of the presented axiomatization for variables and schemata in HOAS.

More specifically we provide a model for a Classical Higher Order Logic over datatypes of processes and names together with a number of extralogical axioms including extensionality of schemata

```
Axiom proc_ext : (p,q:name->proc)(x:name)
      (notin x (nu p)) -> (notin x (nu q)) -> (p x)=(q x) -> p=q.
```

These contradict the Axiom of Unique Choice, thus making essential the use of triposes, as opposed to plain topos logic.

Datatypes are intepreted as presheaves over the category of finite variable substitutions, while predicates are interpreted as subpresheaves which commute with substitution in the subcategory of presheaves over *injective* variable substitutions.

1. M. Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. 14^{th} LICS*. IEEE, 1999. `http://www.dcs.ed.ac.uk/~mxh/lics99hoas.ps.gz`

2. F. Honsell, M. Miculan, I. Scagnetto. π-calculus in (Co)Inductive Type Theory. To appear in TCS, 1998.
`ftp://ftp.dimi.uniud.it/pub/miculan/TCS99.ps.gz`

Some Logics for :=

Samin Ishtiaq
Queen Mary & Westfield College, University of London
London, England
`si@dcs.qmw.ac.uk`

Hoare-type program logics have structural properties which often conflict with their uses as meta-languages for state-ful programming languages. For example, the structural rule of Contraction allows us to merge distinct portions of the store, and the structural rule of Exchange allows us to vary the order of evaluation – neither of which are intended by the programmer. We consider some proposals for weaker meta-logics as program logics.

Functional Programming with Dependent Types

Zhaohui Luo and James McKinna
Computer Science Department, University of Durham
Durham, United Kingdom
{Zhaohui.Luo, J.H.McKinna}@durham.ac.uk

In this talk, we shall discuss some issues in strong functional programming with dependent types (SFPD). Two difficult issues frequently recur in studies in this area: the removal of "non-informative" (propositional) types, and the need for explicit transfer functions between provably equal, but definitionally distinct, types. In addressing the first, we argue that an SFPD language should *not* incorporate logical propositions or proof objects. One way we propose to achieve this is to exclude all empty types; in other words, an SFPD language should only consist of non-empty data types and families of non-empty types. We shall also discuss some practical issues in SFPD. In particular, we propose the use of coercive subtyping as developed in Durham, both in addressing the second issue, and more generally towards making the use of dependent types easier.

About the Arithmetic Universe via Type Theory

Maria Emilia Maietti
School of Computer Science, University of Birmingham
Edgbaston, Birmingham, United Kingdom
mem@cs.bham.ac.uk

The notion of arithmetic universe was introduced by A. Joyal to give a categorical proof of Gödel's incompleteness theorem. He built an instance of it by an exact completion process from a Skolem Theory (that is a completion to add quotients to a category similar to passing from a type theory to its setoid version).

Here, we define an arithmetic universe as a pretopos with parameterized recursive objects and we prove that within it we can build free categories and free internal diagrams as in Joyal's construction. We prove this by using the internal type theory of an arithmetic universe.

Indeed we have a typed calculus based on dependent types and proof-terms, which is valid and complete with respect to the class of these arithmetic universes. This typed calculus consists of terminal type, indexed sum types, extensional equality types, the false type, disjoint union types with disjointness axiom, effective quotient types on proof-irrelevance equivalence relations, - all these corresponding to a pretopos - and finally the list types corresponding to parameterized recursive objects.

Always by means of the internal type theory we prove a conjecture, stated by Vickers and others working in the field, that the category of models of a finite sketch with finite limits into an arithmetic universe is a reflective subcategory of the corresponding category of functors, but only in the case that the cones are decidable.

For the finite case, this conjecture is a constructive and predicative formulation of a classic theorem proved by Kennison, who considered the category of models and functors over a small sketch with values in the category *Set* of *ZFC* sets. In his proof he used Freyd's adjoint functor theorem, quantification over all models and cardinality arguments with the replacement axiom. In the finite and decidable case, however, we prove this by using predicative coherent logic, which does not have implication, lists and quotients.

The 3 Gap Theorem: Specification and Proof in Coq

Micaela Mayero
INRIA-Rocquencourt
Le Chesnay, France
`Micaela.Mayero@inria.fr`

The objective of this work is to prove a mathematics theorem using the Coq proof assistant.

