Research Proposal for the NWO *Vrije Competitie* EW

Formalizing the C99 standard in HOL, Isabelle and Coq

Institute for Computing and Information Sciences
Radboud University Nijmegen

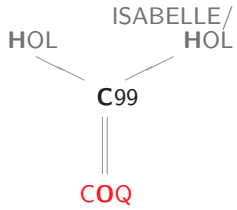
1

1a Project Title

Formalizing the C99 standard in HOL, Isabelle and Coq

1b Project Acronym

CH_2O



(The acronym is explained by the picture on the right. A solution of this substance in water is called *formalin*, which is another abbreviation of the project title. Therefore the project might also be called *the formalin project*, or in Dutch: *de C standaard op sterk water*.)

1c Principal Investigator

Freek Wiedijk

1d Renewed Application

No

2

2a Scientific Summary

The C programming language is one of the most popular in the world. It is a primary choice of language from the smallest microcontroller with only a few hundred bytes of RAM to the largest supercomputer that runs at petaflops speeds. The current official description of the C language – the C99 standard,

issued by ANSI and ISO together – is written in English and does not use a mathematically precise formalism. This makes it inherently incomplete and ambiguous.

Our project is to create a mathematically precise version of the C99 standard. We will *formalize* the standard using proof assistants (interactive theorem provers). The formalizations that we will create will closely follow the existing C99 standard text. Specifically we will also describe the C preprocessor and the C standard library, and will address features that in a formal treatment are often left out: unspecified and undefined behavior due to unknown evaluation order, casts between pointers and integers, floating point arithmetic and non-local control flow (goto statements, setjmp/longjmp functions, signal handling).

The results of the project will be as accessible as possible. We will develop the C99 formalization in three matching versions, for the proof assistants HOL, Isabelle and Coq. All will be published as open source, under a BSD-style license.

The project will produce a mathematically precise version of the C standard, and its results will be a main contribution to the technology for formal verification of C programs.

2b Abstract for Laymen (in Dutch)

Software (in de computer, maar ook in andere apparaten) gedraagt zich over het algemeen redelijk. Er *crasht* wel eens iets of er blijft eens iets hangen – met soms vervelende gevolgen – of de computer raakt geïnfecteerd met virussen en andere *malware* – óók vervelend – maar over het algemeen blijft het gedrag acceptabel. Al hebben producenten van software beperkt tijd en geld, ze slagen er toch in een redelijk evenwicht tussen *features*, snelheid en betrouwbaarheid te bereiken.

Eén van de redenen hiervoor is dat een computer conceptueel is verdeeld in *lagen*. Er bestaat bijvoorbeeld een afspraak tussen de maker en de gebruikers van de processor van een computer over hoe deze zich dient te gedragen. De maker hoeft zich daardoor niet te bekommeren om alle details van het gebruik, en kan zich volledig op snelheid en betrouwbaarheid concentreren. Net zo zijn er afspraken over gedrag van programmeertalen en besturingssystemen. Deze afspraken staan opgeschreven in *standaards*. De huidige standaard voor de programmeertaal C heet de *C99* standaard.

Het zou voor iedereen plezierig zijn als software betrouwbaarder was, maar er zijn omstandigheden waarbij een crash of ander verkeerd gedrag *echt* onacceptabel is. Voorbeelden zijn medische software en software in de lucht- of ruimtevaartindustrie. Ook is betrouwbaarheid erg belangrijk wanneer software in enorme aantallen in hardware wordt gebrand.

Een belangrijke bron van softwareproblemen (zowel van crashes als van virusinfecties) is het gebruik van de relatief primitieve programmeertaal C. Deze stamt uit de jaren zeventig, staat dicht bij de machine – wat om redenen van efficiëntie en mogelijkheden aantrekkelijk is – en is één van de meest gebruikte programmeertalen ter wereld.

