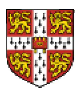
Tools and Techniques for Verification of System Infrastructure

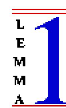
A Festschrift in Honour of Professor Michael J. C. Gordon FRS

The Royal Society, London, 25–26 March 2008

Sponsored by


UNIVERSITY OF CAMBRIDGE
Computer Laboratory

| galois |



Tools and Techniques for
Verification of System Infrastructure

Prof. Michael J. C. Gordon FRS

Foreword

The scope of "Tools and Techniques for Verification of System Infrastructure" (TTVSI) is described by the organisers with the following statement.

> Today's increasingly computer-based society is dependent on the correctness and reliability of crucial infrastructure, such as programming languages, compilers, networks, and microprocessors. One way to achieve the required level of assurance is to use formal specification and proof, and tool support for this approach has steadily grown to the point where the specification and verification of important system infrastructure is now feasible.

This infrastructure creates and runs the computers and communication networks that underlie our world. It can be looked at from a number of perspectives:

- underlying scientific ideas and methods such as logic and mathematics;
- specific theories that support the design of algorithms, protocols etc;
- designs and implementations (hardware and software) that provide services;
- tools that are used to design, build, validate and deploy implementations.

An area where all these are illustrated is the infrastructure needed to maintain the security of online identity and commerce. The underlying theory includes properties of elliptic curves from algebraic geometry, which provide a foundation for the design of cryptographic techniques. Elliptic curve calculations are performed on phones in the course of routine communications. The properties that are relied on by cryptography are very challenging for theorem proving: only recently did Laurent Théry succeed in proving the associativity of addition of elliptic curves using Coq (a challenge posed by Joe Hurd). This may seem like an abstruse achievement, but in fact it is a crucial result needed for verifying properties of encryption systems. The mechanisation of pure mathematics is not an ivory tower activity: it is needed to develop tools for establishing the trustworthiness of concrete devices like phones.

Support for security is only one of many kinds of infrastructure, but all kinds require a combination of science, mathematics, logic and engineering. The design of electronic systems for controlling an aircraft (or automobile or train or medical device) shares with security the need for mathematical theories, though the details of the mathematics differs. The tools used to create and validate designs and their implementations must be trustworthy and the only way to achieve this is to use a rigorous and scientific approach.

The TTVSI organisers have assembled a stunning array of speakers from universities, research laboratories and industry around the world. Presentations range from abstract theory to concrete implementation methods.

I would like to thank Joe, Konrad and Richard for organising this amazing occasion. It is a great honour for me and my family and I hope it will be an inspirational and exciting scientific event for you.

Mike Gordon, University of Cambridge

Preface

A research leader's legacy is not only his or her own research, but also the students and other people he or she has mentored and inspired. As representatives of Mike Gordon's students and Research Associates we have pleasure in presenting these proceedings as a record of the event to mark Mike's 60th birthday and several decades as a leader in his field.

It was Mike's wish for this event to be research focussed, and we have endeavoured to achieve that. We would like to thank all of the invited speakers for agreeing to speak, the poster contributors, and the sponsors of TTVSI, namely University of Cambridge, Galois Inc., and Lemma 1 Ltd, whose support we gratefully acknowledge. We are also grateful to the Royal Society for allowing us to host this event at its premises in London.

Our thanks also to Caroline Matthews, Tom Melham, Andrew Pitts and Carol Speed for their help. Carol and Caroline took on considerable extra work to make this event possible.

Last, but by no means least, we would like to thank Mike himself, whose work we are here to acknowledge. We echo Robin Milner's belief that there is still more to come!

We hope all of the participants find the presentations and discussions stimulating and go away with new ideas and renewed enthusiasm for their research.

Richard Boulton, Icera Inc.
Joe Hurd, Galois Inc.
Konrad Slind, University of Utah

Table of Contents

Invited Talks

Memories and Reflections for Mike Gordon ... 1
Robin Milner

Synthesizing Implementations Using a Theorem Prover ... 3
Mike Gordon

Proof Search Debugging Tools in ACL2 ... 5
Matt Kaufmann, J Strother Moore

A Bit of Social Choice Theory in HOL: Arrow and Gibbard-Satterthwaite ... 9
Tobias Nipkow

Decision Procedures for Parametric Theories ... 11
Sava Krstić, Amit Goel, Jim Grundy, Cesare Tinelli

A Framework for Algorithm Level Hardware Modelling ... 13
Ziyad Hanna, Tom Melham

Micro-architecture Verification, Compiler Verification: What Next? ... 15
Xavier Leroy

Formalizing an Analytic Proof of the Prime Number Theorem ... 17
John Harrison

Network Protocols: The Terror and the Glory ... 23
Peter Sewell (and Adam Biltcliffe, Steve Bishop, Michael Dales, Matthew Fairbairn, Sam Jansen, Michael Norrish, Tom Ridge, Michael Smith, Keith Wansbrough)

Proving and Computing: Certifying Large Prime Numbers ... 25
Laurent Théry

Defining a C++ Semantics ... 27
Michael Norrish

Automation for Interactive Proof: Techniques, Lessons and Prospects ... 29
Lawrence C. Paulson

Poster Abstracts

Which C Semantics to Embed in the Front-end of a Formally Verified Compiler? 31
Sandrine Blazy

Verifying Systems to Reduce Human Error 32
P. Curzon, R. Rukšėnas, J. Back, A. Blandford

Defining Checkability Classes for CIL Bytecode 33
David Greaves

Automatic Extraction of Computed Function (for Verification of Machine Code) 34
Magnus O. Myreen

Ott: Effective Tool Support for the Working Semanticist 35
Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, Rok Strniša

Validating Design Optimisation 36
Kong Woei Susanto, Wayne Luk, Jose Gabriel Coutinho, Tim Todman

How to Prove False Using the Variable Convention 37
Christian Urban

Reasoning about Weak Memory Models 38
Nathan Chong, Samin Ishtiaq

The ARM Instruction Set Architecture in HOL 39
Anthony Fox

A Continuous Relaxation for Proving Discrete ACL2 Theorems over Real Closed Fields 40
Paul B. Jackson, Grant Olney Passmore

A Sound Semantics for OCaml_{light} 41
Scott Owens

A Meta Model for the Formal Verification of Networks on a Chip 42
Julien Schmaltz

Exploring Algebraic Property Dependencies for Metarouting 43
Balraj Singh, Alexander J. T. Gurney, Timothy G. Griffin

A Deep Embedding of a Decidable Fragment of Separation Logic in HOL .. 44
Thomas Tuerk

Invited Talks

Memories and Reflections for Mike Gordon

Robin Milner

University of Cambridge

It is an honour and a pleasure to take part in the event to celebrate Mike Gordon's career (so far; because we expect there is more to come). My memories are about what we did together thirty years ago. My reflections are on what will appear, thirty years hence, to have been the impact of machine-assisted reasoning—especially on computer science itself.

Of course, my main interaction with Mike was in the development of LCF and ML. The LCF system came into being for two reasons. First, I was excited about Scott's domain theory, in particular because you could express both syntax and semantics in Scott's logic for domains, so you ought to be able to prove a compiler correct. Second, when I arrive at John McCarthy's AI lab at Stanford in 1971 there was great concern that someone in the lab should write some software embodying some theory (thus proving that theory is virtuous), in order to loosen the purse strings of funding agencies; my just-mentioned enthusiasm and my new-boy status led me to volunteer ... .

At that time I was impressed by the polarisation of existing software for formal proof. At one pole was the superb work on Automath, by de Bruijn and his group, which checked your own proof construction; at the other pole was the superb development of fully automatic proof based upon Robinson's resolution principle. Surely there was something in between, which was more than taking half of each of these? Malcolm Newey, Richard Weyrauch and I worked in this middle region, and found some embryonic tactics (not really needing a meta language to express them), notably for simplification. We did a few proofs, including the major part of a correctness proof for a very simple compiler. And at just the right time I had to leave Stanford for Edinburgh, so this gave the opportunity for a fresh start.

In Edinburgh from 1974, Malcolm and I—joined by Lockwood Morris—started again, with the obscure idea that you would drive the proof finding in the very functional language whose semantics fits so well in domain theory. We did the first ML implementation, and more—but not many more—proofs. After a year or so my good luck in finding good people persisted at full strength; along came Mike Gordon and Chris Wadsworth. Mike wrote the first semantics of ML in Scott-Strachey style; Chris did a great job of structuring our theories.

Meanwhile, Mike must have been wondering when this great battleship would go to sea and fight some real battles. This was, I think, the real beginning of useful machine-assisted proving, in systems which provably can produce nothing (of the right type) but proofs. Mike observed that there was one kind of proving that our system could do all by itself: simplification by equations. He also observes that correctness of hardware circuits is expressed equationally, and that (therefore) 99% of a hardware correctness proof is by simplification. This was

the killer application that got LCF off the ground. Another thing that got it off the ground was that my student, one Avra Cohn, had to share a room with a stranger: Michael Gordon.

At this point—around 1980—I would like to go fast-forward three decades. We have many wondeful proof systems, and a good few have adopted the LCF principle of a meta-type system that ensured validity of proof. Increasingly, difficult results—old and new—are achieved with these systems. Everyone can see that they are gaining strength year by year. I have done almost no more work in this direction, but I value enormously the experience of doing it thirty years ago, with far-sighted people like Mike Gordon. Instead I have been developing theories of computation. In particular, I work with models of interactive processes that are beginning to be useful in the real world of ubiquitous computing, business processing and even biology. Informatic systems are getting bigger and bigger; correspondingly, in order to understand them, the models that we need to build are not only numerous, but increasingly complex. And, as the human race becomes more critically dependent upon informatic artefacts, the relationships between these models, in what I am calling the **Tower of Informatic Models**, are in greater and greater need of ratification.

I would like to end this note with a conjecture: that, although machine-assisted proof will continue to work in the service of mathematics and other sciences, its main applicability will be in ratifying the tower of informatic models. I know from my own experience that theorems in this region, though not profound mathematically, are an essential part of a profound development of computer science. Like the proof of compiler correctness (indeed, that is one of them!), they are rather straightforward. These proofs will not merely be applied after a model is developed; they will be part of the model-developer's tool kit.

Let me give just one example from my experience. In 1980 I developed CCS, a Calculus of Communicating Systems, which is a *specific* calculus of interactive processes. In the past five years I have developed a *generic* process calculus, based upon a kind of graph. To justify part of the claim that this calculus in indeed generic, I have 'proved' that it recovers the theory of CCS in the following sense: two CCS processes are behaviourally equivalent if and only if their representatives in the generic calculus are equivalent. This proof is detailed, not profound, and quite possibly incorrect (in, one hopes, only minor detail!). My work in developing the generic calculus would have been greatly helped if a proof assistant had been at my finger tips.

So it is not just the *correctness of reasoning*, but the *methodology of theory development*, that is going to gain from the forward march of HOL, Isabelle and other similarly advanced proof engines. The fine state of these systems is due to the insight, courage and technical prowess of a select band of designers. Mike Gordon has been in the vanguard of this group from the beginning.

Synthesizing Implementations Using a Theorem Prover

Mike Gordon
University of Cambridge

I will discuss the use of theorem proving as a way to create implementations (both hardware and software) that are guaranteed to be correct. The talk will contain a historical perspective, an account of some current work and then a general discussion leading to challenges for the future.

Using theorem proving to synthesize implementations is a very old idea. About forty years ago researchers at Stanford (Green and Yates) and Carnegie Mellon University (Waldinger) independently showed that a program to implement a relational input-output specification $R(x, y)$ could be extracted from a first-order resolution proof of $\forall x.\exists y.R(x, y)$ where the axioms used for the proof specify the programming language for the implementation. This idea was the starting point for much fruitful research on deductive program synthesis.

Another approach is to prove $\vdash \forall x.\exists y.R(x, y)$ in a constructive logic and then extract a function f from the proof such that $\forall x.R(x, f(x))$. Unlike the first-order resolution method that aimed to be fully automatic, the constructive logic theorems are proved with human guidance using a proof assistant such as Nuprl (Constable) or Coq (Huet and Coquand).

I will discuss some recent and current work at Cambridge and the University of Utah that has aspects of both these methods. The general idea is that, given a specification S, one proves a theorem $\vdash \mathsf{Implements}(I, S)$ for an appropriate predicate $\mathsf{Implements}$. This theorem is generated automatically from S and contains the desired implementation I as a sub-term. As well as delivering I, the theorem certifies that I meets the specification S. The similarity with the early program synthesis work is that synthesis is automatic and is the side effect of a theorem prover run. However it differs in that the specification S is more algorithmic (a functional program rather than a set of clauses), so generating the theorem resembles compilation and doesn't need artificial intelligence. The similarity with the constructive logic method is that for each S we essentially do a constructive proof of $\vdash \exists I.\mathsf{Implements}(I, S)$, though the witness I is built explicitly by a programmed proof strategy, rather than being an implicit consequence of doing proofs in a constructive system.

