ISSN 2186-7437

NII Shonan Meeting Report

No. 2011-3

Dependently Typed Programming

Shin-Cheng Mu
Conor McBride
Stephanie Weirich

September 14–17, 2011

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

Dependently Typed Programming

Organizers:
Shin-Cheng Mu (Academia Sinica, Taiwan)
Conor McBride (University of Strathclyde, UK)
Stephanie Weirich (University of Pennsylvania, USA)

September 14–17, 2011

A thriving trend in computing science is to seek formal, reliable means of ensuring that programs possess crucial correctness properties. One path towards this goal is the design of high-level programming languages which enforces that program, by their very own structure, behave correctly. In the past decade, the use of type systems to guarantee that data and program components be used in appropriate ways has enjoyed wide success and is still a main focus both from researchers and developers.

With more expressive type systems, the programmer is allowed to specify more fine-grained properties the program is supposed to hold. Dependent type systems, allowing types to refer to (hence depend on) data, are particularly powerful since they reflect the reality that what defines appropriate program behaviours is often dynamic. With the type system, programmers are free to communicate the design of software to computers and negotiate their place in the spectrum of precision from basic memory safety to total correctness.

Dependent types have their origin in the constructive foundations of mathematics, and have played an import role in theorem provers. Advanced proof assistants, such as Coq, has been used to formalise mathematics (e.g. the proof of the four colour theorem) and verify compilers. Only recently have they begun to be integrated into programming languages. The results so far have turned out to be fruitful. A number of dependently typed programming language and systems have been proposed, including Cayenne, DML, Epigram, mega, ATS, Agda, Guru, etc, showing a maturity beyond the proof-of-concept stage. Papers on dependent types submitted to and published in major conferences have significantly increased in number. Some more mainstream functional programming languages, such as Haskell, also start to adopt features such as GADT and type family that are strongly influenced by dependent types.

Dependently typed programming, however, is yet to be considered a practical tool and an efficient device to ensure program correctness and to reduce development costs. Many issues remain to be resolved, including but not limited to:

- design of a small but expressive core language on which to built up a verifiable metatheory;
- metaprogramming, reflectivity, and the possibility of representing dependently typed terms in a dependently typed language;

- representing variable-binding and supporting both structural inspection and functional usage;
- interpreting representations to automate problem solving;
- separating and redesigning a language of proof as distinct from a language for "programming";
- integrating extensional reasoning about functions with computation in dependent type systems, while at the same time concealing the internal structure of proofs;
- modelling interaction and communication in distributed, concurrent systems using dependent types.

This workshop aims to provide a venue for people actively working in this field so that these issues could be discussed.

Overview of Talks

A Dependently Typed Core Language With Termination Guarantees

Abel, Andreas (Ludwig-Maximilians-Universität München)

Implementations of dependently typed languages and proof assistants like Coq and Agda are complex because they involve, besides code for evaluation and type-checking, modules such as unification, dependently typed pattern matching and termination checking. To make life for the user of a dependently typed language bearable, these modules often feature heuristics that are, especially in their interaction, theoretically less explored and, in practice, not bug-free.

To reduce the trusted code base of a proof assistant, small core languages with comparably simple type checking have been proposed into which the surface language is to be translated. An example is PiSigma, a core language designed by Altenkirch et al., which features only dependently typed functions (Pi), dependently typed pairs, a type for propositional equality, a universe of types, recursion and tags in enumeration types together with case distinction over tags. A drawback of PiSigma is that termination of recursion and logical consistency is not guaranteed by the type system, due to the presence of the self-contained type universe and the absence of a termination checker.

We propose TypeCore, a similar language, yet with a predicative polymorphic universe hierarchy and termination guarantees built into the type system. The main addition is a well-founded set of pseudo-ordinals one which one can recurse to define types and programs, thus, ensuring termination. The language avoids awkward encodings by offering primitives for mutual and measure-based recursion. Bounded existential and universal quantification over pseudo-ordinals allows one to define the equivalent of inductive and coinductive types. Naturally, this gives a semantics of advanced recursion patterns like induction-coinduction and induction-recursion.