Er zijn twee manieren om de huidige situatie te verbeteren. Men kan overgaan op minder primitieve, meer *wiskundige* programmeertalen. Of men kan om rede-

nen van efficiëntie en flexibiliteit talen als C blijven gebruiken, maar daarbij wiskundig *verifiëren* – bewijzen – dat er geen problemen kunnen optreden. Hiervoor worden verschillende soorten programma's gebruikt, waaronder de zogenaamde *bewijsassistenten*. Drie van de belangrijkste bewijsassistenten zijn HOL (Brits), Isabelle (Brits/Duits) en Coq (Frans). Momenteel worden voor practische programmaverificatie overigens vooral andere, meer geautomatiseerde, programma's gebruikt. Deze zijn evenwel minder flexibel dan bewijsassistenten en kunnen alleen een benadering van volledige verificatie geven.

Het project bestaat uit het verwerkbaar maken van de C standaard voor bewijsassistenten. Dit heet *formaliseren*. Tot nog toe bestaan er alleen formalisaties van C waarin te lastige constructies zijn weggelaten. Wij zullen de *hele* C standaard formaliseren. Op korte termijn levert dit een precisering van de C standaard op. Op langere termijn zal dit een essentieel ingrediënt zijn voor serieuze verificatie van C programma's met bewijsassistenten.

Om praktisch bruikbaar te zijn zal ook met bewijsassistenten het grootste deel van het redeneren over C programma's volautomatisch moeten gebeuren. Maar een geformaliseerde C standaard is daarbij een absolute vereiste voor een maximale graad van precisie en betrouwbaarheid.

2c Keywords

- C programming language
- programming language standardization
- programming language semantics
- proof assistants, interactive theorem proving
- program verification
- formal methods

3 Classification

NOAG-ict 2005-2010 themes:

- 3.6. *Intelligente systemen*
- 3.7. *Methoden voor ontwerpen en bouwen*

4 Composition of the Research Team

- dr. Freek Wiedijk, RU, principal investigator
- prof. dr. Herman Geuvers, RU, intended promotor
- *PhD student* (to be appointed on this project), RU
- dr. Jean-Christophe Filliâtre, CNRS
- dipl.-inf. Andreas Lochbihler, KIT
- dr. James McKinna, RU
- dr. Michael Norrish, NICTA
- dr. Erik Poll, RU

RU = Institute for Computing and Information Sciences, Radboud University Nijmegen
CNRS = Centre National de la Recherche Scientifique (and also INRIA Saclay – Île-de-France, and LRI, Université Paris Sud 11)
KIT = Fakultät für Informatik, Karlsruhe Institute of Technology
NICTA = Canberra Research Laboratory, National ICT Australia

Currently at RU we have a very good student, Robbert Krebbers, who will be getting his masters degree soon and who has a strong interest in this project.

5 Research School

IPA, Institute for Programming research and Algorithmics

6

6a Description of the Proposed Research

Scientific Problem and Research Goals. The three currently most popular programming languages are Java [23], C [20,28] and C++ [43] (in that order, both in the LangPop [31] and TIOBE [44] indexes). The first and third are strongly influenced by the second, which means that C is currently the most influential programming language ever. Also C runs on almost every computer in existence. Many more types of systems are regularly programmed in C than in Java or C++. In that sense it is the most widely used programming language of the three. Finally C is the native language of most modern operating systems, due to its tight connection to Unix.

But C programs can also be very dangerous! It is very easy for them to have bugs that make the program crash or behave badly in other ways. Null pointers can be dereferenced, arrays can be accessed outside their bounds, memory can be used after it is freed, or conversely can be forgotten to be freed after it is no longer needed leading to memory leaks. Furthermore C programs can be developed with too specific an interpretation of the language in mind, giving problems later.

There are various efforts to use *formal methods* to try to mitigate these problems. One tries to establish properties of the program using various mathematical methods, to make sure there will be no problems. These methods range from approaches like *static analyzers* and *model checking* to approaches where the program is annotated and *verification conditions* are generated and then proved, either automatically or interactively. These methods have recently been quite successful, with for example various projects proving the C source code of microkernel operating systems correct [13,29].

