1a) **Project Title.** *Proof Checking the Proof Checker.*
1b) **Project Acronym.** PCPC.
1c) **Principal Investigator.** Freek Wiedijk.
1d) **Renewed Application.** No.

2a) **Summary.** It is considered a fact of life that all serious computer programs contain errors, so-called ‘bugs’. Empirical data indicates that production software has around two bugs per thousand lines of source code, and even programs used on space missions by NASA are believed to have around 0.1 bugs per thousand lines of code [26].

Interactive theorem proving is a technology for building programs that almost certainly have *zero* bugs per thousand lines of code. Already some significant programs have been shown to be fully correct. For instance both the certified C compiler of Xavier Leroy [25, 8] and the programs from the proof of the Four Color Theorem by Georges Gonthier [13] have been formally shown – with a fully computer-checked proof – to do *precisely* what they should do, and therefore are guaranteed to be bug-free.

This technology of interactive theorem proving for software correctness is on the verge of becoming widely applicable. A sign that this moment has not yet arrived is that currently it is not even used by the very people who build tools for it. Thus far, no system for interactive theorem proving has been formally proved bug-free.

The *Proof Checking the Proof Checker* project will change this situation. At the end of this project one of the best systems for interactive theorem proving will have used its own technology to establish that it is completely sound. Furthermore not just a model, but the actual source code of the program will have been proved correct.

2b) **Abstract for laymen (in Dutch).** Het wordt als onontkoombaar gezien dat ieder serieus computerprogramma fouten bevat, ‘bugs’ genaamd. In de praktijk bevat productiesoftware rond de twee bugs per duizend regels broncode, en zelfs van programma’s voor ruimtemissies door NASA wordt geschat dat ze rond de 0,1 bugs per duizend regels bevatten.

Interactief stellingenbewijzen is een technologie om programma’s te maken die met grote zekerheid *nul* bugs per duizend regels programmatekst hebben. Momenteel zijn al enkele bijzonder ingewikkelde programma’s op deze manier volledig correct bewezen. Zo zijn er de gevalideerde C compiler van Xavier Leroy en de programma’s voor het bewijs van de vierkleurenstelling door Georges Gonthier, waarvan formeel is aangetoond – en met een volledig door de computer gecontroleerd bewijs – dat ze *exact* doen wat ze moeten doen, dus dat ze gegarandeerd bugvrij zijn.

De technologie van het interactief stellingenbewijzen voor programmacorrectheid staat op het punt om algemeen toepasbaar te worden. Een teken dat dit punt nog niet is bereikt is dat deze technlogie momenteel niet eens wordt gebruikt door de mensen die de software hiervoor ontwikkelen. Tot nog toe is geen enkel systeem voor interactief stellingenbewijzen formeel bugvrij bewezen.

Het *Proof Checking the Proof Checker* project zal hier verandering in brengen. Aan het eind van het project zal van één van de beste systemen voor interactief stellingenbewijzen door middel van zijn eigen technologie betrouwbaar bewezen zijn. Bovendien zal niet slechts van een model, maar van de volledige broncode van het programma de correctheid zijn aangetoond.

2c) **Keywords.** Theorem proving, proof assistants, formal methods, program correctness, software reliability, HOL.

3) **Classification.** *Informatica*, NOAG-ict 2005-2010 theme 3.6: *Intelligente systemen* (Computationele logica, Redeneersystemen, Semantiek).

4) **Composition of the Research Team.**

name	*primary field*	*affiliation*
Prof. dr. H.P. Barendregt	mathematical logic	RU
Prof. dr. J.H. Geuvers (*promotor*)	theorem proving	RU
Dr. F. Wiedijk	theorem proving	RU
Dr. J.H. McKinna	functional programming	RU
Drs. R.S.S. O'Connor	theorem proving	RU
Drs. D.G. Synek	functional programming	RU
Dr. J.R. Harrison	theorem proving	Intel
PhD student		RU

RU = Institute for Computing and Information Sciences
Radboud University Nijmegen
Nijmegen, The Netherlands

Intel = Intel Corporation
Hillsboro, Oregon, USA

5) **Research School.** IPA, Institute for Programming research and Algorithmics.