The theorem choosen is the three gap theorem (Steinhaus conjecture) which proves that given N points placed consecutively around the circle by an angle between 0 and 1, then the points partition the circle into arcs or gaps of at most three different lengths.

The proof used is Tony Van Ravenstein's proof. This kind of proof based on the geometrical intuitions is a challenge for the proof assistant systems. Moreover, this proof uses the real numbers and some of their properties, the fractional parts for example, which were not formalized in Coq.

We start by giving a survey of the proof in order to put in evidence the tools nedeed, then we speak about the real numbers formalization and the choice made in Coq.

OLEG, a Type Theory with Holes, Lets and Guesses

or "You Can't Put a Hole where a Hole Don't Belong"

Conor McBride
University of Edinburgh
Edinburgh, Scotland
`ctm@dcs.ed.ac.uk`

OLEG is Luo's Extended Calculus of Constructions, augmented to account for proofs in progress as well as complete constructions: a valid context may be viewed as the state of a theorem prover, with tactics as admissible rules.

To this end, OLEG introduces two binding operators new to ECC:

- ! ('let') provides completed programs and theorems to be named locally
- ? ('hole') names a new unknown (or 'metavariable'); candidate values ('guesses') may be attached to holes, but with no computational effect

The binding of holes in the context and in partial proof terms is a natural way to control both their scope and the variables which may occur in any candidate value. The location of ?-bindings is restricted, so that they may be solved independently with values of their indicated types.

This treatment involves neither explicit substitution nor unwarranted skolemisation of holes. The standard tactics of refinement proof may be reconstructed without difficulty.

OLEG has been partially implemented, by rearranging LEGO.

IPTS: A Formalism Extending PTS with Implicit Arguments

Alexandre Miquel
Coq Project, INRIA Rocquencourt
Le Chesnay, France
`Alexandre.Miquel@inria.fr`

Implicit arguments are subterms in PTS that can be automatically inferred and that the user would like to omit.

From examples in Coq, we show that a too loose criterion of implicitness may be source of confusion and strange phenomenons, in the sense that sometimes, terms that appear to be equal for the user are in fact not convertible in their internal representation, an even not provably equal.

To answer the problem, we propose a calculus extending the PTS's by weakening the convertibility rule in order to ignore arguments marked as 'implicit' (but still present in the syntax). To avoid inconsistencies, such a weakening forces some syntactic restrictions: we will be interested in a restriction based on an extractibility criterion very similar to the program extraction in Coq.

Better, we can turn the extended PTS into a lighter form – called IPTS – where implicit information has totally disappeared. This lighter form of PTS can be seen as a λ-calculus à la Curry with two product operators – an explicit and an implicit one. Although type-checking is undecidable in this calculus, the typing relation enjoys good meta-theoretic properties, like $\beta\eta$-subject reduction.

Finally, we focus on the implicit variant of ECC – the Calculus of Constructions with universes. We build a model and a pre-model based on coherence spaces, and show that IECC is consistent and enjoys the strong normalization property.

Prototypes with Dependent Types

Petri Mäenpää
Nokia Telecommunications
TKK, Finland
`pmaenpaa@cs.hut.fi`

This talk shows how to give dependent types to a limited form of prototype objects, in the sense of object-oriented programming. This provides a new way of defining data types and functions on them in type theory.

Predicative Toposes

Erik Palmgren
Department of Mathematics, Uppsala University
Uppsala, Sweden
`palmgren@math.uu.se`

We discuss a notion of predicative topos, suitable for formalisation in Martin-Löf type theory, which is sufficiently strong to develop models of constructive set theory (CZF).

We show that predicative toposes are closed under the construction of internal sheaves, just as elementary toposes. As an application we give a constructive model of CZF plus countable choice and an omniscience principle. (The talk describes joint work with Ieke Moerdijk, Utrecht.)

Abstractions in Type Theory

Henrik Persson
Chalmers University of Technology
Göteborg, Sweden
henrikp@cs.chalmers.se

We discuss the importance of abstractions in type theory, and in particular our development of some formalisations in the type-checker Agda. In these formalisations, we use abstractions heavily.