The most precise of these formal methods is the use of proof assistants, also called interactive theorem provers. Currently these are only used to prove conditions generated by other tools, and do not have the ability to keep track

of the full verification themselves. The research goal of this project is to take a decisive step towards a situation where proof assistants have enough *knowledge* to do the whole verification of a program from start to finish. This does *not* mean that all proofs will have to be done interactively by the user of the proof assistant. They could be fully automatic. However, the goal is to have the proof assistant *keep track* of everything. The main advantage of using proof assistants *all the way* over the current approaches is that the verifications will be fully transparent and coherent.

The current C standard [20] is written in English, and as such is inherently ambiguous (although of course the authors have tried hard to minimize that ambiguity) and not in a shape that is usable in a proof assistant. The goal of the project is to create a version of the C standard that *is* usable in a proof assistant. This is called *formalizing*.

Note that our research goal is not to formalize a *proof* of anything. There will be formalized proofs, to try things out. But the goal is to formally *define* something, that is, the C99 language. The formalizations that will be the outcome of the project will consist of a very carefully crafted chain of *definitions*.

Research Approach. For the detailed research plan, see Section 7a below.

Our research approach is straightforward. We will investigate current formalizations of significant fragments of the C language, and will also investigate the suitability of various tools for coding programming language semantics (especially the Ott tool [42]). Then we will select the most appropriate tool set, and start translating the C99 standard text [20] into a formal form. We will start with the aspects that are most difficult, and scientifically most interesting. These will be aspects of the C language generally left out by the current C formalizations. Probably we will focus first on non-local control flow: `goto` statements, `setjmp`/`longjmp` functions and signal handling.

The core of the formalized C99 standard will be a structured operational semantics of C, a *small step* semantics. The basis of this semantics will be the description of a *state*, which both contains a description of a *memory* (in the C way of looking at it) and an outside world with at least *files* and *time*.

In parallel we will define a framework for capturing the notion of 'a C semantics'. C has various dialects: Kernighan & Richie's version of C [28], the old C89 standard [1], the current C99 standard [20], and so on. Furthermore in a way every C compiler defines its own C dialect. We will define a *space* in which all these C dialects live, to make it as easy as possible to understand exactly what our formalized C99 standard amounts to.

We will develop the formalization for HOL [22], Isabelle [37] and Coq [7,15] simultaneously. In practice this will mean that we develop the formalization in some 'master' format (maybe using one of the three proof assistants, maybe using the Ott system [42], and maybe using a hybrid of one of the systems and Ott), and then generate the other formalizations from that. This will mean that we will not be able to use features that are specific for one of the proof assistants.

In practice this means that we will have to work on the level of the HOL system, as the other two systems have richer logics than HOL.

Scientific Significance and Urgence of the Proposed Research. There currently is a large body of successful work on formalization of programming languages cleaner than C – like the pioneering work on the semantics of SML [24,46], but especially several large projects on Java semantics [6,27,30] – while thus far there has been much less work on C. C is one of the most important programming languages ever and is being heavily used, especially for security- and safety-critical embedded software. Full formalization of the C language therefore will be a milestone for the theorem proving community, as well as for computer science as a whole.

C is smaller than most other programming languages – which makes formalization feasible given the time and resources of the project – but also 'dirtier' – which makes it very challenging and scientifically interesting. Many 'clean' theoretical approaches will break down when trying to handle aspects of C like non-local control flow and 'unsafe' casts between pointers and integers. Our research team has an especially strong expertise about the issues involved, and is in a unique position to address these problems. Furthermore, the research team has been especially selected with the aim not to restrict ourselves to a single proof assistant, in order for our results to be maximally usable.

The scientific significance of the project is that this will *not* be about an academic fragment of the C language, it will be about the *real thing*. It will show which semantic methods work in a realistic setting, instead of just for an academic simplification. Initiatives like the POPLmark challenge [3] have been very important and influential, but only concerned themselves with 'toy' languages. To find semantic methods that work for all of C will be highly non-trivial and scientifically very challenging.