The Case of the Smart Case

Altenkirch, Thorsten (University of Nottingham), based on joint work with Andreas Abel, Thomas Anberre, Nils-Anders Danielsson and Shin-Cheng Mu

When constructing dependently typed programs we would often like to use a case construct such that the type checker is aware of the choices we have made. This problem can reduced to deciding beta-equality with assumptions of a certain form. I suggest that this problem is decidable in simple cases but we have so far been unable to prove this. Maybe somebody else has looked at this already.

Wander Types; Realizing Data Structure Through Coinduction/Recursion

Capretta, Venanzio (Radboud University Nijmegen)

Normalisation for "Outrageous but Meaningful Coincidences"

Danielsson, Nils Anders (Chalmers University of Technology and University of Gothenburg)

Last year McBride presented a new technique for defining dependently typed languages in a well-typed way (without using raw terms). However, he left the definition of normalisation functions to future work. I will show how one can implement a normalisation function for a simple, dependently typed logical framework defined using McBrides technique.

Refinement Types for Modular Cryptographic Verification

Fournet, Cédric (Microsoft Research). Joint work with Markulf Kohlweiss and Pierre-Yves Strub.

Type systems are effective tools for verifying the security of cryptographic programs. They provide automation, modularity and scalability, and have been applied to large security protocols. However, they traditionally rely on abstract assumptions on the underlying cryptographic primitives, expressed in symbolic models. Cryptographers usually reason on security assumptions using lower level, computational models that precisely account for the complexity and success probability of attacks. These models are more realistic, but they are harder to formalize and automate.

We present the first modular automated program verification method based on standard cryptographic assumptions. We show how to verify ideal functionalities and protocols written in ML by typing them against new cryptographic interfaces using F7, a refinement type checker coupled with an SMT-solver. We develop a probabilistic core calculus for F7 and formalize its type safety in Coq.

We build typed module and interfaces for MACs, signatures, and encryptions, and establish their authenticity and secrecy properties. We relate their ideal functionalities and concrete implementations, using game-based program transformations behind typed interfaces. We illustrate our method on a series of protocol implementations.

Modularising inductive families

Gibbons, Jeremy (University of Oxford). Joint work with HsiangShang Ko

Dependently typed programmers are encouraged to use inductive families to integrate constraints with data construction. Different constraints are used in different contexts, leading to different versions of datatypes for the same data structure. Modular implementation of common operations for these structurally similar datatypes has been a longstanding problem. We propose a datatype-generic solution based on McBrides datatype ornaments, exploiting an isomorphism whose interpretation borrows ideas from realisability. Relevant properties of the operations are separately proven for each constraint, and after the programmer selects several constraints to impose on a basic datatype and synthesises an inductive family incorporating those constraints, the operations can be routinely upgraded to work with the synthesised inductive family.

Dependent Polynomial Functors for Inductive Families

Hamana, Makoto (Gunma University)

It is well-known that every algebraic datatype is characterised as the initial algebra of a polynomial functor on sets. We extend the characterisation to inductive families. Specifically, we show that inductive families are characterised as initial algebras of Gambino and Hylands dependent polynomial functors. We also show that the differentiation of dependent polynomial functors provides zipper data structures on inductive families. This is joint work with Marcelo Fiore.

Parametricity for Dependent Types

Jansson, Patrik (Chalmers University of Technology and University of Gothenburg)

Call-by-Value Reasoning and General Recursion

Kimmell, Garrin (University of Kansas)

Moving dependent types into the realm of everyday programming practice has been limited, in part, by the restriction of current languages to terminating programs to preserve soundness. The Trellys project is an initiative to relax this restriction by integrating a general-purpose dependently typed language, including general recursion, with a sound type theory. In this talk, I describe one approach that we are currently examining to accomplish this goal, based on a syntactic separation between a sound proof fragment and a general programming fragment. While the syntax of proof and program are distinct, the language allows proofs over programmatic terms (a property we refer to as "freedom of speech") and for programmatic terms to include proofs.

I will describe a number of features of the language that facilitate call-by-value equational reasoning in the presence of general recursion. These include partial evaluation modulo propositional equality, sound call-by-name reduction based on terminating arguments, and multiconversion to preserve dependency during multiple steps of equational reasoning.