The proof of an implementation theorem $\vdash \mathsf{Implements}(I, S)$ can use a number of techniques. One method is to prove a sequence of refinement theorems: $\vdash \mathsf{Implements}(I_1, S) \;\wedge\; \vdash \mathsf{Implements}(I_2, I_1) \;\wedge\; \cdots \;\wedge\; \vdash \mathsf{Implements}(I_n, I)$. Another method, called translation validation, is to use an unverified compiler to create I from S and then to invoke a theorem prover to establish $\vdash \mathsf{Implements}(I, S)$. A third approach is to use a verified compiler $\mathcal{C}$ which has been proved to satisfy the general theorem: $\vdash \forall S.\mathsf{Implements}(\mathcal{C}(S), S)$. These three techniques have differing pragmatic trade-offs. A verified compiler is the best, but proving the compiler correct may be a challenge. Translation valida-

tion can in principle be used with off-the-shelf compilers, but generating the validations may be hard.

The Utah-Cambridge work has been exploring combinations of translation validation and refinement. The overall flow from specification S to implementation I is by proving a sequence of theorems that refine S to I, but particular refinement steps $\vdash \mathsf{Implements}(I_{m+1}, I_m)$ may be accomplished either by applying logical rules (e.g. rewriting I_m) or by conventional compilation of I_m to I_{m+1}, followed by a translation validation. I will discuss examples illustrating both hardware and software synthesis.

This work synthesizes implementations from specifications written directly in logic. This raises the question of whether raw logic is a good specification language. Many theorem proving systems have logics that contain terms corresponding to functional programs: ACL2 supports an applicative subset of Common Lisp; PVS, the various HOL proof assistants, Isabelle, Coq and Nuprl all have terms that correspond to programs in a typed higher order functional programming language. The question, therefore, is related to the utility of languages like Lisp, Scheme, ML or Haskell and, furthermore, whether there is value in having a way to create guaranteed-correct implementations from specifications in these languages.

There is plenty of evidence that functional programming is useful, but only a subset of applications need high assurance of implementation correctness. The application area we have investigated is implementing the arithmetic operations needed for cryptography in both hardware and machine code.

Programs written in real world functional languages like ML and Haskell should be easier to reason about than programs in imperative languages. However, the Milner-Tofte-Harper semantics of Standard ML, though truly an amazing achievement, is probably too complex to be a basis for the mechanical verification of ML programs and, as far as I know, Haskell has no complete mathematical description (though fragments of it do). The functional languages that live inside theorem provers are designed to be suitable for proof, but are primitive. There is a danger that the overlap between real-world functional programming and tractable-for-theorem-proving functional programming is small.

To encourage research in high assurance functional programming it would be valuable to have a tool-independent language with an explicit mathematical definition simple enough to support theorem proving. Programs written in this could then be analysed using PVS, HOL, ProofPower, Isabelle, Coq, Nuprl etc. The way ACL2 created a tractable language from Common Lisp is an inspiring model. Can a tractable language be created from Haskell or ML, or are they too semantically complex? Scott Owens' fully formalized semantics of OCaml light (a subset of OCaml) is thought-provoking. It is intended "to be a suitable substrate for the verification of OCaml programs", but is much more complex than the native languages inside existing theorem provers. It may be that adding new programming-oriented terms to higher order logic is an easier way to get a tractable language than simplifying existing programming languages. Both approaches are worth pursuing and maybe they will eventually meet.

Proof Search Debugging Tools in ACL2

Matt Kaufmann[1] and J Strother Moore[2]

[1] Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA,
kaufmann@cs.utexas.edu,
WWW home page: http://cs.utexas.edu/users/kaufmann
[2] Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712, USA,
moore@cs.utexas.edu
WWW home page: http://cs.utexas.edu/users/moore

Suppose the ACL2 [2] user wishes to prove that reversing a list three times is equivalent to reversing it once, a theorem called `triple-rev` below.

```
(defthm triple-rev
  (equal (rev (rev (rev a)))
         (rev a)))
```

The ACL2 user might think through the argument informally as follows. (We follow ACL2 syntax by using a semicolon (';') to start a comment up to the end of the line.)

> Reversing a list twice is an identity. So `(rev (rev a))` rewrites to `a` and we're done.
>
> To formalize and mechanize this, I'll prove the key lemma, `double-rev`, `(rev (rev x))` = `x`, and have ACL2 use that as a rewrite rule.
>
> Then, when I ask the system to prove `triple-rev`, it will automatically apply the rewrite rule to the underlined subterm below and reduce the left-hand side of `triple-rev` to the right-hand side:
>
> ```
>    (rev (rev (rev a)))      ; lhs of triple-rev
>  =
>    (rev a)                  ; rewritten lhs
>  =
>    (rev a)                  ; rhs of triple-rev
> ```
>
> **Q.E.D.**

Note that this hypothetical user invents `double-rev` and then thinks about how to make ACL2 use it automatically. An alternative would be for ACL2 to require the user to provide a detailed proof.

The basic philosophy behind the ACL2 system is that it should be as automatic as possible, guided by rules derived from definitions and theorems that have already been certified by the system. This is empowering for several reasons. The user is not responsible for soundness: rules in the data base are vetted

by the system using already vetted rules. The user is relieved of much tedious detail: terms and patterns in the goal conjecture trigger the application of vetted rules and the system does this quickly and accurately. In essence, the ACL2 user strives to develop a general and powerful collection of rules for manipulating the concepts in a given class of problems. This gives the system the flexibility often to tolerate minor changes in previously checked work. For example, if ACL2 can prove some property of function `f` and then the definition of `f` is undone and modified in some slight way so that the property still holds, ACL2 is often able to discover the proof "again." (We put the word in quotes because the two proofs are different because the two `f`s are different.) Because models of hardware and software are often modified to reflect changing designs, this ability to "replay" proofs automatically is important.

Returning to the `triple-rev` example, a problem with the ACL2 approach is revealed when we realize that `double-rev` is not a theorem! One can read the prover's output to discover that `(rev (rev x))` is not `x` if `x` is a list structure that terminates with a non-`nil` value. The formula imagined by our hypothetical user is not valid; for example, if `x` is `7`, `(rev (rev x))` is `nil`, not `7`. An accurate formalization of `double-rev` is

```
(defthm double-rev
  (implies (true-listp x)
           (equal (rev (rev x))
                  x))).
```

While this particular mistake could be avoided in a strongly typed system, it illustrates a very common class of mistakes: missing hypotheses. Having discovered this correction to `double-rev` we ought to rethink its intended use in the bigger proof we are constructing.

But people are sloppy. Working top down to prove some main theorem (e.g., `triple-rev`) we invent or remember "lemmas" (e.g., the faulty version of `double-rev`) without getting them exactly right. We imagine the operation of the rules derived from these faulty lemmas and design proof sketches based on these powerful rules. Then we set out to formalize our thinking. In interaction with a theorem prover, we debug statements of our lemmas and catch all the little mistakes. But in the heat of the chase, we forget to carry what we learn back to our main proof sketch. When we finally try to check the main proof, the actual rules carry us astray. The proof fails.

Furthermore, since each newly conjectured lemma requires a proof, the general state of affairs is that the user is in a deep recursive descent through a tree of conjectures rooted in an imagined proof of the main one. Many theorems have been accurately stated and proved while many others remain to be. At any one time, the user is working on the next lemma or theorem and is expecting its formal proof to follow the previously imagined sketch.

In a case like `triple-rev` – where only one new lemma is involved – it might be easy to remember to modify the main theorem or its proof. But in interesting proofs about hardware and software, hundreds of crucial properties and relations might be involved and it is simply impossible to keep them all in mind.

It is illusory to think that the problem is avoided by working bottom up or with a goal manager. Formalization and symbolic manipulation are inevitably carried out late in whatever intellectual process occurs as we think about a given formal challenge. The early stage of proof discovery is necessarily informal since often we must invent concepts that are nowhere formalized or even expressed. There will often be a gap between this early informal stage, where mistakes often occur, and its subsequent mechanized formalization.

It is also illusory to think the problem is solved by printing out everything the system is doing. To give the reader a feel for the size of the problem, consider Liu's M6 model of the Java Virtual Machine [3]. The model is about 160 pages of ACL2 and includes all JVM data types except floats. It models multi-threading, dynamic class loading, class initialization, and synchronization via monitors. We have a data base of about 5,000 rules designed primarily to allow us to construct a proof about an invariant on the class table under dynamic class loading. But the rules are also sufficient to allow us to prove the correctness of simple JVM bytecode programs.

The system can "automatically" prove the correctness of the bytecode produced by the Sun Microsystems `javac` compiler for

```
public static int fact(int n){
    if (n>0)
      {return n*fact(n-1);}
    else return 1;
  }.
```

The statement of correctness is that when the program runs on a non-negative 32-bit Java `int`, n, it terminates and returns the (possibly negative) integer represented in twos-complement notation by the low-order 32-bits of $n!$. Here, $n!$ is the true mathematical value of factorial. To prove this, the system does an induction on n and attempts to apply the 5,000 named rules in its data base. It tries a total of about 63,215 applications to terms in the problem. Only about half are useful. Many of these applications are on branches of the proof search that are ultimately pruned (e.g., simplification done while backchaining on unsuccessful paths). The actual proof involved 180 different rules. The entire search takes 2 seconds on a 2 GHz Intel Core Duo running ACL2 built on SBCL Common Lisp. Furthermore, the system can tolerate minor correctness-preserving changes to the bytecode.

ACL2 prints a description of the evolving proof attempt, but the printed steps are very large. For example, one formula, which takes 67 lines to print, is reported in real time to have simplified to the conjunction of 11 formulas. The 11 formulas take a total of 6,886 lines to print – over 100 pages. This single simplification step is justified by the combined application of 162 different rules. When ACL2 fails or ceases printing, the user can get a rough idea of how things went by navigating through this proof output using text processing tools like Emacs.

The Method described in [1] for debugging failed proofs involves inspecting certain critical subgoals, called *checkpoints*, and looking for subterms within

them that should have been simplified by known rules or that suggest the need for new rules, keeping in mind the informal proof sketch for the goal.

It is not surprising that with so many rule applications and such large formulas, it can be difficult for the user to debug failed proofs. Proof search debugging tools are necessary.

References

1. M. Kaufmann, P. Manolios, and J S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, Boston, MA., 2000.
2. M. Kaufmann and J S. Moore. The ACL2 home page. In *http://www.cs.utexas.edu/users/moore/acl2/.* Dept. of Computer Sciences, University of Texas at Austin, 2008.
3. H. Liu and J S. Moore. Java program verification via a JVM deep embedding in ACL2. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *17th International Conference on Theorem Proving in Higher Order Logics: TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 184–200. Springer, 2004.

A Bit of Social Choice Theory in HOL: Arrow and Gibbard-Satterthwaite

Tobias Nipkow

Institut für Informatik, Technische Universität München

In 1950, Kenneth Arrow, co-recipient of the 1972 Nobel Prize in Economics, proved his famous impossibility theorem [1] about voting systems: under certain minimal plausible fairness conditions, the only voting system left is a dictatorship. In 1973, Gibbard [4] and Satterthwaite [7] independently proved a similar result where nonmanipulabiliy forces a dictatorship. These two theorems are closely related staples of social choice theory [8] and numerous proofs for them have been given in the literature. Yet logicians, as usual, have found fault with (some of) these proofs: "The standard and textbook proofs of Arrow's general impossibility theorem are, like the original proofs, invalid." [6]. This was written 30 yeasr ago but is still valid. In this paper I formalize some recently published proofs of Arrow's theorem [2, 3, 5] in HOL and find that in places they still suffer from opaqueness and missing cases. Then I derive Gibbard-Satterthwaite from Arrow; here the standard construction appears to be free of holes.