The outcome of this research therefore will not only be of practical significance, but will also be important from a theoretical point of view, as it will show which of the theoretical methods *scale* to a real life language.

Comparison with Other Research. Existing C verification tools and techniques (i) only use a formalization that leaves out some of the hairier features of the C language *and/or* (ii) do not make the C semantics explicit and precise (formal) at all *and/or* (iii) are restricted in the properties that can be expressed and verified. The aim of our project is to lay a foundation for a C verification technology that has none of these restrictions.

There are already various fragments of C with a formalized semantics, (and even more formalizations of the semantics of Java, a language that is close to C). The most important formalizations of fragments of C are:

- The Cholera formalization from Michael Norrish's PhD thesis [38], written in HOL. Michael Norrish is part of our project team, and the starting point of the project will probably be this formalization.

- The Clight formalization from Xavier Leroy's Compcert project [11,33], written in Coq.
- The C0 formalization by Dirk Leinenbach [32], developed in the context of the Verisoft project and written in Isabelle.

Other significant formalizations of fragments of C are Paul Black's HOL formalization [10], Jeffrey Cook and Sakthi Subramanian's Nqthm formalization [14] and Harvey Tuch's Isabelle formalization [45]. There are also formal C semantics that are not expressed using a proof assistant, like Nikolaos Papaspyrou's denotational semantics [40].

Systems for actual C verification are currently not built on top of a formal C semantics. Instead they use tools that generate proof obligations which are then handled separately. Most of these tools do not involve proof assistants. The four most significant projects that *do* allow a user to prove proof obligations (verification conditions) interactively using a proof assistant are the following.

- The Frama-C system has a verification plug-in called Jessie written by Jean-Christophe Filliâtre and Claude Marché [35]. It is a successor to their Caduceus tool [18]. Both run on top of Jean-Christophe Filliâtre's Why tool [17]. These tools support multiple proof assistants and other back-ends, but originate in the Coq community.
- Part of the framework at NICTA used in the L4.verified project [29] is a tool by Michael Norrish for translating an annotated C program into the SIMPL programming language in the Isabelle system.
- The HOL-Boogie system translates a C program into a program in the BoogiePL programming language in the Isabelle system [12].
- The Key-C system allows C verification with the KeY system (which is usually used for Java verification) [36].

Other important systems for C verification that use program annotations which then are verified (although not interactively) are:

- The VCC system [13] (which *can* generate statements for a proof assistant, but in practice is used with SMT solvers like Z3).
- The HAVOC system [5].
- The VeriFast system [26].

Of course the currently most practical systems for C verification are static analyzers – like the one that is part of the Frama-C framework [2] – and systems based on model checking like SLAM and the Static Driver Verifier [4], BLAST [9] and Zing [41].

Comparison with Existing Research in the Research Group. The project is a collaboration in the ICIS research institute of the Radboud University Nijmegen between the Foundations Group and the Java Verification group. These groups have extensive experience with a large range of proof assistants and verification of imperative programs from the C/Java family.

On the research team we also have Michael Norrish as a HOL specialist, Andreas Lochbihler as an Isabelle specialist and Jean-Christophe Filliâtre as a Coq specialist These are all world leaders in verification of programs from the C/Java family.

Some Questions and Answers. Here are some potential questions about the project, with our answers. The first two questions complement each other:

Hasn't this be done already? Are there any scientifically interesting questions left? Existing formalizations all leave out one or more of the more subtle parts of the standard, like unknown evaluation order, casts between pointers and integers, floating point arithmetic and non-local control flow. Also no existing formalization deals with either the C syntax (including the C preprocessor) or the C standard library. Dealing with all the details of the standard, and especially with the 'feature interaction' between them will certainly be non-trivial and scientifically highly challenging.