6a) **Description of the Proposed Research**

– *Scientific Problem and Research Goals.*

Here is the motivation for the project summarized in four short sentences: *Compilers compile themselves. Proof assistants do not yet prove themselves correct. This shows that compiler technology is more mature than proof assistant technology. The goal of the project is to remove this distinction.*

Software is notorious for being unreliable. Everyone has experience with software misbehaving, and occasionally software errors have very costly consequences [31]. The fact that software errors are generally affectionally called 'bugs' does not make this less significant.
There are various approaches to improve this situation, of which one class is called *formal methods*. The formal methods consist of applying techniques

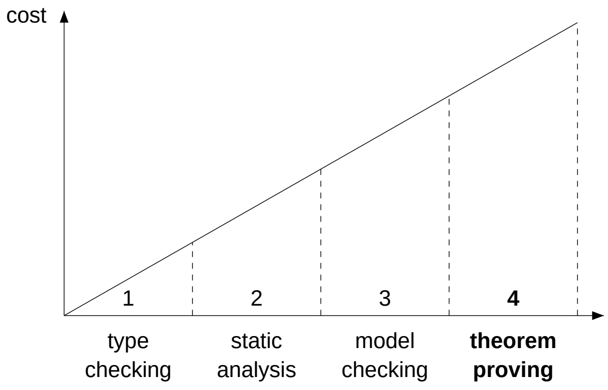

Fig. 1. The main formal methods.

from mathematics and specifically from mathematical logic to establish properties of computer programs. There are various formal methods that differ in the amount of information that they provide about software and in the amount of work that it costs to establish that information. Jean-Raymond Abrial has the suggestive diagram shown in Fig. 1 [2], which summarizes the main formal methods that are currently popular. The axes of this diagram are merely suggestive, but it should be clear that strongly typed programming is cheap but not does give much reliability, while theorem proving is very expensive but potentially gives the highest reliability possible. Because of the high cost of theorem proving, in practice it currently is mostly used for correctness of software for human transportation (with spacecraft an extreme case) and for medical applications.

There actually is a spectrum of theorem proving going from fully *automated* theorem proving (ATP) to *interactive* theorem proving. A system for interactive theorem proving is called a *proof assistant* or *proof checker*, and a proof developed inside such a system is called a *formalization*. The most important current proof assistants are (see also [41]):

HOL proof assistants [14]		
HOL4 [33]	Norrish & Slind	UK, USA & Australia
HOL Light [16]	Harrison	UK & USA
ProofPower [27]	Arthan	UK
Isabelle [32]	Nipkow, Paulson & Wenzel	UK & Germany
non-HOL proof assistants		
Coq [9]	Barras & Herbelin	France
B Method [1]	Abrial	France
PVS [37]	Owre, Rushby & Shankar	USA
ACL2 [23]	Moore	USA
Mizar [30]	Trybulec	Poland & Japan

Currently interactive theorem proving for software correctness is mostly used with mathematical models of software or for relatively simple programs. It is not yet applied much to the actual source code of existing software. In particular the proof assistants themselves are still considered to be out of reach of current interactive theorem proving technology. This is a noticeable difference with compilers. Proof assistants and compilers are very similar systems, but a compiler *is* used for itself. Compilers routinely compile their own source code, as shown in Fig˙ 2.
The analogous situation for a proof assistant is slightly more complex because there both a compiler *and* a proof assistant are involved. Both systems process the same source code, but the compiler produces an executable program while the proof assistant just produces a Boolean value stating that the code is correct. This situation is shown in Fig. 3.
Actual proof assistants, like compilers, have thousands of lines of source code, and are in practice too big for the kind of scrutiny that interactive theorem proving requires. However, here is where the *de Bruijn criterion* [3] comes in. Proof assistants often have an architecture where only a small part of the program needs to be trusted for the whole system to be mathematically sound. (I.e., if that part of the program is correct, it will be impossible

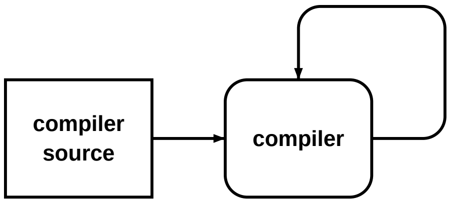

Fig. 2. A compiler compiling itself.