A simple use of abstraction is when formal proofs are developed assuming the existence of a datatype with certain properties. This datatype can later be constructed and its properties proved. In this way, the main development becomes independent of the particular representation of the datatype, which often makes the formal proof simpler to convey, both to machines and humans. For example, we have formalised Buchberger's algorithm for computing Grbner bases in this style; assuming a datatype of polynomials with certain properties, we proved the existence of Grbner bases.

A more refined version of this approach is to assume a datatype with a universal property. From this property, most other properties can then be proved, independently of the representation of the datatype. For example, we have formalised the polynomial ring in this way, assuming only a way to construct monoid copowers. Similarly, we have formalised the multiset order, assuming a free Abelian monoid as representation of multisets.

Algebra with Coq

Loïc Pottier
projet LEMME, INRIA
Sophia Antipolis Cedex, France
pottier@sophia.inria.fr

We will present a development of basic commutative algebra using the system Coq and its interface Ctcoq. It includes, following the book "Algebra" of Serge Lang, basic notions about categories, groups, rings, modules and fields (available at
http://pauillac.inria.fr/coq/contribs/algebra.html).

We have used several original features of Coq and Ctcoq to represent comfortably mathematical structures. First is the record data structure, implemented in Coq with inductive types with one constructor. An algebraic structure is represented by a record containing its operations and their logical properties. Second are the coercions between types, that allows to embed structures (a group contains its underlying monoid, which contains a semi-group, etc), and to consider a single complex structure with several points of view (a ring is a module on itself, an algebra, a group, etc). With some work, we can obtain multiple inheritance with this mechanism. Third are structural edition and extensible pretty printing allowed by Ctcoq, very useful to show mathematical notations, to hide unnecessary parts of big expressions without forgetting them during manipulations.

But we had to go round the absence of quotient types and subtypes in Coq: quotients and subsets are too basic in mathematics to avoid them. To work with quotients, we have used the notion of "setoid" (type+congruence): to build a quotient set is then trivial, because we just have to replace the equality. To work with subtypes (more exactly subsets) is less easy. Given a set(oid) E, a type F, and a function $f : F- > E$, we define a subset of E by identifying the image of f with the setoid given by F and the relation $x \equiv y \Leftrightarrow f(x) \equiv f(y)$. One of the main problem in practice is that coercions of Coq do not work with subsets.

For every kind of structure (group, ring, etc) we made an intensive use of free structures. In particular, it allows to define clearly the notion of polynomial ring, and the general notion of generated sub-structure. Moreover, proofs about these notions are greatly helped by this categorical approach. As an example consider the problem of defining the sub-module generated by a subset V of a R-module M. We can define it a the set of linear combinations

$$\{\sum_{1\leq i\leq n} a_i x_i | a_i \in R, x_i \in V\}$$

But this implies to make tedious proofs about dependent types (indexes are integers which must sastify bounding constraints). More simple is to define this sub-module as the image of the free-module ganerated by V, by the module homomorphism canonicaly defined by the inclusion of V in M and the universal property of free-modules. More, if we want algorithmic efficiency, we can restrict us to canonical forms in the free-module.

This development is currently used in a development of an interactive environment for teaching mathematics in first years of university, and in a formalization and a certification of an algorithmic method of algebraic geometry for polynomial interpolation ("mthode d'Horace").

Pure Type Systems and Static Analyses

Frédéric Prost
LIP, Ecole Normale Supérieure de Lyon
Lyon, France
`Frederic.Prost@ens-lyon.fr`

Non-standard type inference has been shown fruitful for functional program analyses. Many analyses, e.g. strictness, dead code, control flow, binding time have been formalized using many non-standard type systems. Common features can be found in all these systems. A typed programming language is given. The type system is then modified by the addition of annotations on types or term constructors. Those annotations denote semantical properties of programs.

One aim of this talk is to give, from a theoretical perspective, hints to the following question: *what is the meaning of an annotation?*. To answer this question we use a formalism slightly different from Pure Type Systems one.

During the early 90's Barendregt along with Berardi proposed a nice formalism to describe in a very precise way some subtle differences between type systems. Pure Type Systems (PTS) are parameterized by a set of constants, a set of axioms and a set of rules.