References

1. Kenneth Arrow. A difficulty in the concept of social welfare. *The Journal of Political Economy*, 58:328–346, 1950.
2. John Geanakoplos. Three brief proofs of Arrow's impossibility theorem. Technical report, Cowles Foundation Discussion Paper, 2001.
3. John Geanakoplos. Three brief proofs of Arrow's impossibility theorem. *Economic Theory*, 26:211–215, 2005.
4. Allan Gibbard. Manipulation of voting schemes: A general result. *Econometrica*, 41:587–601, 1973.
5. Noam Nisan. Introduction to mechanism design (for computer scientists). In N. Nisan, T. Roughgarden, E. Tardos, and V. Vazirani, editors, *Algorithmic Game Theory*. Cambridge University Press, 2007.
6. R. Routley. Repairing proofs of Arrow's general impossibility theorem and enlarging the scope of the theorem. *Notre Dame Journal of Formal Logic*, 20:879–890, 1979.
7. Mark Satterthwaite. Strategy-proofness and Arrow's conditions: Existence and correspondence theorems for voting procedures and social welfare functions. *Journal of Economic Theory*, 10:187–217, 1975.
8. Alan D. Taylor. *Social Choice and the Mathematics of Manipulation*. Cambridge University Press, 2005.

Decision Procedures for Parametric Theories

Sava Krstić[1], Amit Goel[1], Jim Grundy[1], and Cesare Tinelli[2]

[1] Strategic CAD Labs, Intel Corporation
[2] Department of Computer Science, The University of Iowa

Formal methods for hardware or software development require checking validity (or, dually, satisfiability) of formulas in logical theories modeling relevant datatypes. Satisfiability procedures have been devised for the basic ones—reals, integers, arrays, lists, tuples, queues, and so on—especially when restricted to formulas in some some quantifier-free fragment of first-order logic. Thanks to a seminal result by Nelson and Oppen, these basic procedures can often be modularly combined to cover formulas that mingle several datatypes.

Most research on *Satisfiability Modulo Theories (SMT)* has traditionally used classical first-order logic as a foundation for defining the language of satisfiability procedures, or *SMT solvers*, and reasoning about their correctness. However, the untypedness of this most familiar logic is a major limitation. It unnecessarily complicates correctness arguments for combination methods and restricts the applicability of sufficient conditions for their completeness. Unsurprisingly, researchers have recently begun to frame SMT problems directly in terms of typed logics, and to develop combination results for these logics. Ahead of the theory, solvers supporting the PVS system, solvers of the CVC family, and some others adopted a typed setting early on.

The SMT-LIB standardization initiative also proposes a version of many-sorted first-order logic as an initial underlying logic for SMT. We see this as a step in the right direction, but only the first one, because the many-sorted logic's rudimentary type system is still inadequate for working with typical cases of combined theories and their solvers. For example, in this logic one can define a generic theory of lists using a sort List for the lists and the sort E for the list elements. Then, a theory of integer lists can be defined formally as the union of the list theory with the integer theory, modulo the identification of the sort E with the integer sort of the second theory. This combination mechanism gets quickly out of hand if we want to reason about, say, lists of arrays of lists of integers, and it cannot be used at all to specify *arbitrarily* nested lists. Because of the frequent occurrence of such combined datatypes in verification practice, this is a serious shortcoming.

Fortunately, virtually all structured datatypes arising in formal methods are *parametric*, the way arrays or lists are. Combined datatypes like those mentioned above are constructed simply by parameter instantiation. For this reason, we believe that the logic for SMT should directly support parametric types and, consequently, parametric polymorphism. Our main contribution is a Nelson-Oppen-style framework and results for theories combinable by parameter instantiation.

The key concept of *parametric theory* can likely be defined in various logics with polymorphic types, most easily in the higher-order logic of the *HOL* family

$$\Sigma_{\mathsf{Eq}} = \langle \mathsf{Bool} \mid =^{\alpha^2\to\mathsf{Bool}}, \mathsf{ite}^{[\mathsf{Bool},\alpha,\alpha]\to\alpha}, \mathsf{true}^{\mathsf{Bool}}, \mathsf{false}^{\mathsf{Bool}}, \neg^{\mathsf{Bool}\to\mathsf{Bool}}, \wedge^{\mathsf{Bool}^2\to\mathsf{Bool}}, \ldots \rangle$$
$$\Sigma_{\mathsf{UF}} = \langle \Rightarrow \mid @^{[\alpha\Rightarrow\beta,\alpha]\to\beta} \rangle$$
$$\Sigma_{\mathsf{Int}} = \langle \mathsf{Int} \mid 0^{\mathsf{Int}}, 1^{\mathsf{Int}}, (-1)^{\mathsf{Int}}, \ldots, +^{\mathsf{Int}^2\to\mathsf{Int}}, -^{\mathsf{Int}^2\to\mathsf{Int}}, \times^{\mathsf{Int}^2\to\mathsf{Int}}, \leq^{\mathsf{Int}^2\to\mathsf{Bool}}, \ldots \rangle$$
$$\Sigma_{\times} = \langle \times \mid \langle\text{-},\text{-}\rangle^{[\alpha,\beta]\to\alpha\times\beta}, \mathsf{fst}^{\alpha\times\beta\to\alpha}, \mathsf{snd}^{\alpha\times\beta\to\beta} \rangle$$
$$\Sigma_{\mathsf{Array}} = \langle \mathsf{Array} \mid \mathsf{mk_array}^{\beta\to\mathsf{Array}(\alpha,\beta)}, \mathsf{read}^{[\mathsf{Array}(\alpha,\beta),\alpha]\to\beta}, \mathsf{write}^{[\mathsf{Array}(\alpha,\beta),\alpha,\beta]\to\mathsf{Array}(\alpha,\beta)} \rangle$$
$$\Sigma_{\mathsf{List}} = \langle \mathsf{List} \mid \mathsf{cons}^{[\alpha,\mathsf{List}(\alpha)]\to\mathsf{List}(\alpha)}, \mathsf{nil}^{\mathsf{List}(\alpha)}, \mathsf{head}^{[\mathsf{List}(\alpha),\alpha]\to\mathsf{Bool}}, \mathsf{tail}^{[\mathsf{List}(\alpha),\mathsf{List}(\alpha)]\to\mathsf{Bool}} \rangle$$

Fig. 1. Signatures for theories of some familiar datatypes. The constants' arities are shown as superscripts. Σ_{Eq} contains the type operator Bool and standard logical constants. All other signatures by definition implicitly contain Σ_{Eq}. In Σ_{UF}, the symbol UF is for *uninterpreted functions* and the intended meaning of @ is the function application. The list functions head and tail are partial, so are represented as predicates.

of theorem provers. For the syntax, the simple notion of a theory's *signature* is central. A signature consists of a set of *type operators* and a set of *constants* over this set of operators, as illustrated in Figure 1. Each theory is defined by its semantics—the meaning of type operators as functions that take sets as arguments and produce sets as results, and the meaning of constants as indexed families of functions. (The meaning of cons is the family $\{\mathsf{cons}_E \mid E \text{ is a set}\}$.)

The meanings of type operators and constants must be *parametric* just like the List constructor and the family $\{\mathsf{cons}_E\}$ are parametric. There is an easily perceived uniformity in the way parametric operators and constants compute their results "making no assumptions on their arguments", but the concept is not easy to pin down exactly. We use a definition similar to *Reynolds parametricity* in programming languages.

We can prove that complete solvers for several signature-disjoint theories (sharing only symbols in Σ_{Eq}) can be used in the Nelson-Oppen style to produce a complete solver for the combined theory.

A striking outcome of this result is that in practically oriented SMT research that deals with common datatypes, the vexatious stable infiniteness condition of the traditional Nelson-Oppen approach does not need to be mentioned. Its role is played by a milder flexibility condition that we prove is automatically satisfied for all parametric theories. Thus, our combination results do not subsume existing results nor are subsumed by them. Our results apply more widely because most of the datatypes relevant in applications are described by theories that satisfy our parametricity requirements without necessarily satisfying the stable infiniteness requirements of other combination methods. Our main result about combination of multiple pairwise disjoint parametric theories would be difficult even to state in many-sorted languages, because, as mentioned, many-sorted logic is not well-suited for working with elaborate combinations of theories, while in a logic with parametric types such combinations are straightforward.

A Framework for Algorithm Level Hardware Modelling

Ziyad Hanna and Tom Melham

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX2 0EX, UK

This talk describes work in progress that addresses the increasing challenges of functional validation of computer hardware, typified by modern microarchitecture designs. The growing size and complexity of such systems is outstripping current testing methods, and functional validation has become the main challenge computer engineers face in producing correct and high quality systems [1].

An architecture's *design model*, expressed at register transfer level, usually serves as its de facto specification. It is typically a very detailed model, expressed at a low level of abstraction involving physical design aspects. Validating the design model is very time consuming, because it is entangled with so much non-functional detail—and because the designs themselves are very complex.

Abstraction is commonly invoked to deal with this complexity. The conventional idea is to write abstract, *high-level models* that exclude implementation details to focus on specification and validation. But in practise a high-level model usually serves as a reference for driving further elaboration of the design, as well as for functional validation. It has therefore tended to compromise its role in functional validation by including details driven by implementation concerns. The model's level of abstraction is dragged down, until at some point it is no longer justifiable to maintain it in parallel with the design model. The high-level model is then abandoned, wasting the costly effort spent on its construction.

Another drawback is the lack of a formal link between the high-level model and the design model. Validation work done on a high-level model typically has to be repeated on the design model, so the return on investment in maintaining a high-level model becomes suspect.

The aim of our work is to devise a framework for abstract modelling and high-level validation focusing *solely* on the algorithms in our designs, and to assess the contribution to functional correctness that this strict focus can provide. The semantics of our models is based on Abstract State Machines [2, 5], a natural framework for hardware models. *Algorithm-level models* with this semantics are expressed at a high level of description in a hardware-oriented adaptation of AsmL [6], an executable object-oriented language with rich data types.

The framework we propose provides for both formal property verification, to verify functional correctness of our models, and refinement verification, to connect them through a chain of correctness preserving refinements to design models and ultimately to implementations. It is intended that verification of the refinement relationship between the most detailed model in our framework and the design model will be be done using the *assertion program* approach [7] to GSTE model checking [8].

This talk describes some of the details of our proposed framework, focusing on our implementation of the Asml-based modelling language. Our starting point was to write an interpreter for AsmL-S [6], a simplified version of AsmL with a clearly-defined operational semantics. This was then extended with hardware-specific datatypes and provided with a symbolic simulation facility to underpin verification and refinement.

Although object-oriented in presentation, AsmL is quite naturally embedded in a typed functional setting, in which abstract state is represented mathematically by the elements of types, or sets, and the model's specification of the transitions between states is represented by a function that computes collections of state updates. We have therefore expressed the published semantics of AsmL within a mechanized, but mathematically-precise, language of functions and types—namely the *reFLect* functional programming language [4]. Abstract data types are represented by *reFLect* types; the current state of a model is represented by a collection of (named) values from these types; the transition system is represented by a function that computes updates to the state; and a run of the system is represented by the sequence of successive abstract states.

By situating our semantics in a functional language, we provide a well-defined and directly executable semantics similar in spirit to an embedding in the HOL logic [3]. By using *reFLect*'s inbuilt reflection features, we do this in a way that merges the 'deep' and 'shallow' embedding approaches that have been so extensively investigated in theorem proving research. Values in our system are also represented using reFLect's reflected term language, providing a foundation for verification reasoning by symbolic simulation.

References

1. Bob Bentley. Validating a modern microprocessor. In *Computer Aided Verification: 17th International Conference*, volume 3576 of *LNCS*. Springer Verlag, 2005.
2. Egon Börger and Robert Stark. *Abstract State Machines*. Springer-Verlag Berlin Heidelberg, 2003.
3. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
4. Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, March 2006.
5. Yuri Gurevich. Evolving algebras: An attempt to discover semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, River Edge, NJ, 1993.
6. Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343(3):370–412, 2005.
7. Edward Smith. A method for generation of GSTE assertion graphs. In Robert Jones and Ian Harris, editors, *High-Level Design Validation and Test Workshop, 2005. Tenth IEEE International*, volume 10, pages 160–167. IEEE Computer Society, 2005.
8. Jin Yang and Carl-Johan H. Seger. Generalized symbolic trajectory evaluation - abstraction in action. In Mark Aagaard and John W. O'Leary, editors, *Proceedings of the 4th International Conference on Formal Methods in Computer-Aided Design*, volume 2517 of *LNCS*, pages 70–87. Springer, 2002.

Micro-architecture Verification, Compiler Verification: What Next?