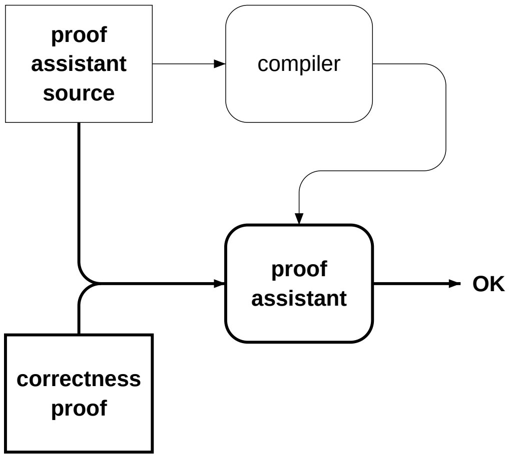

Fig. 3. A proof checker checking itself.

to prove *false*, even if the rest of the program has bugs.) This part of the program is called the *kernel* or *core* of the program, and it is *not* too big to be proved correct within a reasonable time. The de Bruijn criterion leads to a modified version of Fig. 3, shown in Fig. 4. This diagram presents what we will realize in the *Proof Checking the Proof Checker* project.

– *Relation to Existing Research.*
A self-correctness proof of a proof assistant has not been done yet, but significant work in that direction already exists. (What has not been done yet is prove correct the *actual production source code* of a proof assistant which is used on a large scale for serious work.) For example there are the verification of the Ivy proof checker in ACL2 by Olga Shumsky [28], the formalization of HOL in HOL from the nineties by Joakim von Wright [40], the formalization of nqthm (the precursor of ACL2) in Coq and vice versa by Gilles Dowek and Bob Boyer [11], and the work on formalizing type theory in Lego by Randy Pollack and James McKinna [38, 29]. However, there are two projects that stand out and surpass earlier efforts:

  • **Coq in Coq.** In the late nineties Bruno Barras, together with his advisor Benjamin Werner, formalized the theory behind the Coq proof assistant, and *extracted* from it a program that was functionally equivalent to the Coq kernel [4–7]. This program can be used to recheck many Coq formalizations and is guaranteed to be correct. However, in practice no one uses it, and the difference in size and sophistication between this program and the actual Coq kernel that everyone uses is huge. The extracted Coq in Coq program is a few hundreds lines of code, while the

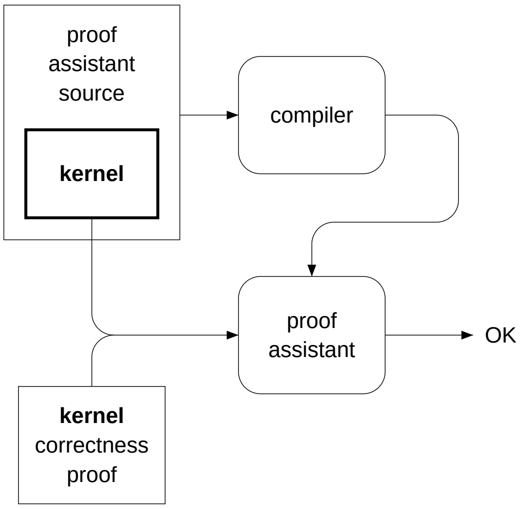

Fig. 4. The *Proof Checking the Proof Checker* project.

actual Coq kernel is more than ten thousand lines of code. Also the real Coq kernel uses programming language features that are not available when one extracts a program from a formalization.

The methodology of the Coq in Coq project is shown in Fig. 5. First a beautiful piece of mathematics is coded in the computer, which then is refined into an executable program. This is also the approach used with the B Method [1].