Those three sets are enough define most of known type systems. Originally, Barendregt used this formalism to build a classification of 8 type systems ordered by inclusion (the biggest one being the calculus of constructions).

We allow the building of types over *variable sorts.* Thus, there are judgments of the form $x : X : \alpha$, meaning that variable x is of variable type X which is of *variable* sort α. Similarly to Coq *Prop* and *Spec* we use *different* sort of the same level to denote *different* properties.

We show that our slightly different version of PTS still has good properties. We will show that adding a fourth kind of abstraction (namely the abstraction of sort relatively to terms), to the three standard ones (the ones of the λ-cube), preserves Strong Normalization. Moreover, we will explain how this abstraction may be used to formalize various static analyzes.

A Meta-Logical Framework for LF

Carsten Schürmann
Carnegie Mellon University
Pittsburgh, USA
`carsten@cs.cmu.edu`

In the judgments-as-types representation technique of the logical framework LF, functions are used to encode higher-order abstract syntax as well as parametric and hypothetical judgments. Adequacy of such encodings requires weak function spaces which means that meta-theoretic proofs by structural induction or case analysis are not expressible in the same framework.

In previous work, we have developed a meta-logic $\mathcal{M}_2$ incorporating recursive functions which are strictly separated from LF functions. We demonstrated that $\mathcal{M}_2$ is sound and suitable for implementation. Our theorem prover Twelf has automatically proven a number of meta-theorems from the area of programming languages and logics. On the other hand, $\mathcal{M}_2$ has a serious limitation: it can only be used to reason about *closed* objects.

In my talk I present work in progress on an extension of $\mathcal{M}_2$ to reason about objects which are valid in *regular* contexts. Regular contexts are described inductively and must satisfy exchange, weakening and contraction. I will also report on a prototype implementation which we have used to obtain automatic proofs of the diamond property for parallel reduction and cut-elimination for intuitionistic logic (among others).

Kruskal's Tree Theorem in a Constructive Theory of Inductive Definitions

Monika Seisenberger
Ludwig-Maximilians-University
Munich, Germany
`seisenb@rz.mathematik.uni-muenchen.de`

We give a constructive proof of Kruskal's tree theorem using inductive definitions and discuss the connection to the ordinal theoretic proof in [Sch79] respectively [RW93].

[**RW93**] Michael Rathjen and Andreas Weiermann. Proof–theoretic investigations on Kruskal's theorem. *Annals of Pure and Applied Logic*, 60:49–88, 1993.

[**Sch79**] Diana Schmidt. Well–orderings and their maximal order types, 1979. Habilitationsschrift, Mathematisches Institut der Universität Heidelberg.

Why Meaning Explanations Do not Justify U : U

Anton Setzer
Uppsala University
Uppsala, Sweden
`setzer@math.uu.se`

Hilbert's program was to develop proofs of the consistency of theories for carrying out mathematics by using finitary methods. By Gödel's results we know that finitary methods do not suffice. The most general version of a revised Hilbert program would be to replace finitary methods by principles, for which we have philosophical arguments of their validity. Of course philosophical arguments are never as secure as mathematical proofs, but we know we cannot do better.

The currently both mathematically and philosophically most developed version of such a revised Hilbert program is the proof of the consistency of mathematical theories in extensions of Martin-Löf's constructive type theory. The mathematical part of this is well developed and currently we are able to carry it out for Kripke-Platek set theory extended by a Π_3-reflecting admissible, which suffices for most what is needed in mathematics.

The philosophical part of the argument is to show via Martin-Löf's meaning explanations that from a proof in type theory the validity of what is proved follows. The problem, one is always confronted with when arguing for Martin-Löf's type theory as a foundation for mathematics, is the fact that the first version of it was inconsistent: it had the principle type : type, which in modern formulations of type theory corresponds to a universe having a code for itself as an element. There has been recently a discussion on an email list about, what is wrong with meaning explanations for such a universe, but without a conclusive answer.

In this talk we will show why we can**not** give a meaning to a universe having a code for itself. If time permits we will show how to modify meaning explanations in order to justify the Mahlo universe.

Metamathematical Analysis of a Theory of Specifications

Nora Szasz
Universidad de la República
Montevideo, Uruguay
`nora@fing.edu.uy`

We present a formulation of a Theory of Specifications based on Martin-Löf's Type Theory. The theory contains types types for representing specifications, whose corresponding notion of implementation is that of a pair formed by a program and a correctness proof.