Xavier Leroy

INRIA Paris-Rocquencourt
Domaine de Voluceau, B.P. 105, 78153 Le Chesnay, France
`Xavier.Leroy@inria.fr`

Formal verification of software, using static analysis, model checking, program proof and combinations of these techniques, is slowly getting acceptance in the critical software industry, for example in the case of "fly-by-wire" systems and smart card software. Unlike testing, formal verification can not only show the presence of bugs, but also the absence of whole classes of bugs. To fully realize this potential, however, it is necessary to establish trust in (1) the verification tools themselves (either via tool verification or production of verifiable evidence) and (2) the execution path that leads from the source code of the program or the model thereof used for verification to actual execution of the program. The latter path is long and convoluted, typically involving code generation (from a model written in a domain-specific language), compilation to assembly language (including code optimization passes), assembling and linking to produce machine code, on-the-fly translation of machine code to elementary operations of the processor's micro-architecture, and execution of these operations by hardware circuits. All these steps are potential sources of errors that may cause correct programs to execute incorrectly.

As part of the Compcert project[1], we have been working on the formal verification of a moderately-optimizing compiler for a large subset of the C language down to assembly language for the PowerPC processor [1–3]. Using the Coq proof assistant, we proved a semantic preservation theorem for each pass of the compiler, therefore guaranteeing that the assembly code produced behaves as prescribed by the semantics of the source program. This implies that any property of the observable behavior of the source program carries over to the generated code. Moreover, most of the compiler itself is written directly in the functional subset of the Coq specification language, from which executable Caml code is automatically extracted.

Theorem proving has also been successfully applied to lower layers of the program execution stack. Moore's Piton project [4] proves end-to-end correctness between the Piton assembly-level language and the NDL netlist implementing the FM9001 custom microprocessor. More recently, Fox [5] used HOL to express a full formal specification of the ARM instruction set and to verify the correctness of the ARM6 micro-architecture – an implementation of the ARM architecture that is very widely used in embedded systems. This result is impressive and, as one of its outcomes, provides an invaluable resource: a fully

[1] http://compcert.inria.fr/

formal specification, validated by the correctness proof for a micro-architecture, of a real-world processor architecture – something that processor manufacturers have consistently failed to provide so far.

Much verification work remains to be done on hardware architectures, compilers, code generators, verification tools, and their interfaces. Of particular interest to us is the combination of program provers with verified code generation technology. One approach that we have started to investigate is a fully trusted execution path for programs written and proved correct in Coq, including the verification of Coq's code extraction mechanism, of a compiler for the mini-ML programming language that is the target of this extraction, and of a run-time system (GC and memory allocator) for this compiler. Another approach is the prover-assisted code generation approach of Hurd *et al* [6].

The road towards high-assurance environments for program development, verification and execution is long, but one thing is already clear: proof assistants such as HOL, Isabelle, Coq, etc, have a crucial role to play in such an effort.

References

1. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd symposium Principles of Programming Languages, ACM Press (2006) 42–54
2. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a C compiler front-end. In: FM 2006: Int. Symp. on Formal Methods. Volume 4085 of Lecture Notes in Computer Science, Springer (2006) 460–475
3. Bertot, Y., Grégoire, B., Leroy, X.: A structured approach to proving compiler optimizations based on dataflow analysis. In: Types for Proofs and Programs, Workshop TYPES 2004. Volume 3839 of Lecture Notes in Computer Science, Springer (2006) 66–81
4. Moore, J.S.: Piton: a mechanically verified assembly-language. Kluwer (1996)
5. Fox, A.C.J.: Formal specification and verification of ARM6. In: Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003. Volume 2758 of Lecture Notes in Computer Science, Springer (2003) 25–40
6. Hurd, J., Fox, A., Gordon, M., Slind, K.: ARM verification (abstract). In: High Confidence Software and Systems: HCSS 2007. (May 2007)

Formalizing an Analytic Proof of the Prime Number Theorem *(extended abstract)*

Dedicated to Mike Gordon on the occasion of his 60th birthday

John Harrison

Intel Corporation, JF1-13
2111 NE 25th Avenue, Hillsboro OR 97124, USA
johnh@ichips.intel.com

1 Formalizing mathematics: pure and applied

I've always been interested in using theorem provers both for "practical" applications in formally verifying computer systems, and for the "pure" formalization of traditional mathematical proofs. I particularly like situations where there is an interplay between the two. For example, in my PhD thesis [5], written under Mike Gordon's supervision, I developed a formalization of some elementary real analysis. This was subsequently used in very practical verification applications [6], where in fact I even needed to formalize *more* pure mathematics, such as power series for the cotangent function and basic theorems about diophantine approximation.

I first joined Mike Gordon's HVG (Hardware Verification Group) to work on an embedding in HOL of the hardware description language ELLA. Mike had already directed several similar research projects, and one concept first clearly articulated as a result of these activities was the now-standard distinction between 'deep' and 'shallow' embeddings of languages [3]. Since I was interested in formalizing real analysis, Mike encouraged me to direct my attention to case studies involving arithmetic, and this was the starting-point for my subsequent research. Right from the beginning, Mike was very enthusiastic about my formalization of the reals from first principles using Dedekind cuts. Mike had been involved in Robin Milner's group developing the original Edinburgh LCF [4], a central feature of which was the idea of extending the logical basis with derived inference rules to preserve soundness. Now that Mike had applied the LCF approach to higher-order logic, suitable as a general foundation for mathematics, it was possible to extend this idea and even develop mathematical concepts themselves in a 'correct by construction' way using definitions. So a definitional construction of the reals fitted in very well with the ideals Mike had for the HOL project, an interest in applications combined with an emphasis on careful foundations that has now become commonplace.

In this paper I want to describe a formalization that was undertaken purely for fun, involving complex analysis [8] and culminating in a proof of the Prime

Number Theorem. Nevertheless, it doesn't seem entirely far-fetched to imagine some "practical" applications of this result in the future. For example a weak form of the PNT is implicitly used to justify the termination of the breakthrough AKS primality test [1], and some simpler properties of prime numbers have been used in the verification of arithmetical algorithms by the present author [7]. But I certainly don't need to give any such justification, because Mike Gordon, as well as introducing me to the fascinating world of theorem proving, has always placed a welcome emphasis on doing "research that's fun".

2 Mathematical machinery versus brute force

Formalizing the PNT in itself is not a new accomplishment, since that has already been done very impressively by a team led by Jeremy Avigad [2]. However, that formalization was of the so-called "elementary" Erdös-Selberg proof — elementary in the sense that no higher analysis is used, not in the sense of simplicity. The usual proof in analytic number theory textbooks relies on Cauchy's residue theorem from complex analysis, and the fact that there are no zeros of the Riemann ζ-function for $\Re z \geq 1$. This analytic proof in itself is simpler and clearer than the elementary one, but at the price of requiring much more mathematical "machinery" as a precondition. For this reason, Robert Solovay has suggested an analytic proof of the PNT as a good challenge in the formalization of mathematics [12]. In the full version of this paper we plan to give a more detailed comparison of the elementary and analytic proofs and expand on some of the issues briefly sketched below.

Of course, it might sometimes make sense to play to the particular strengths of computer theorem provers by using a different proof from the one that humans might find appealing; cf. the comments in [13]. For example, see the proof of the Kochen-Specker paradox from quantum mechanics, worked out as an extended example in the present author's HOL Light tutorial. In presenting the proof informally, one would naturally reduce the number of cases by cleverly exploiting symmetry, whereas with a theorem prover it's simpler just to run through the cases by brute force. For a more challenging example, consider proving the associativity of the chord-and-tangent addition operation on elliptic curves. This has been done formally in Coq by Laurent Théry and Guillaume Hanrot [10], with some of the key parts using enormous algebraic computations that were on the edge of feasibility; indeed similar computational issues have obstructed a related project by Joe Hurd. When I mentioned the practical difficulties caused by this example to Dan Grayson, he suggested a more 'human-oriented' proof:

> But why not enter one of the usual human-understandable proofs that + is associative? Too many prerequisites from algebraic geometry? [...] The proof I like most is to use the Riemann-Roch theorem to set up a bijection between the rational points of an elliptic curve and the elements of the group of isomorphism classes of invertible sheaves of degree 0. That's a lot of background theory, probably too much for this stage of

development, but then the "real" reason for associativity is that tensor product of R-modules is an associative operation up to isomorphism.

Indeed, it seems to the present author that formalizing mathematical machinery on that level is probably still many years away. So it is slightly depressing to reflect that we may be forced to formalize unnatural or 'hacky' proofs because not enough people are working on the systematic development of general mathematical machinery. The work reported in this paper is modest by comparison with the reasoning mentioned in that quotation, but still it represents a small step in the direction of formalizing non-trivial proofs in analytic number theory in the style of a mainstream textbook or research paper.

3 Formalizing Newman's proof of the PNT

There are numerous different analytic proofs of the PNT, but there seems to be a general consensus that an approach developed by Newman, just using Cauchy's integral formula for a simple bounded contour, is the simplest known. We took as our text to formalize the book by Newman himself [9], more specifically the "second proof" on pp. 72-74 using the analytic lemma on pp. 68-70. While Newman writes in a friendly and accessible style, he sometimes assumes quite a lot of background or leaves some non-trivial steps to the reader. The overall PNT proof naturally splits up into five parts, which are presented by Newman in somewhat distinct styles and with widely varying levels of explicitness.

1. The Newman-Ingham "Tauberian" analytical lemma.
2. Basic properties of the Riemann ζ-function and its derivative, including the Euler product.
3. Chebyshev's elementary proof that $\sum_{p \leq n} \frac{\log p}{p} - \log n$ is bounded.
4. Application of analytic lemma to get summability of $\sum_n (\sum_{p \leq n} \frac{\log p}{p} - \log n - c)/n$ for some constant c.
5. Derivation from that summability that $\sum_{p \leq n} \frac{\log p}{p} - \log n$ tends to a limit.
6. Derivation of the PNT from that limit using partial summation.

We have compared the main parts of our formalization against reverse-engineered TeX for corresponding passages in Newman's book (thanks to Freek Wiedijk for composing these!) The *de Bruijn factor* [11], the size ratio of the gzipped formal proof text versus the gzipped TeX (gzipped for a crude approximation to 'information content'), varies widely:

Part of proof	dB factor
1 Analytical lemma	8.2
2 ζ-function	81.3
3 Chebyshev bound	28.2
4 Summability	11.0
5 Limit	5.4
6 PNT	30.4

It is commonly found that the de Bruijn factor for typical formalizations is about 4, so these are very high. However, the really high figures are for parts where Newman is not really giving a proof in any sense. The quotations that follow are the sum total of Newman's text for parts 2, 3 and 6, which take over half of the 4939 lines in the complete HOL Light formalization. In no cases can Newman's passage really be called a proof, so the comparison is hardly fair:

2 Let us begin with the well-known fact about the ζ-function: $(z-1)\zeta(z)$ is analytic and zero free throughout $\Re z \geq 1$.

3 In this section we begin with Tchebyshev's observation that $\sum_{p\leq n} \frac{\log p}{p} - \log n$ is bounded, which he derived in a direct elementary way from the prime factorization on $n!$

6 The point is that the Prime Number Theorem is easily derived from '$\sum_{p\leq n} \frac{\log p}{p} - \log n$ converges to a limit' by a simple summation by parts which we leave to the reader.

If we restrict ourselves to parts 1, 4 and 5, the de Bruijn factor is about 8, still higher than normal, but not outrageously so. And indeed, although the proof did not present any profound difficulties, we found that it took more time to formalize Newman's text than we have grown to expect for other formalizations. This may indicate that Newman's style is fairly terse and leaves much to the reader (this does seem to be the case), or that in this area, we sometimes have to work hard to prove things that are obvious informally (this is certainly true for the winding number of the contour mentioned later). For instance, a simple transformation in part 4, reversing the order of summation in this double series for $\Re z > 1$, needed to be justified by a proof, even if not a very difficult one, whereas it is simply posited without comment by Newman:

$$f(z) = \sum_{n=1}^{\infty} \frac{1}{n^z} \Big(\sum_{p\leq n} \frac{\log p}{p} \Big) = \sum_{p} \frac{\log p}{p} \Big[\sum_{n\geq p} \frac{1}{n^z} \Big].$$

To give something of the flavour of the proof, the centerpiece of Newman's approach is the analytical lemma; this is the only part that uses non-trivial facts about the complex numbers and is thus the locus of the analytical 'machinery':

> **Theorem.** *Suppose $|a_n| \leq 1$, and form the series $\sum a_n n^{-z}$ which clearly converges to an analytic function $F(z)$ for $\Re z > 1$. If, in fact, $F(z)$ is analytic throughout $\Re z \geq 1$, then $\sum a_n n^{-z}$ converges throughout $\Re z \geq 1$.*

The proof involves applying Cauchy's integral formula round a contour and then performing some careful estimations of the sizes of the various line integrals involved. The contour we use, traversed counterclockwise, is shown in figure 1; this is slightly different from Newman's (using horizontal straight-line segments rather than continuing the arc of the circle), though only because we found it easier to understand informally, not because of any particular problem of formalization. The place where formalization presents a striking contrast with

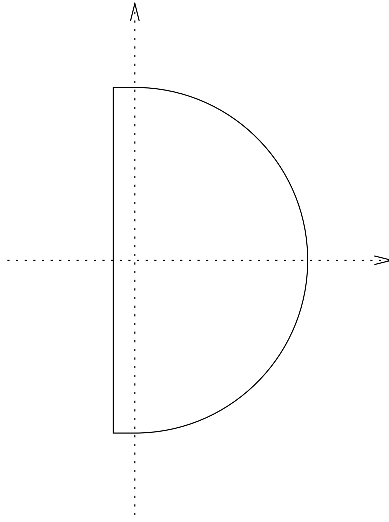


Fig. 1. Contour used in application of Cauchy's integral theorem

informal perception is that in order to apply Cauchy's integral formula, one must verify that the winding number of this contour, formally

$$\frac{1}{2\pi i}\int_\gamma dz/z$$

is indeed 1, indicating that the curve winds exactly once round the origin counterclockwise. Intuitively this is obvious. When formalized in the right way, it is not difficult, but it needs some systematic general lemmas about winding numbers of composite paths.