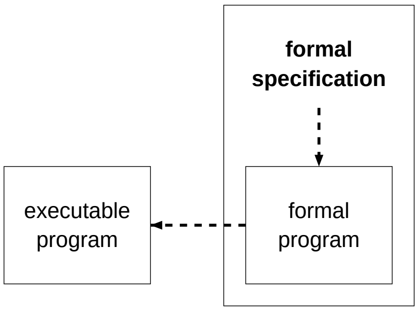

Fig. 5. Developing software as mathematics.

- **HOL in HOL.** In 2006 John Harrison proved the correctness of a simplified version of the kernel of his HOL Light proof assistant [18]. Here the distance between the actual code of the system and the version proved correct is much smaller. Mostly some features were removed to make the effort simpler, but the code that was proved correct was essentially code from the real system. For example, the function definition from the actual HOL Light kernel source code:

```
let vsubst =
  let rec vsubst ilist tm =
    match tm with
      Var(_,_) -> rev_assocd tm ilist tm
    | Const(_,_) -> tm
    | Comb(s,t) -> let s' = vsubst ilist s and t' = vsubst ilist t in
                   if s' == s & t' == t then tm else Comb(s',t')
    | Abs(v,s) -> let ilist' = filter (fun (t,x) -> x <> v) ilist in
                  if ilist' = [] then tm else
                  let s' = vsubst ilist' s in
                  if s' == s then tm else
                  if exists (fun (t,x) -> vfree_in v t & vfree_in x s) ilist'
                  then let v' = variant [s'] v in
                       Abs(v',vsubst ((v',v)::ilist') s)
                  else Abs(v,s') in
  fun theta ->
    if theta = [] then (fun tm -> tm) else
    if forall (fun (t,x) -> type_of t = snd(dest_var x)) theta
    then vsubst theta else failwith "vsubst: Bad substitution list"
```

is represented in the HOL in HOL formalization as:

```
let VSUBST = define
  `(VSUBST ilist (Var x ty) = REV_ASSOCD (Var x ty) ilist (Var x ty)) /\
   (VSUBST ilist (Equal ty) = Equal ty) /\
   (VSUBST ilist (Select ty) = Select ty) /\
   (VSUBST ilist (Comb s t) = Comb (VSUBST ilist s) (VSUBST ilist t)) /\
   (VSUBST ilist (Abs x ty t) =
        let ilist' = FILTER (\(s',s). ~(s = Var x ty)) ilist in
        let t' = VSUBST ilist' t in
        if EX (\(s',s). VFREE_IN (Var x ty) s' /\ VFREE_IN s t) ilist'
        then let z = VARIANT t' x ty in
             let ilist'' = CONS (Var z ty,Var x ty) ilist' in
             Abs z ty (VSUBST ilist'' t)
        else Abs x ty t')`;;
```

The remainder of the HOL Light kernel after the features were removed was more than half of the code. However, the feature removal meant that the version of the system that was proved correct could not actually check existing HOL proofs. Also, for the correctness proof the OCaml source code was translated into HOL definitions by hand. Although it is easy to see by comparing source files that this translation is correct, it still leaves a gap.

In HOL in HOL an opposite direction was followed from the Coq in Coq project. While in Coq in Coq the software was the *result* of the work, in HOL in HOL it was the starting point. HOL in HOL is not about creating software, but about taking existing software – real production code – and proving it correct. This is sketched in Fig. 6. From the point of view of software verification this seems to be the more interesting and challenging direction.

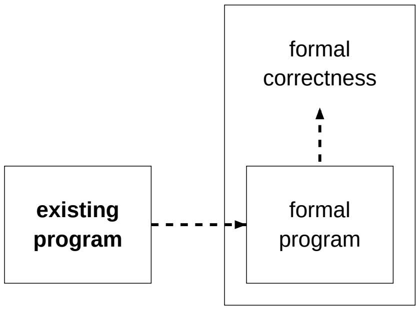

Fig. 6. Proving existing software correct.