We aim at a theory with rules for simultaneously building programs together with their correctness proofs, but such that program extraction is straightforward.

We start by formulating a Calculus for Program Verification that, roughly speaking, consists of two copies of Martin-Löf's Type Theory (one for data types and programs and one for propositions and proofs) in such a way that no computational object can depend on a logical one. We then formulate the Theory of Specifications together with a confluent and normalizing computational relation on terms and types. This relation maps judgements in the Theory of Specifications into judgements in the Verification Calculus. As a consequence, every implementation of S of the Theory of Specifications computes to a pair composed of a data type A and a predicate P over A of the Verification Calculus, as well as every implementation of S computes to a program p of data type A and a proof that $P(p)$ holds. This implies that the program extraction process is immediate from an inhabitant of a specification after computing its normal form.

An Application of Cut Elimination to a System with Depedent Types

Jan von Plato
University of Helsinki
Helsinki, Finland
vonplato@helsinki.fi

Dependent types arise naturally in elementary geometry, through conditions for the execution of geometric constructions. A way is presented for the extension of the structural analysis of first-order proofs, via cut-free derivability in sequent calculus, to a system of affine plane geometry. Its axioms are turned into nonlogical rules of a suitable form and the dependent type structure is expressed through well-formedness of instances of these rules.

Isar — a Generic Interpretative Approach to Readable Formal Proof Documents

Markus Wenzel
TU Munich, Germany
wenzelm@in.tum.de

We present a generic approach to readable formal proof documents, called *Intelligible semi-automated reasoning* (*Isar*). It addresses the major problem of existing interactive theorem proving systems that there is no appropriate notion of proof available that is suitable for human communication, or even just maintenance. Isar's main aspect is its formal language for natural deduction proofs, which sets out to bridge the semantic gap between internal notions of proof given by state-of-the-art interactive theorem proving systems and an appropriate level of abstraction for user-level work. The Isar language is both human readable and machine-checkable, by virtue of the Isar/VM interpreter.

Compared to existing declarative theorem proving systems, Isar avoids several shortcomings: it is based on a few basic principles only, it is quite independent of the underlying logic, and supports a broad range of automated proof methods. Interactive proof development is supported as well. Most of the Isar concepts have already been implemented within Isabelle.

Efficient Reflexion

Benjamin Werner
INRIA Rocquencourt
Le Chesnay, France
`Benjamin.Werner@inria.fr`

It is a new trend to use the ability of type theory for reflexion, in order to enhance the efficiency of proofs by equational reasoning with respect to size, proof-checking time and automation. This gives raise to interesting engineering problems.

I will discuss some aproaches which have different advantages and weak points. In particular, I will show how the introduction of explicit traces for the rewrite sequence may, in some cases, enhance efficiency and introduce more flexibility in the used strategies.

Abstracts of Demos

Plastic: An Implementation of Typed LF with Coercions

Paul Callaghan
Computer Assisted Reasoning Group, University of Durham
Durham, England
p.c.callaghan@durham.ac.uk

The Typed Logical Framework (LF) (Luo 1994, "Computation and Reasoning", chapter 9) is a general type theory in which other type theories may be specified, as suits a particular application. LF been extended by inductive families and universes, and more recently, by Coercive Subtyping. 'Plastic' is an implementation of LF in the form of a proof assistant.

There are several motivations for this new implementation. The most significant at the moment is to support Durham's research in Coercive Subtyping, which treats subtyping as an extension of LF. Other motivations include research with implementation techniques for type theory and interfaces (especially multiple contexts and meta-variables), and use as the basis for practical, domain-specific reasoning tools, e.g. a mathematical vernacular system. Such tools will work *by* type theory but should not require detailed knowledge *of* type theory to be used successfully.

Plastic contains standard type synthesis and conversion algorithms, plus the usual apparatus of parser, context mechanism, pretty-printer, emacs interface (via Proof General). It is written in Haskell. The inductive families have been implemented as fundamental, allowing several important optimisations; hence their elimination (i.e., reduction) is particularly fast. A form of meta-variables has been implemented on top of the base system, using unification and a constraint-passing type synthesis algorithm. This allows a basic form of argument synthesis and an adequate proof facility, via Refine and Intros. Note that there is no "goal state" as implemented in other proof assistants. Libraries and non-trivial test cases are currently being written.