Nevertheless, despite these reservations, the proof presents no fundamental problems and we finally derive the Prime Number Theorem. The usual informal statement is that $\pi(n) \sim n/\log(n)$, where $\pi(x)$ denotes the number of prime numbers $\leq x$ and '$\sim$' indicates that the ratio of the two sides tends to 1 as $n \to \infty$. In our HOL formalization we do not use the auxiliary concepts $\pi(x)$ and '$\sim$' (though we easily could), but spell things out, where '&' is the type cast $\mathbb{N} \to \mathbb{R}$:

```
|- ((\n. &(CARD {p | prime p /\ p <= n}) / (&n / log(&n)))
    ---> &1) sequentially
```

References

1. M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160:781–793, 2004.

2. J. Avigad, K. Donnelly, D. Gray, and P. Raff. A formally verified proof of the prime number theorem. *ACM Transactions on Computational Logic*, 2007.
3. R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions A: Computer Science and Technology*, pages 129–156, Nijmegen, The Netherlands, 1993. North-Holland.
4. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
5. J. Harrison. *Theorem Proving with the Real Numbers*. Springer-Verlag, 1998. Revised version of author's PhD thesis.
6. J. Harrison. Formal verification of floating point trigonometric functions. In W. A. Hunt and S. D. Johnson, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.
7. J. Harrison. Isolating critical cases for reciprocals using integer factorization. In J.-C. Bajard and M. Schulte, editors, *Proceedings, 16th IEEE Symposium on Computer Arithmetic*, pages 148–157, Santiago de Compostela, Spain, 2003. IEEE Computer Society. Currently available from symposium Web site at `http://www.dec.usc.es/arith16/papers/paper-150.pdf`.
8. J. Harrison. Formalizing basic complex analysis. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 151–165. University of Białystok, 2007.
9. D. J. Newman. *Analytic Number Theory*, volume 177 of *Graduate Texts in Mathematics*. Springer-Verlag, 1998.
10. L. Théry and G. Hanrot. Primality proving with elliptic curves. In K. Schneider and J. Brandt, editors, *Proceedings of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007*, volume 4732 of *Lecture Notes in Computer Science*, pages 319–333, Kaiserslautern, Germany, 2007. Springer-Verlag.
11. F. Wiedijk. The de Bruijn factor. See `http://www.cs.ru.nl/~freek/factor/`, 2000.
12. F. Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
13. L. Wos and G. W. Pieper. *A Fascinating Country in the World of Computing: Your Guide to Automated Reasoning*. World Scientific, 1999.

Network Protocols: The Terror and the Glory

Peter Sewell[1]

Joint work with: Adam Biltcliffe[1], Steve Bishop[1], Michael Dales[2], Matthew Fairbairn[1], Sam Jansen[2], Michael Norrish[3], Tom Ridge[1], Michael Smith[1], and Keith Wansbrough[1]

[1]University of Cambridge [2]Intel Research [3]NICTA, Canberra

The major network protocols that underpin the Internet, such as IP, UDP, TCP, and so on, are remarkable in many ways. On the one hand, they work, on the whole, very well. They originated in the 1970s and 1980s, but thoughtful initial design, and gradual improvements, have allowed the network to cope with many orders of magnitude of expansion since then.

On the other hand, they are highly complex artefacts, and are specified only by a combination of informal prose RFCs and the executable code of the various deployed implementations (it is telling that one of the standard reference works is essentially an annotated walkthrough of the source code of one implementation [1]). There is no precise definition of what the protocols are, or of what it really means for them to work correctly. This complexity and imprecision has a pervasive cost for those who must work with the protocols.

In this talk I will reflect on a line of work, from 2000–2008, in which we produced precise specifications for UDP, TCP, and the Sockets API used by applications to interact with them [2–7]. For TCP, we specified both the low-level protocol, in terms of individual TCP segments on the wire, retransmission, etc. [5, 6], and the functional specification of what the protocol should guarantee, in terms of reliable byte streams [7].

Taking the deployed implementations as primary, the relationship between the low-level specification and (selected) implementations was validated experimentally, by testing that traces collected from an instrumented real-world network were admitted by the specification. A precise abstraction function between the two specifications was defined, and likewise experimentally validated. These validations are a pragmatic alternative to full proof, which on this scale (tens of thousands of lines of specification and code, multi-threaded and time-dependent) would be challenging.

Throughout, the HOL proof assistant, developed by Mike Gordon and his colleagues [8, 9], provided vital tool support. Our specifications are expressed in HOL, and our validation process involves special-purpose symbolic evaluators within HOL, automatically producing, for each real-world trace, a fully expansive proof that that trace is admitted by the specification.

This work shows how the complexities of extant real-world systems infrastructure can be addressed with machine-supported semantics. However, the deployed base of the existing protocols makes it hard to simplify that complexity.

For newly designed systems, where protocol design, specification, testing, and proof can be intertwined, one might hope to have even greater benefits from rigorous specifications, and at much less cost. As an experiment along these lines, we specified a MAC protocol for the SWIFT optically switched network, in collaboration with the network and protocol designers [10], validating the specification against their FPGA and ns2 implementations. Working at design-time, rather than 30 years after the fact, did indeed bring the expected benefits.

Together, these works confirm that working rigorously with real-world system designs, using tools such as HOL, is both feasible and desirable. There remains a cultural problem, of how one can stimulate the formation of the mixed collaborations that are needed for such work, to introduce rigorous semantics at the early stages of system design.

References

1. Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated Vol. 2: The Implementation.* 1995.
2. Andrei Serjantov, Peter Sewell, and Keith Wansbrough. The UDP calculus: Rigorous semantics for real networking. In *Proceedings of TACS 2001: Theoretical Aspects of Computer Software (Sendai), LNCS 2215*, pages 535–559, October 2001.
3. Keith Wansbrough, Michael Norrish, Peter Sewell, and Andrei Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *Proceedings of ESOP 2002: the 11th European Symposium on Programming (Grenoble), LNCS 2305*, pages 278–294, April 2002.
4. Michael Norrish, Peter Sewell, and Keith Wansbrough. Rigour is good for you, and feasible: reflections on formal treatments of C and UDP sockets. In *Proceedings of the 10th ACM SIGOPS European Workshop (Saint-Emilion)*, pages 49–53, September 2002.
5. Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In *Proceedings of SIGCOMM 2005: ACM Conference on Computer Communications (Philadelphia),* published as Vol. 35, No. 4 of *Computer Communication Review*, pages 265–276, August 2005.
6. Steven Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proceedings of POPL 2006: The 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston)*, pages 55–66, January 2006.
7. Tom Ridge, Michael Norrish, and Peter Sewell. A rigorous approach to networking: TCP, from implementation to protocol to service. In *Proc. FM'08: 15th International Symposium on Formal Methods (Turku, Finland).*
8. M. J. C. Gordon and T. Melham, editors. *Introduction to HOL: a theorem proving environment.* Cambridge University Press, 1993.
9. The HOL 4 system, Kananaskis-3 release. http://hol.sourceforge.net/.
10. Adam Biltcliffe, Michael Dales, Sam Jansen, Thomas Ridge, and Peter Sewell. Rigorous protocol design in practice: An optical packet-switch MAC in HOL. In *Proceedings of ICNP, the 14th IEEE International Conference on Network Protocols (Santa Barbara)*, November 2006. 10pp.

Proving and Computing: Certifying Large Prime Numbers

Laurent Théry

Marelle Project INRIA, France
Laurent.Thery@inria.fr

One of most satisfying aspect of proof systems like Coq [7] is the capability to combine proving and computing freely. Programs are purely functional with a syntax close enough to OCAML [5]. Following LCF style, proofs are built using tactics but internally consist in exhibiting an object (a program) of a given type. This type represents the proposition to be proved. Altogether, this gives a very simple framework where proving and computing are unified. In this presentation, we show how this combination is crucial in order to get an effective way of certifying the primality of relatively large numbers with a very high level of confidence. Also, we believe that this application illustrates four interesting aspects of using proof systems to do formal verifications.

First of all, with respect to formal verification, we are in an ideal situation. Defining what a prime number is can be done in 5 definitions only: one for natural numbers, one for addition, one for multiplication, one for divisibility, and finally one for primality. So no matter how elaborate our techniques to assert primality are, the statement we prove at the end is always of the form `prime p`, so fundamentally depends of these 5 definitions only.

Second, we have tried to minimise as much as possible the proving part. Theorem proving is a time consuming activity and the evaluation of our programs inside the prover is an order of magnitude slower than the one of real-life programming languages. Instead of trying to find prime numbers, we only check inside the prover that a given number is prime. Furthermore, we make intensive use of certificates, generated outside the prover, to simplify this checking phase. For primality, there exists a variety of certificates [1]. Pocklington certificates are used for numbers n such that $n-1$ can be easily factorised. Lucas test is the most effective way of proving primality of Mersenne numbers, i.e. numbers of the form 2^p-1. Finally, elliptic curves are used for arbitrary numbers [3].

Third, this application gave us the opportunity to revisit in a formal setting some standard purely functional datastructures [6]. The definition of primality uses Peano numbers to represent natural numbers. For example, 3 is represented as `(S (S (S O)))`. This unary representation has the advantage of simplicity but is almost useless for computing. Representing numbers as lists of booleans is a bit better. Still, some simple operations like getting the leading bit requires linear time. In our application, we use binary trees to represent numbers instead. This gives the possibility to apply divide-and-conquer strategies to implement the usual arithmetic operations. For example, multiplication is implemented using the well-known Karatsuba algorithm [4].

Finally, formalising certificates based on elliptic curves has revealed to be a non-trivial task. The actual problem is in proving that an elliptic curve with the usual composition law forms a group. In particular, the associativity of this law is the difficult part. For this, we took an elementary road that requires some intensive polynomial computations [2] and amounts to developing a small certified kernel of computer algebra system inside the prover. This illustrates again the ubiquity of computing.

References

1. J. Brillhart, D. H. Lehmer, and J. L. Selfridge. New primality criteria and factorizations of $2^m \pm 1$. *Mathematics of Computation*, 29:620–647, 1975.
2. Stefan Friedl. An elementary proof of the group law for elliptic curves. Excerpt from undergraduate thesis, Regensburg (1998).
3. Shafi Goldwasser and Joe Kilian. Almost all primes can be quickly certified. In *Proceedings of the 18th STOC*, pages 316–329, 1986.
4. Anatolii A. Karatsuba and Yu Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Soviet Physics-Doklad*, 7:595–596, 1963.
5. Xavier Leroy. Objective Caml. Available at http://pauillac.inria.fr/ocaml/, 1997.
6. Chris Okasaki. *Purely Functional Datastructures.* Cambridge University Press, 1998.
7. The COQ development team. The Coq Proof Assistant Reference Manual v7.2. Technical Report 255, INRIA, 2002. Available at http://coq.inria.fr/doc.

Defining a C++ Semantics

Michael Norrish

Canberra Research Lab., NICTA

Introduction In 2005, I was approached by QinetiQ to write a semantics for C++ in the style of the C semantics I wrote for my PhD [2]. After some discussion, we decided that I could concentrate on the dynamic semantics, and that I might ignore as much of the (complicated) static semantics as I liked. In particular, I decided that this allowed me to ignore the complicated rules for operator and function overloading. This overloading is entirely resolved at compile-time and might in this way be thought of as a static matter.