The C standard is a document of over 500 pages, written by a committee of almost 200 members. Isn't that much too large to be completely formalized by a single PhD student? We will not *prove* much, we will write down series of definitions. When using proof assistants writing definitions is relatively easy, while proving is what is labor intensive. We postpone the most labor intensive activity.

You want to formalize C. Why not formalize the standards for C++ or Java? In the Rationale for the C standard [21] one of the items that consists the *Spirit of C* is

Keep the language small and simple.

Indeed C is much smaller that the other two languages. Formalizing the full C++ or Java standards would be much too big for a single PhD. *(Note for the referees: a 'vrije competitie' project only pays for either a single four year PhD student, or for a two or three year postdoc.)*

You will write down a lot of formal definitions, but you will not prove *much about it. Will you be able to get the formalization correct that way?* You are right, it is extremely hard to get a formalization that is *fully* correct that way. However, we *will* prove properties of some sample programs in the project, as a sanity check. Also, once a first draft of the formalized standard is out, many people will start looking at it and proving properties of it, and then remaining 'bugs' in the formalization will be found fast. The fact that the formalization will be available in more than one proof assistant also helps in this respect.

Building a C verification environment on top of a formal language semantics is not efficient. Why not keep the C semantics out of this and rely on tools? Only by making the semantics explicit – open for inspection as a language semantics – can one get everything exactly right. We agree that having the

semantics 'implicit' in the behavior of the tools is much more efficient from a practical point of view. We see this as a short term versus long term issue. In the short term working with unformalized tools is more efficient. In the long term you *do* want to link these tools in some way to a formal version of the semantics. Our work will be essential at that point.
Also, in the case of very safety-critical software – e.g., according to the DO-178B standard for avionics software [19] – the tools used need to be certified as well. For that kind of certification a language semantics is essential.

A formal semantics for C is the wrong thing. Shouldn't you encourage people to move on to better, more modern programming languages? 'C together with formal proofs about it' really *should* be seen as a 'better, more modern programming language'. When using C in such a way, one is having one's cake and eating it too.

6b Application Perspective

A formal version of the full C standard will be an important artifact. When establishing a property of a C program, it will be very attractive to be able to claim that it has been proved with respect to the *full* official standard. This kind of 'knowledge' about the C semantics in the current state of the art is mostly implicit in various tools. These tools – when from outside academia – are often considered a competitive advantage of the companies that produced them, and are then *not* freely available. For instance this holds for the C verification tool by Michael Norrish as used in the L4.verified project [29].

A formalization of the C99 standard has three main applications:

- The C99 formalization will make the C99 standard utterly precise. This will be useful for compiler writers, who will get the means to establish how the standard needs to be understood without having to deal with the ambiguities of the English language. Programmers writing C programs get the same benefit.
- There already are various projects to prove C compilers correct, like the Compcert project of Xavier Leroy [11,33]. These projects need a semantics of a version of C. These currently are subsets of full C, with names like Clight or C0. With a formal version of the C99 semantics, the correctness of the compiler becomes provable with respect to the full *official* standard.
- Currently people proving C programs correct with proof assistants use tools like VCC [13] and Frama-C [35] which generate *verification conditions* from C source annotated in the style of Hoare logic. These tools implicitly 'know' about the semantics of C, but this knowledge is not explicit. A more thorough approach will be to have such a tool not just generate the verification conditions, but to also have it synthesize formal *proofs* about the properties of the program. This will be less efficient, computationally, but it will be fundamentally more reliable.