How to Reify Fresh Type Variables?

Kiselyov, Oleg (Fleet Numerical Meteorology and Oceanography Center) and Shan, Chung-chieh (Rutgers University)

To type-check a use of rank-2 polymorphism, a type checker generates (gensyms) a fresh type variable and watches for its scope, following the proof rule for universal introduction. Due to this implementation, rank-2 polymorphism has been used (some would say abused) to express static capabilities and assure a wide variety of safety properties statically, including

- isolation of mutable state threads,
- bounds of array indices,

- orderly access and timely disposal of resources such as file handles,
- evaluating only closed code while manipulating open code,
- lexical scoping of variables in generated code, and
- distinction of perturbation variables in automatic differentiation.

In each of these applications, the fresh types generated by the type checker seems to reflect fresh values generated by the program at run time. For example, when generating code as a first-order data structure, we need gensym not only at the type level to ensure lexical scope but also at the value level to make unique concrete variable names. Can dependent types expose and eliminate this reflection for the superfluous encoding that it is? In other words, can our code for generating fresh values be reused during type checking to assure their safety properties more directly? In particular, whereas generated types are only implicitly fresh and are equipped with no operations, generated values are explicitly fresh and are equipped with operations such as comparison. Can we use generated variable names to automate weakening?

Higher-Order Model Checking and Dependent Type Inference

Kobayashi, Naoki (Tohoku University)

We have recently been working to construct a software model checker for a subset of ML on top of a higher-order model checker (or, a model checker for higher-order recursion schemes). In the talk, I plan to discuss what is higher-order model checking, how a software model checker can be constructed, and how it is related to dependent type inference.

Looking at Dialogs as Editing of Objects with a Dependent Type

Nordström, Bengt (Chalmers University of Technology)

A Proof Theoretic Perspective on Dependently Typed Functional Programming

McKinna, James (Radboud Universiteit Nijmegen)

A Canonical Locally Named Representation of Binding

Pollack, Randy (Edinburgh University), joint work with Masahiko Sato

Ez is Easy

Sato, Masahiko (Kyoto University)

A Framework for Extraction of Programs from Proofs Using Postulated Axioms

Anton Setzer (Swansea University), joint work with Chi Ming Chuang

Working with real numbers in dependent type theory is difficult since one usually has to work with computational real numbers, which behave differently from real numbers occurring in set theory. One way to avoid this difficulty is to formulate the reals by postulating axioms stating the existence of real numbers, operations on them and axioms. These real numbers remain abstract. One can define the concrete real numbers, which can be approximated by Cauchy sequences, by signed digit representations, or other methods. These concrete real numbers have computational content. If one shows that the concrete real numbers are closed under certain operations, one obtains functions which operate on the approximations. These allow to compute from Cauchy sequences for the inputs Cauchy sequences for the output, or from signed digit representations of inputs signed digit representations of the output.

This approach generalises as well to other settings. One can for instance abstractly formulate properties of a software system, relate those abstract entities to measurable entities, and therefore obtain computations on those measurable entities.

The question is whether, if one defines functions this way, the operations on the computational values still have computational content. This is not necessarily the case since axioms might prevent elements of algebraic data types from normalising to constructor head normal form. We show that under certain conditions this is not a problem: if the conditions are fulfilled the elements of algebraic data types extracted will always evaluated to constructor head normal form.

In order to formulate this theorem fully mathematical we will introduce a framework of an abstract type theory based on algebraic and coalgebraic data types, recursively defined functions by pattern matching and postulates. This is given by just defining a sequence of judgements (which introduce constants and reductions) and categorising them. We introduce conditions and show that if these conditions are fulfilled normalisation to constructor head normal form is guaranteed.

This theory has been applied to a proof in Agda that signed digit representable real numbers are closed under average, multiplication and the rationals. We were able to feasible compute signed digit representations using this approach using a compiled version of Agda.