I did commit to describing multiple inheritance, exceptions, templates (for all that these are also a "compile-time" feature), object lifetimes and namespaces. This abstract highlights some of the issues encountered in performing this task. The result of the commission was a document summarising the semantics, and some 12 000 lines of HOL4 sources.

From Naïvety to Namespaces One can divide C's variables into two sorts: stack-allocated locals and globals of permanent lifetime. (A variable's visibility is an orthogonal issue.) Dynamically accessing a variable is simple: it must either be a local, necessarily masking earlier locals and globals, or it is a global. Because of this, a C semantics can maintain two maps for variables, its globals and its current variables. When a function is entered, one sets the current map to be the global map, and then subsequently updates this with any further local variables (including function parameters) that are encountered, masking any other variables of the same name.

In the presence of namespaces (or classes), this simplistic scheme falls apart. Consider Figure 1: on entering function `ns1::f`, one cannot simply use the names belonging to namespace `ns1`. Done incorrectly, this will change the `x` reference in `f` to point to the wrong variable. When one includes the strange things that happen to names in the presence of class declarations, the underlying semantics becomes even more complicated. Eventually, this was modelled with an explicit "name resolution" phase.

```
int x = 3;
namespace ns1 {
  int f(int n) { return n + x; }
  int x = 2;
}
```

```
val x = ref 3
structure ns1 = struct
  fun f n = n + !x
  val x = ref 2
end
```

Fig. 1. C++ program (and SML equivalent) illustrating name resolution.

Non-Interleaving Function Calls in a Small-step Style The C++ Standard [1] (following C) insists on peculiar rules for the order of evaluation for expressions. These effectively require that expression evaluation be defined in a small-step style. Unfortunately, at the same time, the standard insists that function calls do not interleave, which is easiest to capture in a big-step semantics. The earlier C semantics in [2] merged these styles in an ugly way. The new C++ approach uses a mixture of continuations and traditional recursion on the syntactic structure of expressions. This isn't without its uglinesses; it's possible that a purely continuation-based approach would be cleaner. Unfortunately, continuations can themselves introduce significant clutter, as well as some incomprehensibility.

Validation and Proofs: the Ghost at My Banquet The major problem with a semantics as large as this one is that it is hard to judge its correctness. Proofs of type soundness and progression are often used to provide some form of validation, but in the absence of a good mechanisation of the type system, this option was not available with this semantics. In the case of C++, the more important question is whether or not the semantics corresponds to the "real" definition of the language. There are no deep proofs in the semantics, but there are some "sanity" theorems of various forms.

Other Features Two significant pieces of existing work enabled important parts of the semantics to be taken "off the shelf". In particular, the treatment of multiple inheritance followed that given in the paper by Wasserrab *et al* [4]. The only significant adjustment required when incorporating this material was to remove the Java-style assumption that "everything is a pointer to something on the heap". Capturing the layout of objects with `virtual` ancestors was particularly involved.

Secondly, the semantics also drew on the description of templates in Siek and Taha [3]. C++ templates are one of the language's most complicated features, and are modelled as a separate phase that happens before any dynamic behaviour occurs.

References

1. *Programming Languages—C++*, 2003. ISO/IEC 14882:2003(E).
2. Michael Norrish. *C formalised in HOL*. PhD thesis, Computer Laboratory, University of Cambridge, 1998. Also available as Technical Report 453 from `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-453.pdf`.
3. Jeremy Siek and Walid Taha. A semantic analysis of C++ templates. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 304–327. Springer, 2006.
4. Daniel Wasserrab, Tobias Nipkow, Gregor Snelting, and Frank Tip. An operational semantics and type safety proof for multiple inheritance in C++. In *OOPSLA '06: Object oriented programming, systems, languages, and applications*. ACM Press, 2006. Available from `http://isabelle.in.tum.de/~nipkow/pubs/oopsla06.html`.

Automation for Interactive Proof: Techniques, Lessons and Prospects

Lawrence C. Paulson

Computer Laboratory, University of Cambridge, England

LP15@cam.ac.uk

Introduction. The idea of supporting interactive provers using automatic theorem provers (ATPs) is an old one. An important early effort is the KIV system [1], which has been integrated with 3TAP. Hurd has written his own resolution prover, Metis, and integrated it with HOL4 [3].

The *sledgehammer* tool, introduced in Isabelle2007, largely eliminates the need for problem preparation by automatically removing higher-order features and collecting relevant lemmas before it calls ATPs. It exploits multi-core architectures.

Integration. Sledgehammer integrates Isabelle with the automatic provers E, SPASS and Vampire. Other provers can easily be added to this list. Claire Quigley implemented background processing so that users are not forced to wait for a response. We have true "one-click" invocation: the system examines Isabelle's full lemma library and selects lemmas that appear relevant to the problem [5]. Higher-order problems are transformed into first-order ones using a novel and effective translation [4]. We relied on systematic, extensive experimentation to fine-tune many parameters of the system. This integration is both original and highly usable.

Proof reconstruction. Proof reconstruction is based on Hurd's Metis prover [3]. Susanto [6] integrated the Metis prover with Isabelle, including proof reconstruction, following the existing HOL4 integration. Sledgehammer translates the output produced by E, SPASS or Vampire and generates one or more Metis calls that prove the required theorem. These proof scripts allow Isabelle to reconstruct the proofs, and allow proofs to be re-run without repeating the expensive ATP calls.

Relevance filtering. Jia Meng devised methods for selecting, from a huge lemma library, the few lemmas relevant to a given problem [5]. Relevance filtering is necessary because automatic provers deliver poor results when given the standard lemma collection (consisting of a few hundred theorems) used with Isabelle's own automatic tools. We have developed and evaluated many strategies based on occurrences of constants in lemmas. Our methods work well enough that we can now use them with Isabelle's full lemma library of 7000 theorems.

Higher-order translations. Jia Meng examined many approaches to translating higher-order problems to first-order logic [4]. The complexity of Isabelle's type system precludes using an untyped translation as Hurd did, so the question is how much type information to retain. For the removal of λ-abstractions, we have compared combinators with λ-lifting. We have succeeded in finding a translation that delivers a good success rate.

Experiences. Sledgehammer cannot prove really difficult statements. It generally proves theorems that are simple consequences of other theorems already in the Isabelle library. Even this can replace an hour of tedious mental work. Thus it is valuable for beginners, but some Isabelle veterans now depend on it. Concerning future work, the clearest needs are support for arithmetic and integration with the automatic higher-order theorem prover LEO-II [2].

Acknowledgements. Jia Meng, Claire Quigley and Kong Woei Susanto worked for the project and made major contributions to it. The research was funded by the EPSRC grant GR/S57198/01 *Automation for Interactive Proof.*

References

1. Wolfgang Ahrendt, Bernhard Beckert, Reiner Hähnle, Wolfram Menzel, Wolfgang Reif, Gerhard Schellhorn, and Peter H. Schmitt. Integrating automated and interactive theorem proving. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction— A Basis for Applications*, volume II. Systems and Implementation Techniques, pages 97–116. Kluwer Academic Publishers, 1998.
2. Christoph Benzmüller, Lawrence C. Paulson, Frank Theiß, and Arnaud Fietzke. LEO-II - a cooperative automatic theorem prover for higher-order logic. submitted, 2008.
3. Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, September 2003.
4. Jia Meng and Lawrence C. Paulson. Translating higher-order problems to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, January 2008.
5. Jia Meng and Lawrence C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. *Journal of Applied Logic*, in press.
6. Lawrence C. Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics: TPHOLs 2007*, LNCS 4732, pages 232–245. Springer, 2007.

Poster Abstracts

Which C Semantics to Embed in the Front-end of a Formally Verified Compiler?

Sandrine Blazy

ENSIIE, Sandrine.Blazy@ensiie.fr

We have been developing and formally verifying in Coq a moderately optimising compiler (called Compcert) for a large subset of the C language. This compiler comprises a back-end translating the Cminor intermediate language to PowerPC assembly code [3] and a front-end translating the Clight subset of C to Cminor. Clight features all the types and operators of C as well as all the structured control statements of C, but excludes unstructured control.

We have re-architected a previous front-end [1] around the use of the CIL library [2]. CIL provides an industrial-strength parser and type-checker for the C language, as well as a simplifier that eliminates or explicates many features of this language. CIL is written in Caml and is also used in other tools dedicated to the verification of C programs. As CIL performs too many simplifications, we have deactivated those that are not wanted in the context of a verified compiler.

Our formalisation of C in Coq has been extended in two ways. Firstly, the abstract syntax describes a larger subset of C including the C `struct` and `union` types as well as a limited `switch` statement. The representation of recursive `struct` and `union` types relies on a fixpoint operator. The `switch` statement is defined for the three languages of the front-end. Secondly, the semantics of C is defined coinductively using natural semantics rules for divergence, thus modelling non-terminating programs. The proofs of semantic preservation of the front-end have also been reused and extended in order to handle these changes.

The main difficulty while designing our semantics of Clight was to find the right level of abstraction between on the one hand a precise semantics enabling the proof of correctness properties of non-trivial code transformations as performed by a compiler, and on the other hand a semantics that is less strict than the C standard. For example, thanks to an abstract memory model [4], some popular violations of the C standard are specified in our semantics, but many other violations cannot be accounted for. As a result of this work, the Compcert compiler is now able to compile some realistic examples of C source code.

References

1. Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM*, volume 4085 of *LNCS*, pages 460–475, 2006.
2. George C. Necula et al. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, 2002.
3. Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM Press, 2006.
4. Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. To appear in JAR, 2008.

Verifying Systems to Reduce Human Error

P. Curzon[1]*, R. Rukšėnas[1], J. Back[2], and A. Blandford[2]

[1]Queen Mary, University of London, [2]University College London, Interaction Centre

Our computer-based society is dependent on the reliability of both computers and their operators. Steps of a learned task can have properties that make them hard to remember, triggering cognitive slip errors even when individuals have the expert knowledge to perform a task. Manifestations include omission errors (e.g., forgetting the original after making photocopies), and mode errors (e.g., typing with Caps Lock on). Most errors only result in minor annoyance but they can be catastrophic in safety-critical situations. System infrastructure must be designed to be tolerant of operator slips. Consequently formal verification technology must be able to detect design problems that lead to systematic human error. To explore these issues, we have combined formal verification with empirical investigation of the causes of human error.

We initially developed a simple higher-order logic model of cognitively plausible behaviour. It incorporated features of human behaviour such as reactive, goal-based and termination behaviour embedded in a non-determinsitc framework of action. A series of case studies showed that, even with a simple model, a variety of systematic errors emerge that can be detected by automated verification tools, and that the model can be used to verify design rules. We have further shown how to use the model in a scenario based way demonstrating that various user classes (e.g., beginners and experts) can be represented using different instantiations of the generic user model.

Our laboratory studies identified and manipulated workload variables that influenced the strength of procedural and sensory cues. We found that if a cue was not strong enough to trigger a subsequent step then slip errors were more likely. The understanding gained led to a formulation of the abstract dependencies between salience and cognitive load. We extended our user model based on this formulation, using it as a tool to analyse the dependencies. The analysis yielded more detailed dependencies in the form of formal, generic rules and thus a deeper understanding of the experimental results. In particular, the new rules refine the use of non-determinism, by introducing a hierarchy of choices governed by the salience (strength) of procedural and sensory cues, and the level of cognitive workload imposed by the task performed.

The refined model can be used to verify interactive systems with respect to systematic human error in greater detail. The verification work has also suggested areas where further experiments are needed to fill gaps in our understanding of the causes of slips and so of the formal model. Thus the experimental and verification work have been mutually beneficial, each leading to deeper insights in the other.

* This research is funded by EPSRC grants GR/S67494/01 and GR/S67500/01. See http://www.dcs.qmul.ac.uk/research/imc/hum/

Defining Checkability Classes for CIL Bytecode Extended Abstract

David Greaves

University of Cambridge, Computer Laboratory

Standard CIL bytecode is used in the Mono and .net systems [ECMA-334]. We use CIL to reflect the behaviour of embedded firmware so that interactions between pervasive computing devices may be formally verified. We also embed assertions in the bytecode that must hold when a device joins a community and which can ensure safe operation under network failure.