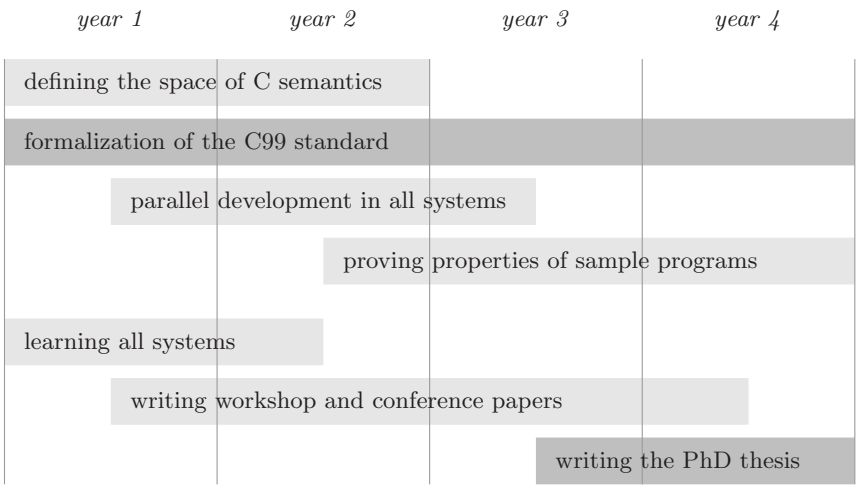


Fig. 1. Project Phases

A fully formalized C99 semantics will get a lot of international attention. In the days of Algol the Netherlands was at the core of the programming language standards community. This project might push the Netherlands back up in this community again, a goal that we will actively pursue by seeking contact with the C standard committee.

7 Project Planning

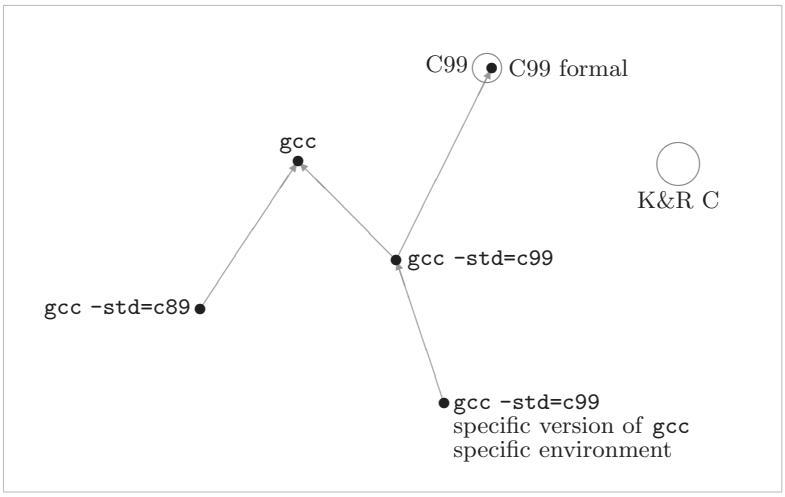
7a Project Phases

The phases of the project are shown in the chart of Fig. 1. They are:

Defining the space of C semantics. We will define a space of C semantics ('semanticses' is not correct English, but we mean the plural of semantics here) and model it in the formalizations as a type `C_semantics`. The C99 standard corresponds to a *point* in this space, which means that the formalization of that standard will be a formal definition of an element

```
C99 : C_semantics
```

Other points in this space will correspond to other variants of C, and to the behavior of specific C compilers on specific machines. We will define a relation 'conforms to' on this space, which makes it possible to express that a given compiler conforms to a given standard. This relation will make the space of C semantics into a lattice, as suggested by the diagram in Fig. 2.
The notion of behavior of a program will not just be on the level of the description of the C standard library (which mostly is about input/output to



The points correspond to *sets* of possible program behaviors, and the arrows correspond to set inclusion. The non-formal language descriptions are represented by circles instead of points, as it can be disputed whether they allow a behavior or not.

Fig. 2. The lattice of C semantics

streams and files). The behavior of programs running in a *freestanding environment* (see Section 5.1.2.1 of [20]) will be represented as well. Therefore the behavior of the program will be in terms of interaction with an *environment*, where this environment then is a parameter that can be *instantiated* with a world consisting of streams and files.