Of course in practice the best approach is a middle ground between Figures 5 and 6. Still, starting from experimentation with a functional program and only when the program turns out to be reasonable begin work on a correctness proof seems a good approach. This was the methodology followed both by Georges Gonthier in his Four Color Theorem verification [13] and by Russell O'Connor in his work on provably correct exact real arithmetic [35].

– *Research Approach.*
The project that we propose here is to finish what John Harrison started with his HOL in HOL formalization. One PhD student should be able to reach the point where the *whole* of the HOL Light kernel is proved correct. The system then will be running a kernel that has been proved correct by itself.
In the HOL in HOL project the HOL code that was proved correct was syntactically closely related to the OCaml source code of the HOL Light kernel, but there was no *formal* relation between the two. For this reason we propose to add automatic translation from HOL code to OCaml (Step 1.1 in Section 7), even if it essentially will produce code that already is there. Conversely, OCaml code will be automatically converted to HOL definitions (Step 3). These translations will be formally proved correct with respect to a formal OCaml semantics (Step 2.2). This means that we will not just prove a *model* of the HOL Light kernel correct, but that the correctness proof will apply to the actual source code.
A more detailed description of the project will be given in Section 7 below.

There are a couple of possible objections to this project that need to be addressed. First, there is a *chicken and egg* problem here. The program that checks the correctness might be wrong, and for this reason accept a fallacious proof, and therefore in fact might not be correct. Of course this possibility exists. However, as a human will be paying attention as well, it is so small that it is only of philosophical importance [39].

More serious is Gödel's second incompleteness theorem [24, 12]. It says that a system cannot prove its own consistency, which applies to a proof assistant too. A proof assistant cannot support a proof that it will never prove *false*. However this is not a serious problem either. What will be proved is not that the system is consistent but that the system implements its logic correctly. Equivalently, the correctness proof will show that the consistency of the logic implies that the system will never accept a proof of *false*.
This consistency can be stated in only a few lines and is very simple, while the real world program being proved correct is hundreds of lines long and very complex.
In fact this approach was used in the HOL in HOL project. There the consistency of the HOL logic is phrased as an 'inaccessible cardinal axiom for the HOL logic':

```
new_type("I",0);;

let I_AXIOM = new_axiom
 `UNIV:ind->bool <_c UNIV:I->bool /\
  (!s:A->bool. s <_c UNIV:I->bool ==>
               {t | t SUBSET s} <_c UNIV:I->bool)`;;
```

This then is shown to imply that the HOL Light implementation will never accept a proof of *false*.
Finally there is the question: 'Why HOL Light?' There are impressive programs that have been validated in other systems. Maybe one of these other systems would be a better choice? The reason for our choice for HOL Light is that it has by far the smallest kernels of the major proof assistants that satisfy the de Bruijn criterion:

system	*kernel size* (*in* 10^3 *lines*)
HOL4	6
HOL Light	0.7
ProofPower	7
Isabelle	5
Coq	14

Note that although HOL Light is a relatively light-weight system, it is one of the best proof assistants available. It has been extensively used for hardware verification at Intel [15, 17], and is among the best systems for formalization of mathematics [42, 43].

– *Scientific Importance and Urgency of the Proposed Research.*
It is very important to use interactive theorem proving on real world programs, programs that are actually used. Only then the technology has to face real problems, and only then will it be developed in the right direction. Having a system verify itself (not just a model or a simplified version, but the actual code with all the dirty tricks to make it run fast) is a good choice for such a verification, and the technology is at a point that this can be done. Clearly now is the time to do this.

– *Relation to the Research Group.*
  The Foundations Group in Nijmegen is internationally renowned for its research into the technology and use of proof assistants, both for mathematics and computer science. There is strong expertise in mathematical logic, functional programming and the use of interactive theorem provers, which makes it the best place in the Netherlands for this project.