In our approach, space is divided into domains and devices are prevented from sending commands over the network in a domain until their internal applications, the *canned application bundles*, have been validated by a domain manager. In order to do this, they offer an *application digest* which is a description of their active behaviour, so that automated reasoning techniques can be run before granting a bundle the right to send commands. There are many potential forms of application digest: we investigate the costs of reflecting the key behaviour of the device using CIL (.net) bytecode, lightly embedded in XML, over HTTP. Accompanying the program is a classification of the *complexity* of the program. Currently, the complexity classification we are using is that the program describes a finite state machine, and hence is of a coding style that can be readily converted to a hardware logic design by certain toolchains used in design automation. In the future, we expect complexities to be arranged in a partial order, with a given automated checker at the domain manager being able to check all programs below and including a given complexity. This assumption motivates the use of CIL bytecode as the description language. Given the relatively long lifetime of certain hardware devices, such as control valves and television sets, we expect checking capabilities to grow greatly while they are deployed. Using a scale of complexity profiles over a full language, such as CIL bytecode, we provide futureproofness.

Our checker covers the major forms of network-level error, which arise from lost packets, network segmentation and device disconnections. A related refinement is network latency: where a safety condition is a function of conditions at different locations, we can check consistency predicates of the make-before-break and break-before-make form.

Our work contrasts with work that proves the correctness of individual network protocols in isolation from the application behaviour. Our approach spots feature and deadlock interactions between actual applications and is then refined to encompass networking protocol errors. Although we have performed checks by essentially converting the aggregate collection of applications into a global LTS, reflection using CIL is sufficiently general that other forms of automated reasoning are not precluded.

Automatic Extraction of Computed Function (for Verification of Machine Code)

Magnus O. Myreen

Computer Laboratory, University of Cambridge

We have developed a total-correctness Hoare logic [3] which can be used to reason about machine code. The Hoare logic has been instantiated to a detailed model of the ARM instruction set [2]. We plan to also instantiate it to a detailed model of x86 developed by Sarkar [1]. The logic has been used in automation [4], which extracts the function computed by machine code, e.g. given the code:

```
L: cmp   r1,#10          | compare r1 with 10
   subcs r1,r1,#10       | subtract 10 from r1,  if cmp gave 10 ≤ r1
   bcs   L               | jump to top,          if cmp gave 10 ≤ r1
```

Symbolic simulation gives the following. Let $\mathsf{r1}\ x$ assert: register 1 has value x.

$$x < 10 \;\Rightarrow\; \{\, \mathsf{r1}\ x * \mathsf{pc}\ p \,\}\ \ code\ \ \{\, \mathsf{r1}\ x * \mathsf{pc}\ (p{+}12) \,\}$$
$$10 \leq x \;\Rightarrow\; \{\, \mathsf{r1}\ x * \mathsf{pc}\ p \,\}\ \ code\ \ \{\, \mathsf{r1}\ (x{-}10) * \mathsf{pc}\ p \,\}$$

A function f is constructed which mimics the effect of the code.

$$f(x) = \mathsf{if}\ x < 10\ \mathsf{then}\ x\ \mathsf{else}\ f(x{-}10)$$

We prove automatically that f executes the given code:

$$\{\, \mathsf{r1}\ x * \mathsf{pc}\ p \,\}\ \ code\ \ \{\, \mathsf{r1}\ f(x) * \mathsf{pc}\ (p{+}12) \,\}$$

The proof uses the induction arising from the definition of f:

$$\forall P.\ \ (\forall x.\ (10 \leq x \Rightarrow P(x{-}10)) \Rightarrow P(x)) \;\Rightarrow\; (\forall x.\ P(x))$$

Our automation has been used in the proof of a Cheney garbage collector, which we are currently integrating into a proof-producing compiler.

References

1. Karl Crary and Susmit Sarkar. Foundational certified code in a metalogical framework. Technical Report CMU-CS-03-108, Carnegie Mellon University, 2003.
2. Anthony Fox. Formal specification and verification of ARM6. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS. Springer, 2003.
3. Magnus O. Myreen and Michael J.C. Gordon. A Hoare logic for realistically modelled machine code. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS. Springer-Verlag, 2007.
4. Magnus O. Myreen, Konrad Slind, and Michael J. C. Gordon. Tractable machine-code verification using extraction of computed function. Submitted to *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS. Springer, 2008.

Ott: Effective Tool Support for the Working Semanticist

http://www.cl.cam.ac.uk/users/pes20/ott

Peter Sewell[1], Francesco Zappa Nardelli[2], Scott Owens[1], Gilles Peskine[2], Thomas Ridge[1], Susmit Sarkar[1], and Rok Strniša[1]

[1] University of Cambridge
[2] INRIA

Ott [3] is a tool for writing definitions of programming languages and calculi. It takes as input a definition of a language syntax and semantics, in a concise and readable ASCII notation that is close to what one would write in informal mathematics. It generates LaTeX to build a typeset version of the definition, and Coq, HOL, and Isabelle/HOL versions of the definition. Additionally, it can be run as a filter, taking a LaTeX/Coq/Isabelle/HOL source file with embedded (symbolic) terms of the defined language, parsing them and replacing them by target-system terms.

Most simply, the tool can be used to aid completely informal LaTeX mathematics. Here it permits the definition, and terms within proofs and exposition, to be written in a clear, editable, ASCII notation, without LaTeX noise. It generates good-quality typeset output. By parsing (and so sort-checking) this input, it quickly catches a range of simple errors, e.g. inconsistent use of judgement forms or metavariable naming conventions.

That same input can be used to generate formal definitions, for Coq, HOL, and Isabelle. It should thereby enable a smooth transition between use of informal and formal mathematics. Additionally, the tool can automatically generate definitions of functions for free variables, single and multiple substitutions, subgrammar checks (e.g. for value subgrammars), context application functions, and binding auxiliary functions. (At present only a fully concrete representation of binding is supported, without quotienting by alpha equivalence.)

The distribution includes several examples, in varying levels of completeness: untyped and simply typed lambda-calculus, a calculus with ML polymorphism, the POPLmark Fsub with and without records, an ML module system taken from (Leroy, JFP 1996) and equipped with an operational semantics, and LJ, a lightweight Java fragment. More substantially, Ott has been used for work on LJAM: a Java Module System, by Rok Strniša [2], and semantics for OCaml light, by Scott Owens [1], both with mechanized soundness proofs.

1. Scott Owens. A sound semantics for OCaml light. In *Proc. ESOP'08, 17th European Symposium on Programming (Budapest)*, March/April 2008.
2. Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java Module System: core design and semantic definition. In *Proc. OOPSLA*, 2007.
3. Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proc. ICFP*, 2007.

Validating Design Optimisation

Kong Woei Susanto, Wayne Luk, Jose Gabriel Coutinho, and Tim Todman

Department of Computing,
Imperial College London, UK
{kws,wl,jgfc,tjt97}@imperial.ac.uk

Programming languages such as C are beginning to be employed for hardware development, since they are widely used in software development and many tools are available. They also enable implementation of various design optimisations.

This poster presents work-in-progress on involving formal validation approaches in our tool chain for heterogeneous reconfigurable systems. The tool chain supports mapping a design, captured in C, to a heterogeneous platform containing an instruction processor, a digital signal processor, and an accelerator based on field-programmable technology. Formal validation techniques are explored in two aspects: the correctness of the optimised design with respect to the original design, and the correctness of the optimisation rules themselves.

In our approach, an optimisation rule is specified in the language CML [1–3] as a source-to-source transformation. A CML rule contains three parts: pattern, condition, and result. *Pattern* is a description of the design before it is optimised. *Condition* is the presumption that underpins the validity of the optimisation. *Result* is a decription of the optimised design. We are investigating the use of an interactive prover for validating the optimisation rules.

The optimisation rules are implemented in the CML compiler. Such rules should only improve design performance, without changing its functional behaviour. CML transformations involve the use of annotations to capture the optimisation. Our approach is based on methods for producing formal descriptions [4, 5] of the design before and after optimisation, such that the behaviour of such formal descriptions can be analysed by the ACL2 Symbolic Simulator.

References

1. A.Brown, W.Luk, and P.H.J.Kelly: Generating Hardware Designs by Source Code Transformations, in *Proc. STS*, 2006.
2. M.Boekhold, I.Karkowski, H.Corporaal, and A.Cilio: A Programmable ANSI C Transformation Engine, in *Proc. ICCC*, 1999.
3. Tim Todman, Jose Gabriel de Coutinho, and Wayne Luk: Customisable Hardware Compilation, in *Journal of SuperComputing 32*, 2005.
4. D.Borrione, P.Georgelin, and V.Rodrigues: Using Macros to Mimic VHDL, in *Computer-Aided Reasoning: ACL2 Case Studies - Chp. 11*, 2000.
5. J.Moore: Symbolic Simulation: An ACL2 Approach, in *Proc. FMCAD*, 1998.

How to Prove False Using the Variable Convention*

Christian Urban
TU Munich (`urbanc@in.tum.de`)

Abstract. Bound variables play an important role in many branches of formal methods. Nearly all informal proofs involving bound variables make use of the variable convention. This poster shows by giving an example that this convention is in general an unsound reasoning principle, i.e. one can use it to prove false.

Summary of the Poster's Content

The variable convention is perhaps one of the most frequently used reasoning principles when reasoning informally about syntax involving binders. Barendregt formulates this convention in his classic book as follows:

> **Variable Convention:** If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.

However this convention is in general an unsound reasoning principle. To see this, consider the following inductively defined (two-place) relation taking two α-equated lambda-terms as arguments:

$$\frac{}{x \mapsto x}\ \text{Var} \qquad \frac{}{t_1\, t_2 \mapsto t_1\, t_2}\ \text{App} \qquad \frac{t \mapsto t'}{\lambda x.t \mapsto t'}\ \text{Lam}$$

Note that the last rule reads "for all x, t and t', if $t \mapsto t'$ is in the relation, then so is $\lambda x.t \mapsto t'$". The following property can be "proved" using the variable convention.

Faulty Lemma: Suppose $t \mapsto t'$. If $y \notin FV(t)$ then $y \notin FV(t')$.

"Proof" By induction on the inductive relation. The Var- and App-cases boil down to straightforward implications.

In the Lam-case we have the induction hypothesis if $y \notin FV(t)$ then $y \notin FV(t')$ and the assumption $y \notin FV(\lambda x.t)$. The goal is to show $y \notin FV(t')$. We use the variable convention to infer that $y \neq x$ where x is the bound variable from $\lambda x.t$ and y is the free variable from the lemma. Using this fact, we know that $y \notin FV(\lambda x.t)$ holds if and only if $y \notin FV(t)$ holds. Hence we can use the induction hypothesis to conclude also this case.

Counter Example: We can easily verify that $\lambda x.x \mapsto x$ is in the relation and $x \notin FV(\lambda x.x)$ holds. But $x \notin FV(x)$ is clearly false.

Nominal Isabelle, a package for the theorem prover Isabelle, has enough infrastructure so that one can use the variable convention in *formal* proofs by structural and rule induction, but in the latter case only for relations that are *variable convention compatible.*

* This work arose from joint work with Michael Norrish, Stefan Berghofer and Randy Pollack.

Reasoning about Weak Memory Models

Nathan Chong and Samin Ishtiaq

ARM Ltd, Cambridge, UK

The ARMv7 architecture defines a weakly consistent memory system. We have formalized the ARM memory model in Coq, inspired by Mike Gordon's 1993 unpublished work on formalizing the Alpha memory model in HOL.

In a weakly consistent memory model, the meaning of a program depends upon the order in which accesses appear to be executed by an observer such as a processor. Programmers can use barrier instructions to force this observability order. The semantics of a data barrier is to define two groups of accesses, A and B, on the executing processor Pe, and to require that all observers see all A accesses before any B access. Group A accesses are defined to include, all memory accesses that are observed by Pe before the data barrier. Group B accesses are defined to include all memory accesses by Pe that occur in program order after the data barrier.

A program is permissible if there exists an observability order in which: (1) The order is a strict partial one; (2) The order complies with uniprocessor rules; (3) Writes are serialized (4) The order complies with barrier semantics; (5) Every read in a program is bound to the last write observed to the same location; (6) A symbolic simulation of the program leaves memory in the same state as given by the final reads.

Consider the following program, in which the shared variables x and y have initial value 0. In the absence of barriers, this program is permissible, which means that processors p2 and p3 have observed the stores from p1 in different orders.

p1	p2	p3
a1: W x 1	b1: R y 1	c1: R x 1
a2: W y 1	b3: R x 0	c3: R y 0

The graph on the left gives a candidate ordering that illustrates this result. Now, we can force a final p3:y=1 by inserting data barriers between b1 and b3, and c1 and c3. This guarantees that p2 and p3 observe p1's write to y before their respective reads. That is, the program as given above is not permissible. This is illustrated in the graph on the right, where the extra edges added by the data barriers b2 (dotted arrows) and c2 (dashed arrows) create an observability cycle between c3 and b3, and so fails the acyclic condition of permissibility.