Plastic is being used in Durham's research on Coercive Subtyping, and incorporates recent theoretical work on Dependent Coercions and Coercion Rules by Luo and Soloviev. The implications of the more general coercion forms are being studied by case studies, e.g. examples from functional programming. The implementation is experimental: new algorithms are required to handle such coercion forms efficiently. The requirements of coherence checking are another important issue. Note that the assumptions made in other implementations may be too restrictive for our work, so we are investigating how they can be relaxed.

In addition to standard examples, we will demonstrate:

- dependent coercions.
- coercion rules: lifting coercions over inductive families.
- overloading of names via coercive subtyping.

- ‘inheritance’ of subtyping in a user’s customised theory via the underlying subkinding of the framework.

More information may be found via the Computer Assisted Reasoning Group web pages at `http://www.dur.ac.uk/CARG`.

Proof of Imperative Programs in the System Coq

Jean-Christophe Filliâtre
LRI, Orsay, France
`filliatr@lri.fr`

We present here a tactic to establish the total correctness of imperative programs in the system Coq. On a theoretical point of view, the method is based on a translation of imperative programs into purely functional ones, itself based on a static analysis of their effects. (That work has been presented on the last TYPES meeting.) On a practical point of view, programs are given in a ML syntax, which includes references, arrays, loops and recursive functions. Programs are annotated in a Floyd-Hoare logic style, with pre- and post-conditions. Loops may be annotated with invariants, and their termination is justified by giving a variant, as for recursive functions. When the tactic is applied to a program, a bundle of proof obligations is generated, which appear as usual Coq goals.

We illustrate the use of that tactic on simple programs (factorial, efficient exponentiation, in imperative and declarative style, etc.) Some bigger proofs of programs have been realized (quicksort, heapsort, Knuth-Pratt-Morris, Floyd’s algorithm, etc.) and are already available on the web page of the system Coq (http://coq.inria.fr).

The Proof Assistant Alfa

Thomas Hallgren
Chalmers University of Technology
Göteborg, Sweden
`hallgren@cs.chalmers.se`

The proof editor Alfa is the most recent contribution to the Alf family of proof editors, which allows computer assisted development of proofs in Per Martin-Lf type theory through direct manipulation of proof objects, using incremental proof checking to provide immediate error feedback.

Alfa improves on earlier editors in the Alf family on at least the following points:

- More flexible user interaction: most editing operations can be performed either with the mouse or the keyboard, reducing the amount of switching between keyboard and mouse and thus speeding up the editing process.

- More user control of proof presentation: this includes simple pretty printing features, like infix operators and quantifier notation, and also the possibility to present proofs in natural deduction style. These features has allowed Alfa to be used by undergraduate students without any knowledge of type theory in courses on logic at both Chalmers and the University of Uppsala.

- Termination checker: like in earlier Alf versions general recursion is allowed, but there now is a termination checker that can be invoked to check that only structural recursion/induction has been used.

Further, the underlying language used in the Alf family of proof editors have moved closer to ordinary functional programming languages. In fact, the language used in Alfa is almost identical to the recently developed language Cayenne, so Alfa can be seen as a syntax directed editor for this programming language.

The demonstration is intended to give an idea of what working with Alfa is like and to show the new features mentioned above.

References/Credits

- Alfa demo abstract online,
  `http://www.cs.chalmers.se/~hallgren/Alfa/demo990612.html`
- Alfa, home page. `http://www.cs.chalmers.se/~hallgren/Alfa/`
- Agda, the proof engine used in Alfa, developed by Catarina Coquand.
  `http://www.cs.chalmers.se/~catarina/agda/`
- Cayenne, a functional language with dependent types by Lennart Augustsson.
  `http://www.cs.chalmers.se/~augustss/cayenne/`
- The Programming Logic Group in Gteborg, where the Alf editors were developed.
  `http://www.cs.chalmers.se/Cs/Research/Logic/`

Demonstration of GF (Grammatical Framework)

Aarne Ranta
Chalmers University of Technology
Göteborg, Sweden
`Aarne.Ranta@cs.chalmers.se`

GF is a Logical Framework (LF) extended with information on concrete syntax. GF grammars are like LF theories in which every function declaration is annotated with a rule telling how the function is to be printed. From the linguistic point of view, it is a grammar formalism based on type theory.