6b) **Application Perspective.** This project will prove a program correct that has been used for many years both in academia (with impressive results like [19]) as well as in industry [10, 17]. This is not about experimentation or a prototype: it is about 'the real thing'. The project proposed here will really mean pushing the boundary of interactive theorem proving for software correctness.
Part of the project – Phase 3 in Section 7 below – will be to investigate how program development can be integrated with validation of the developed code. The project will not just be about proving a single program correct, but also about creating a methodology that is generally applicable. This will then allow many functional programs to be proved correct, including programs that use state, exceptions and non-structural recursion.

7) **Project planning.** Fig. 7 presents an overview of the project schedule. It consists of three phases, with each phases divided in steps as follows:

Phase 1: *Create a fully self-verified HOL Light.*
1.1. *Compile the existing HOL in HOL code to OCaml.*
1.2. *Extend the HOL in HOL code to match the actual kernel.*
1.3. *Run HOL Light on top of the extended HOL in HOL code.*
1.4. *Extend the correctness proof to the extended HOL in HOL code.*
Phase 2: *Obtain further correctness evidence.*
2.1. *HOL in HOL outside OCaml.*
2.1a. *Run the verified kernel inside HOL Light by rewriting.*
2.1b. *Check the correctness proof in other HOL provers.*
2.2. *Prove the OCaml translation correct inside HOL Light.*
Phase 3: *Integrate OCaml and HOL development.*

We will now detail each of these steps:

Phase 1: *Create a fully self-verified HOL Light.*
1.1. *Compile the existing HOL in HOL code to OCaml.*
The HOL in HOL formalization represents the verified code as recursive HOL definitions. The first part of the project is to translate these 'executable' HOL definitions back into OCaml syntax. This way an executable HOL Light-like kernel for a simplified logic is obtained.
The HOL in HOL code will have to be modified and extended many times during the project. For this reason the translation into OCaml will be automated. This is done by writing a printer for HOL terms in OCaml syntax, that then is used to print the function definitions from the HOL in HOL kernel. The result of this step is what amounts to a version of *program extraction* for HOL Light.

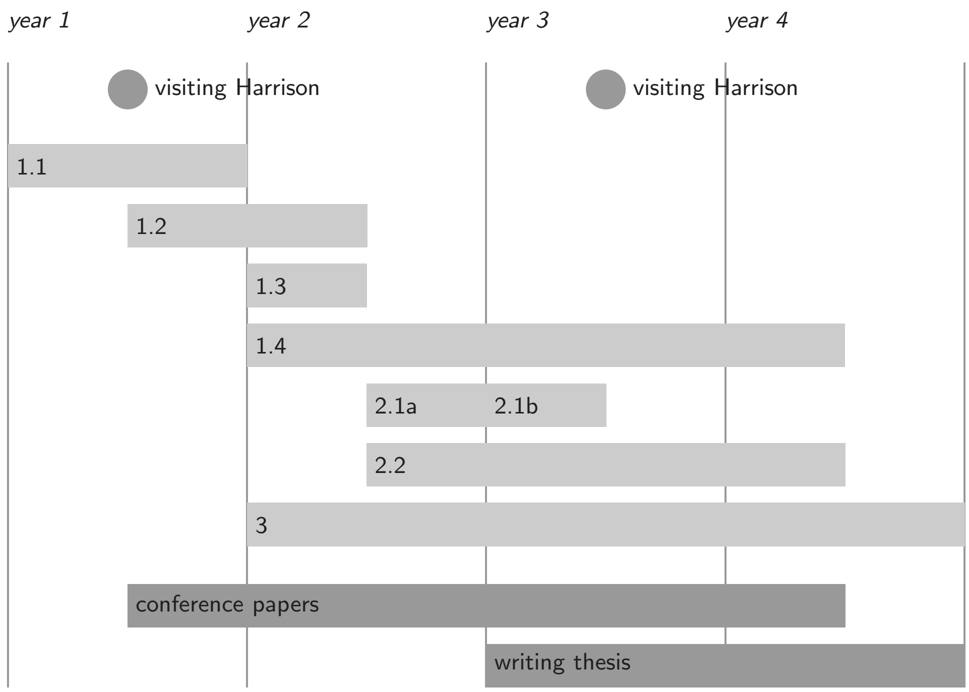

Fig. 7. Project schedule.