The ARM Instruction Set Architecture in HOL

Anthony Fox

University of Cambridge

The ARM instruction set architecture (version ARMv4T) has been formally specified using the HOL4 system. The specification derives from, and has been extensively validated by, a formal verification of the ARM6 micro-architecture, which implements ARMv3. The model has a number of uses: it can be used in the formal verification of other ARM micro-architectures (processor designs); one can reason about and formally verify ARM machine code; and it provides an excellent target for *verified* and *verifying* compilers. The high-fidelity ARM model is feature complete and not simplified. The following are specified:

- all 32-bit ARMv4 instructions;
- all 16-bit Thumb instructions;
- all coprocessor instructions and the coprocessor interface;
- all processor modes (and associated registers); and
- all interrupts and exceptions.

ARM systems are modelled in HOL as modular circuits, which consist of ISA, memory and coprocessor components. Functional specification is used to model the ISA itself, which is treated as a finite state machine with input and output. The approach avoids over-specification of the memory and, at the same time, enables efficient (functional) evaluation.

A series of supporting tools have been developed. A parser and pretty printer relate ARM assembler syntax, machine code and HOL terms. Evaluation in HOL for ground terms (i.e. with no free variables in the state space) is supported, enabling ARM programs to be loaded and run in a logically sound manner. The EmitML tool has been used to generate a Standard ML version of the specification. The SML code is compiled using MLton and this provides fast evaluation (thousands of instructions per second) when running the specification on sizeable programs.

A Hoare style logic for reasoning about machine code has been developed by Magnus Myreen. This has been applied to the ARM model and used to verify non-trivial assembler sub-routines. For example, one rule for addition is:

$$\{R\;15\;p, R\;a\;_, R\;b\;x, R\;c\;y\}$$
$$p : \texttt{ADD a b c}$$
$$\{R\;15\;(p+4), R\;a\;(x+y), R\;b\;x, R\;c\;y\}$$

Here, a, b and c are distinct 32-bit registers that are not the program counter (register fifteen). Support for reasoning about machine arithmetic is provided by HOL's n-bit word library. Further tool support has been added to this library with the recent addition of simplification routines and decision procedures.

A Continuous Relaxation for Proving Discrete ACL2 Theorems over Real Closed Fields

Paul B. Jackson and Grant Olney Passmore

LFCS, School of Informatics, University of Edinburgh

The work described involves the use of a decision procedure for real closed fields (RCF) to prove theorems originally formulated with rational-valued variables in the ACL2 theorem prover. From a theoretical perspective, we have isolated an interesting nonlinear fragment of the rational number field that admits a continuous relaxation. From a practical perspective, we have used some recent additions to ACL2 aimed at interfacing with external tools [2] to implement our procedure as a "trusted clause-processor" relying upon the QEPCAD-B [1] RCF procedure as a trusted oracle.

Our theoretical result is a decidability proof for a fragment of the true first-order theory of the rational number field. This result allows one to soundly relax variables in sentences in this fragment to range over real numbers instead of rational numbers. Specifically, we proved that any universal conjunctive sentence in the language of ordered rings restricted to relation symbols drawn from $\{=, \leq, \geq\}$ holds over the rationals iff it holds over the reals. We showed this is true even when both structures are extended by a collection of continuous real-valued maps whose images over rational vectors are always rational-valued. Moreover, we showed that decidability is preserved under such extensions as long as the new maps are definable in the full language of ordered rings.

ACL2 is an especially nice setting for such a procedure, as proof obligations in ACL2 are nearly always quantifier-free. Our practical result is the implementation of this proof procedure as an ACL2 clause-processor interfacing with QEPCAD-B with the following extensions: First, goals with function and relation symbols not in the language of ordered rings are generalized using techniques often used in the SMT community such as Ackermannization. Second, we incorporate the simple "trivial relaxation" that says any purely universal sentence in the language of ordered rings true over the reals is also true over the rationals.

References

1. Brown, Chris, *QEPCAD-B: a program for computing with semi-algebraic sets using CADs*. ACM SIGSAM Bulletin 37(4), pp 97-108, 2003.
2. Kaufmann, Matt and Moore, J Strother and Ray, Sandip and Reeber, Erik, *Integrating External Deduction Tools with ACL2*. IWIL, 2006.

A Sound Semantics for OCaml$_{light}$

Scott Owens
University of Cambridge

OCaml$_{light}$ [1, 2] is a formal semantics for a substantial subset of the Objective Caml core language. It is written in Ott [3], which generates LaTeX, HOL, Coq, and Isabelle/HOL definitions, and it comprises a small-step operational semantics and a syntactic, non-algorithmic type system. A type soundness theorem has been proved and mechanized in HOL-4. To ensure that the operational semantics accurately models Objective Caml, an executable version of the semantics has been created (and proved equivalent in HOL to the original, relational version) and tested on a number of small test cases.

We take a straightforward approach, formalizing the syntax of OCaml$_{light}$ as a collection of inductive datatypes, and the OCaml$_{light}$ type system and operational semantics as a collection of inductive relations and recursive functions directly over the source syntax. We hypothesize that this clear connection between the source language and its semantics will facilitate verification about OCaml$_{light}$ programs, in addition to the present verification of type soundness. OCaml$_{light}$ covers a large part of Objective Caml, but excludes the language's module and object systems. It includes:

- definitions
  - variant data types (e.g., `type t = I of int | C of char`),
  - record types (e.g., `type t = {f : int; g : bool}`),
  - parametric type constructors (e.g., `type 'a t = C of 'a`),
  - type abbreviations (e.g., `type 'a t = 'a * int`),
  - mutually recursive combinations of the above (excepting abbreviations),
  - exceptions, and values;
- expressions for type annotations, sequencing, and primitive values (functions, lists, tuples, and records);
- `with` (record update), `if`, `while`, `for`, `assert`, `try`, and `raise` expressions;
- let-based polymorphism with an SML-style value restriction;
- mutually-recursive function definitions via `let rec`;
- pattern matching, with nested patterns, `as` patterns, and "or" (`|`) patterns;
- mutable references with `ref`, `!`, and `:=`;
- polymorphic equality (the Objective Caml `=` operator);
- 31-bit word semantics for `int`s (using an existing HOL library); and
- IEEE-754 semantics for `float`s (using an existing HOL library).

Most of the proof effort is a straightforward application of HOL's tactic-based proof mechanisms. The entire proof is around 11000 lines of tactic scripts. The poster will present the OCaml$_{light}$ semantics and also lessons from the verification process.

1. S. Owens. A sound semantics for OCaml$_{light}$. In *Proc. ESOP (2008).*
2. S. Owens. OCaml$_{light}$. `http://www.cl.cam.ac.uk/~so294/ocaml`.
3. P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. In *Proc. ICFP (2007).*

A Meta Model for the Formal Verification of Networks on a Chip

Julien Schmaltz

Raboud University Nijmegen
Institute for Computing and Information Sciences
P.O. Box 9010 6500 GL, Nijmegen, The Netherlands
julien@cs.ru.nl

Systems on a chip (SoCs) are designed through a *platform* based approach: a new SoC is built according to a generic architecture, using pre-designed parameterized modules and processor cores. To meet high performance, networks on a chip (NoCs) constitute a promising communication architecture [1]. Moreover, the current trend in the SoC design community is to raise the level of abstraction and rely on verified parameterized library modules. This requirement will soon extend to communication network kernels, yet a formal theory for this category of functional modules is non existing today.

Our long term objective is to support the validation of abstract specifications for NoCs, and the verification of their correct implementation by a parameterized design. We propose a generic network model that encompasses protocols *and* topologies, routing algorithms and scheduling policies, and applies to a wide variety of architectures. We have expressed our model in the logic of an interactive theorem prover to provide mechanized reasoning support.

The heart of the model is function *GeNoC* [2], which formalizes the interactions between the three key constituents: interfaces, routing and scheduling. Our model makes no assumption on the protocol, the topology, the routing algorithm, or the scheduling policy. To abstract from any particular architecture, we have identified essential properties (considered proof obligations or simply constraints) for each constituent. Those imply the overall correctness of *GeNoC*. Hence, *GeNoC* and its global correctness are a *meta*-model and a *meta*-theorem of all architectures satisfying instances of the proof obligations. The validation of any particular architecture is reduced to the proof that each one of its constituents satisfies the generic constraints. The implementation of our model in an automated proof assistant provides a tool to specify and to validate high-level NoCs specifications. For any concrete architecture, the corresponding instances of the proof obligations are automatically generated. Our approach has been demonstrated on realistic academic designs, as well as industrial designs.

References

1. L. Benini and G. De Micheli. Networks on Chips: A New SoC Paradigm. *Computer*, 35(1):70–78, 2002.
2. J. Schmaltz and D. Borrione. A functional formalization of on chip communications. *Formal Aspects of Computing*, 2007. DOI: 10.1007/s00165-007-0049-0.

Exploring Algebraic Property Dependencies for Metarouting

Balraj Singh (Balraj.Singh@cl.cam.ac.uk),
Alexander J. T. Gurney (Alexander.Gurney@cl.cam.ac.uk),
Timothy G. Griffin (Timothy.Griffin@cl.cam.ac.uk)

Computer Laboratory, University of Cambridge

Internet connectivity has become an essential part of our infrastructure. This connectivity is implemented with dynamic routing protocols. A large hurdle in designing or extending protocols has been the difficulty of combining complex routing policy languages with correctness proofs.

Metarouting [1] is an approach to routing protocol design and implementation based on using a metalanguage to specify a routing language. The metalanguage is designed so that properties important for specific algorithms (such as monotonicity) can be *automatically* inferred, thus liberating the protocol designer from this burden. The fundamental objects in the metalanguage include semigroups and semirings, with potential properties including selectivity, commutativity, distributivity, and so on. These objects can be combined using constructors such as products, lexicographic products and disjoint unions to produce complex structures.

Understanding the implicational theory of the properties inferred is very useful for testing and debugging, meta-language design, and implementing property inference rules. This poster describes our use of automated theorem proving to generate a database containing fragments of the implicational theory of metarouting.

We are using a brute-force approach which enumerates subsets of properties and applies resolution theorem provers to test each set's consistency. Currently we are using Prover9 and the E Equational theorem prover on a cluster of 28 CPUs. After running for about 1 month, we have created a database of over 2000 implicational fragments. This database is being used for automated testing and debugging.

References

1. Griffin, T.G., Sobrinho, J.L.: Metarouting. In: *Proc. ACM SIGCOMM*, August 2005.

A Deep Embedding of a Decidable Fragment of Separation Logic in HOL

Thomas Tuerk

University of Cambridge Computer Laboratory
William Gates Building, JJ Thomson Avenue, Cambridge CB3 0FD, United Kingdom
http://www.cl.cam.ac.uk

Separation logic is an extension of Hoare logic that allows local reasoning about mutable data structures. It's been introduced by O'Hearn, Reynolds and Yang in 2001 [1]. The main idea of separation logic is the usage of a spatial conjunction $p * q$ that asserts that the formulas p and q hold on separate parts of the state. This notion of separation allows local reasoning and an elegant solution to aliasing problems. Moreover, it enables separation logic to be extended to concurrent programs in a natural way [2].

Separation logic has become more and more popular during the last few years. Besides ongoing research, there are several implementations. Smallfoot [3] is one of the oldest and best documented. It is a tool that allows to automatically verify specifications of programs written in a simple, imperative language. It uses light-weight specifications about dynamically allocated pointer structures on a heap that are expressed in a simple fragment of separation logic. Compared to fragments of separation logic used by other tools or current papers, the fragment used by Smallfoot is quite weak. However, it's simplicity leads to decidable verification conditions and therefore allows Smallfoot to be completely automatic.

In this work, I formalised the verification conditions generated by Smallfoot inside the interactive theorem prover HOL. Some documentation and the HOL code can be found in the HOL repository[1]. There are correctness proofs for the inferences used by Smallfoot and an implementation of a decision procedure for the verification conditions inside HOL. Thus, this work is likely to increase the trust in Smallfoot. Moreover, the decision procedure and the implementation of the inference rules can be used as an interactive separation logic calculator.

In the future, I hope to extend this formalisation to a general framework for separation logic in HOL.

References

1. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. *Lecture Notes in Computer Science*, vol. 2142, 2001.
2. Stephen Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.
3. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[1] http://hol.sourceforge.net, subdirectory examples/decidable_separationLogic