In *A Few Billion Lines of Code Later*, the people from Coverity report on their experience with C in the real world [8]. They claim that:

> *The C language does not exist* [...] *While a language may exist as an abstract idea, and even have a pile of paper (a standard) purporting to define it, a standard is not a compiler. What language do people write code in? The character strings accepted by their compiler.*

We want to take this observation seriously, and formalize the C99 in a way that does justice to actual C practice.

Formalization of the C99 standard. This is the core part of the project. We will formalize *all* of the C99 standard, starting with the more scientifically interesting and challenging aspects.

We will start by defining a structural operational (small step) semantics for both expressions and statements, which is general enough that it will be able to accommodate the more difficult aspects of the standard. After that we will proceed to 'flesh out' this semantics by translating the text of the C99 standard, section by section.

We will publish about how our semantics relates to existing formal semantics of fragments of C, comparing our approach to other approaches.

Parallel development in all systems. The C99 formalization will be developed for HOL [22], Isabelle [37] and Coq [7,15]. The HOL system will be HOL4 [39], but the HOL4 version will also be made to work with HOL Light [25].
If the Ott tool [42] turns out not be sufficient for the whole formalization, then software will have to be developed for converting a 'master' formalization into three parallel versions for the three proof assistants.

Proving properties of sample programs. The focus of the project is on *defining* and not on *proving*. Development of the language theory of C is beyond the scope of this project. However, as a sanity check on the formalization various properties of selected programs will be proved. This will establish that the definitions in the formalization behave reasonably.

7b Educational Aspects

The PhD student will take part in the standard educational program of the Faculty of Science of the Radboud University Nijmegen, and of the research school IPA. The student will also attend a summer school related to the topic of the project.

8 Expected Use of Instrumentation

No instrumentation will be needed beyond a standard personal computer.

9 Literature

Five key publications of the research team are:

- Luís Cruz-Filipe, Herman Geuvers and Freek Wiedijk, *C-CoRN, the Constructive Coq Repository at Nijmegen* [16].
- Jean-Christophe Filliâtre and Claude Marché, *The Why/Krakatoa/Caduceus Platform for Deductive Program Verification* [18].
- Bart Jacobs and Erik Poll, *Java Program Verification at Nijmegen: Developments and Perspective* [27].
- Conor McBride and James McKinna, *The View from the Left* [34].
- Michael Norrish, *C formalised in HOL* [38].

References

1. American National Standards Institute. *Programming language, C: ANSI X3.159-1989*. Number 160 in FIPS Publications. ANSI Technical Committee X3J11, 1989.
2. Frama-C Software Analyzers. Value analysis plug-in. `http://frama-c.com/value.html`.

3. Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 50–65, 2005.
4. Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. In *Formal Methods in Computer Aided Design, FMCAD 2010*, 2010.
5. Thomas Ball, Brian Hackett, Shuvendu K. Lahiri, Shaz Qadeer, and Julien Vanegue. Towards Scalable Modular Checking of User-Defined Properties. In G.T. Leavens, P. O'Hearn, and S.K. Rajamani, editors, *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010*, volume 6217 of *LNCS*, pages 1–24, 2010.
6. Gilles Barthe, Pierre Crégut, Benjamin Grégoire, Thomas Jensen, and David Pichardie. The MOBIUS Proof Carrying Code Infrastructure. In *Formal Methods for Components and Objects: 6th International Symposium, FMCO 2007*, pages 1–24, 2008.
7. Yves Bertot and Pierre Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
8. Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.
9. Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5–6), 2007.
10. Paul E. Black. *Axiomatic Semantics Verification of a Secure Web Server*. PhD thesis, Brigham Young University, Utah, USA, 1998.
11. Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.
12. Sascha Böhme, Michał Moskal, Wolfram Schulte, and Burkhart Wolff. HOL-Boogie — An interactive prover-backend for the Verifying C Compiler. *Journal of Automated Reasoning*, 44(1–2):111–144, 2010.
13. Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009*, volume 5674 of *LNCS*, pages 23–42, 2009.
14. Jeffrey Cook and Sakthi Subramanian. A Formal Semantics for C in Nqthm. Technical Report 517D, Trusted Information Systems, 1994.
15. Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2009.
16. Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN: the Constructive Coq Repository at Nijmegen. In A. Asperti, G. Bancerek, and A. Trybulec, editors, *Mathematical Knowledge Management, Proceedings of MKM 2004, Białowieza, Poland*, volume 3119 of *LNCS*, pages 88–103. Springer-Verlag, 2004.
17. Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, 2003.
18. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *LNCS*, pages 173–177, 2007.