Mendler Iteration

Sheard, Tim (Portland State University)

Equations: A Dependently Typed Programming Suite

Sozeau, Matthieu (INRIA)

We present a compiler for definitions made by pattern matching on inductive families in the Coq system. It allows to write structured, recursive dependently-

typed functions, automatically find their realization in the core type theory and generate proofs to ease reasoning on them. The high-level interface allows to write dependently-typed functions on inductive families in a style close to Agda or Epigram, while their low-level implementation is accepted by the vanilla core type theory of Coq. This setup uses the smallest trusted code base possible and high-level tools are provided to maintain a view of the definitions that is abstracted from the low-level manipulations of the compiler. The compiler makes heavy use of type classes and the high-level tactic language of Coq for greater genericity and extensibility.

Ill discuss the issues regarding the treatment of the with rule during program generation and construction of elimination principles and Ill also present our current solution for handling well-founded recursion and the possible design choices available there.

Self-certification: Bootstrapping certified typecheckers in F* with Coq

Swamy, Nikhil (Microsoft Research). Joint work with Pierre-Yves Strub, Cédric Fournet and Juan Chen

Well-established dependently-typed languages like Coq provide a highly reliable way to build and check formal proofs. Researchers have also developed several other dependently-typed programming languages such as Agda, Aura, ATS, Cayenne, Epigram, F*, F7, Fine, Guru, PCML5, Ur, etc., and more are in the works, e.g., Trellys. All these languages shine in their own regard, but they lag behind Coq in the degree of safety provided by their implementations.

This paper proposes a general technique called self-certification that allows a typechecker for a suitably expressive language to be certified for correctness. We have implemented this general technique for F*, a dependently typed language on the .NET platform.

Self-certification (for F*) involves implementing a typechecker for F* in F*, while using all the conveniences F* provides for the compiler-writer (e.g., partiality, effects, implicit conversions, proof automation, libraries). This type checker is given a specification (in F*) strong enough to ensure that it computes valid typing derivations. We obtain a typing derivation for the core typechecker by running it on itself, and we export it to Coq as a type-derivation certificate. By typechecking this derivation (in Coq) and applying the F* metatheory (also mechanized in Coq), we conclude that our type checker is correct. Once certified in this manner, the F* typechecker is emancipated from Coq.

Self-certification leads to an efficient certification schemewe no longer depend on verifying certificates in Coqas well as a more broadly applicable one. For instance, the self-certified F* checker is suitable for use in adversarial settings where Coq is not intended for use, such as run-time certification of mobile code.

When is a Container a Comonad?

Uustalu, Tarmo (Institute of Cybernetics). Joint work with Danel Ahman and James Chapman

Abbott, Altenkirch and Ghani have taught us that a wide variety of set

functors (to be thought of as parameterized types) can be analyzed as containers. A container is given by a set of shapes and a set of positions for every shape. It denotes the functor sending a set to the set of pairs formed of a shape and an assignment of elements from the given set to every position in that shape.

We introduce directed containers to capture the common situation where every position in a datastructure determines another datastructure, informally, the substructure rooted by that position.

We show that every directed container denotes a comonad and that every comonad whose underlying functor is a container is represented by a directed container type. Hence directed containers exactly correspond to those containers that are comonads. We also prove several closure properties of directed containers.

In parallel with developing the theory, we have been formalizing it in Agda.

Almost Everywhere Blue Trees and Bar Induction

Uustalu, Tarmo (Institute of Cybernetics). Joint work with Keiko Nakata and Marc Bezem

Consider infinite binary trees whose nodes are colored red or blue. Do you know how far from being unambigous is the informal idea of an almost everywhere blue tree?

We organize five almost always blueness predicates and their interrelations into a pentagon-shaped diagram. Along the five arcs, the predicates imply each other. The converse implications require additional assumptions: the general fan theorem (FAN), bar induction (BI), the lesser principle of omniscience (LPO) and weak continuity for numbers (WCN); one of them is just false. On a very simple example tree, one of the predicates is undecided intuitionistically, but true resp. false in two consistent but mutually contradictory extensions (LPO, true classically, and WCN, false classically).

We have formalized our theory development in Coq.

Combining Proofs and Programs

Stephanie Weirich (University of Pennsylvania)