As an example of a GF rule, consider

```
Ev : (x:Nat)Prop - x "is even" ;
```

The hyphen - divides the rule into a type-theoretical function declaration and a syntactic rule. In general, the annotations need to be parametrized to describe morphological dependencies in different languages. For instance, the French tule corresponding to the English one above has to control the mode of the sentence and the of the adjective - this can be expressed

```
Ev : (x:Nat)Prop - case (ind)  -> x "est"  pair + adjReg(Gen(x)),
                        (subj) -> x "soit" pair + adjReg(Gen(x)) ;
```

Experimental GF grammars have been written for fragments of seven natural and a few formal languages.

The type theory of GF is Martin-Löf's higher-level type theory. The novelty lies in the syntactic annotation language. The syntactic annotations of GF can be used both for generating and for parsing concrete syntax.

The GF implementation consists of a type checker, parser, and generator. These components work generically for any GF grammar. There are two user interfaces: a graphical one usable for syntax editing much in the same way as ALF, and a line-based one that also has commands for checking and manipulating grammars. These implementations are written in Haskell, using the GUI library Fudgets for the graphical interface.

Parallel grammars can be given to several languages simultaneously, which results in multilingual editing environments. The intended application of GF is multilingual technical documentation on limited domains, especially in domains that need a semantic verification of content, which is provided by the dependent types of constructive type theory. GF was mainly implemented at Xerox Research Centre Europe in Grenoble, except the type checker by Petri Mäenpää, at Nokia Telecommunications and Helsinki University of Technology. At Xerox and Nokia, some experiments with GF are being made for prototyping.

Twelf — a Meta-Logical Framework for Deductive Systems

Carsten Schürmann
Carnegie Mellon University
Pittsburgh, USA
`carsten@cs.cmu.edu`

Twelf is a meta-logical framework for the specification, implementation, and meta-theory of deductive systems from the theory of programming languages and logics. It relies on the LF type theory and the judgments-as-types methodology for specification, a constraint logic programming interpreter for implementation, and a meta-logic for reasoning about object languages encoded in LF. It is a significant extension and complete reimplementation of the Elf system.

Twelf is written in Standard ML and runs under SML of New Jersey and MLWorks on Unix and Window platforms. The current version (1.2) is distributed with a complete manual, example suites, a tutorial in the form of on-line lecture notes, and an Emacs interface. Source and binary distributions are accessible via the Twelf home page `http://www.cs.cmu.edu/~twelf`.

Isabelle/Isar — Demonstrated together with ProofGeneral/isar

Markus Wenzel
TU Munich, Germany
`wenzelm@in.tum.de`

The forthcoming Isabelle99 release offers the *Isar* subsystem (for *Intelligible semi-automated reasoning*) as a high-level proof language interface layer, beyond traditional tactic scripts.

The Isabelle/Isar system provides an interpreter for the Isar formal proof document language, which is discussed in more detail in a separate talk.

Isabelle/Isar input consists either of *proper document constructors*, or *improper auxiliary commands* (for diagnostics, exploration etc.). Proof texts consisting of proper document constructors only admit a purely static reading, thus being intelligible later without requiring dynamic replay that is so typical for traditional proof scripts. Any of the Isabelle/Isar commands may be executed in single-steps, so basically the interpreter has a proof text debugger already built-in.

Employing the *ProofGeneral/isar* instantiation of the generic Emacs interface for interactive proof assistants of LFCS Edinburgh, we arrive at a reasonable environment for *live document editing*. Thus proof texts may be developed incrementally by issuing proper document constructors, including forward and backward tracing of partial documents; intermediate states may be inspected by diagnostic commands.

The Isar subsystem is tightly integrated into the Isabelle/Pure meta-logic implementation. Theories, theorems, proof procedures etc. may be used interchangeably between Isabelle-classic proof scripts and Isabelle/Isar documents. Isar is as generic as Isabelle, able to support a wide range of object-logics. The setup demonstrated here is for Isabelle/HOL.