19. Radio Technical Commission for Aeronautics (RTCA). DO-178B: Software Considerations in Airborne Systems and Equipment Certification, 1982.
20. International Organization for Standardization. *ISO/IEC 9899:1999: Programming languages – C*. ISO Working Group 14, 1999. Draft standard WG14/N1256, the combined C99 + TC1 + TC2 + TC3, dated September 7, 2007.
21. International Organization for Standardization. *Rationale for International Standard – Programming Languages – C*. INCITS J11 and SC22 WG14, 2003. Revision 5.10.
22. Mike Gordon and Tom Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.
23. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.
24. Robert Harper and Chris Stone. An Interpretation of Standard ML in Type Theory. Technical Report CMU–CS–97–147, Carnegie Mellon University, 1997.
25. John Harrison. *The HOL Light manual (1.1)*, 2000.
26. Bart Jacobs and Frank Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2008.
27. Bart Jacobs and Erik Poll. Java Program Verification at Nijmegen: Developments and Perspective. In K. Futatsugi, F. Mizoguchi, and N. Yonezaki, editors, *Software Security – Theories and Systems*, number 3233 in LNCS, pages 134–153. Springer, 2004.
28. Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.
29. Gerwin Klein et al. seL4: formal verification of an OS kernel. In J.N. Matthews and Th. Anderson, editors, *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.
30. Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *TOPLAS*, 28(4):619–695, 2006.
31. LangPop.com. Programming Language Popularity. `http://langpop.com/`.
32. Dirk Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, Saarbrücken, 2008.
33. Xavier Leroy. Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant. In *POPL'06*, 2006.
34. Conor McBride and James McKinna. The View from the Left. *Journal of Functional Programming*, 14(1), 2004.
35. Yannick Moy and Claude Marché. *Jessie Plugin Tutorial,* Beryllium *version*. INRIA, 2009.
36. Oleg Mrk, Daniel Larsson, and Reiner Hhnle. KeY-C: A tool for verification of C programs. In *Proceedings of 21st Conference on Automated Deduction (CADE-21*, 2007.
37. Tobias Nipkow, Larry Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
38. Michael Norrish. C formalised in HOL. Technical Report UCAM-CL-TR-453, University of Cambridge, Computer Laboratory, 1998. PhD thesis University of Cambridge.
39. Michael Norrish and Konrad Slind. *The HOL system, Description*, 2010. `http://hol.sourceforge.net/documentation.html`.
40. Nikolaos Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, 1998.

41. Tomáš Matousek and Filip Zavoral. Extracting Zing Models from C Source Code. In J. van Leeuwen, G.F. Italiano, W. van der Hoek, C. Meinel, H. Sack, and F. Plášil, editors, *SOFSEM 2007: Theory and Practice of Computer Science*, volume 4362 of *LNCS*, 2004.
42. Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):70–122, 2010.
43. Bjarne Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, 2000.
44. TIOBE Software. TIOBE Programming Community index. `http://www.tiobe.com/content/paperinfo/tpci/`.
45. Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, The University of New South Wales, 2008.
46. Myra VanInwegen and Elsa Gunter. HOL-ML. In *Proceedings of the 1993 International Workshop on the HOL theorem proving system and its applications*, volume 780 of *LNCS*, pages 59–72, 1994.

10 Requested Budget

PhD student		
appointment for 4 years	*standard amount*	€ 177.495
personal benchfee	*standard amount*	€ 5.000
Total		€ 182.495

(The amounts given here are indicative: the numbers that currently apply might be different.)