1.2. *Extend the HOL in HOL code to match the actual kernel.*
The current HOL in HOL formalization leaves out some important features from the HOL Light kernel. The HOL in HOL code now is extended to exactly match the actual HOL Light kernel.

1.3. *Run HOL Light on top of the extended HOL in HOL code.*
The translation from 1.1 now is applied to the code from 1.2, and a version of HOL Light is created in which the result of this has replaced the current HOL Light kernel. Because the HOL in HOL definitions closely mimic code from the HOL Light kernel, this is not a large change.

1.4. *Extend the correctness proof to the extended HOL in HOL code.*
Finally the correctness proof from the HOL in HOL project is extended to cover the extended HOL in HOL code.

Phase 2: *Obtain further correctness evidence.*

2.1. *HOL in HOL outside OCaml.*

2.1a. *Run the verified kernel inside HOL Light by rewriting.*

The code from 1.1 that translates HOL definitions into OCaml has not been verified, and might contain bugs that taint the correctness result about the HOL in HOL code. For this reason the HOL in HOL program also is executed *inside* the HOL system, by using the definitions as rewrite rules. This is much slower, but is certain to be correct.

2.1b. *Check the correctness proof in other HOL provers.*
The formalization is imported in other theorem provers to check the correctness by different code written for different compilers. Translators of HOL Light code to Isabelle/HOL [34], and HOL4 and ProofPower [20, 21] are applied, and the results checked in these other systems.

2.2. *Prove the OCaml translation correct inside HOL Light.*
It also is possible to check the correctness of the OCaml translation *inside* HOL. The HOL semantics of OCaml Light by Scott Owens [36] can be combined with a proof producing version of the HOL to OCaml translator from 1.1 to get a validated version of the OCaml code.

Phase 3: *Integrate OCaml and HOL development.*

We manually translated OCaml code to HOL definitions (Step 1.2), and then had the computer translate the result back to OCaml (Step 1.3). For this project that is the practical way to go about it, but for development of validated software in general it is a convoluted way of working. A more integrated style of software development is now developed, with in particular compilation from OCaml to HOL. Programs then can be written in OCaml, while the corresponding HOL definitions are created automatically.

This project is a direct continuation of existing work by John Harrison, and therefore two visits are planned for intensive contact between the PhD student and John Harrison. Both visits are marked in Fig. 7, and are planned for the first and third year of the project. These visits will either consist of the PhD student visiting Intel in the United States, or of John Harrison visiting the Radboud University Nijmegen in the Netherlands. (Ideally there will be one visit in each direction.) These visits will be around one month long. In the project budget below a special travel budget for these visits is included.

– *Educational Aspects.* The PhD student has to be an expert on functional programming (OCaml), theorem proving (HOL) and mathematical logic (HOL semantics). In the first years of the project he or she will interact with experts in these fields, and attend relevant courses at the Radboud University Nijmegen. The PhD student also will attend a summer school related to the field of interactive theorem proving.

8) **Expected Use of Instrumentation.** The only part of the project that involves heavy computation is 2.1a, which is not a bottleneck. Therefore there is no need for instrumentation beyond a regular workstation.

9) **Literature.**
The key publications of the research team relevant for the project are [18], [19], [22], [35] and [41].

1. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. Jean-Raymond Abrial. System Development in Event-B. Course notes and slides for a course at ETH Zürich, 2008.
3. Henk Barendregt and Freek Wiedijk. The Challenge of Computer Mathematics. *Transactions A of the Royal Society*, 363(1835):2351–2375, 2006.
4. Bruno Barras. Coq en Coq. Rapport de Recherche 3026, INRIA, October 1996.
5. Bruno Barras. Verification of the Interface of a Small Proof System in Coq. In E. Gimenez and C. Paulin-Mohring, editors, *Proceedings of the 1996 Workshop on Types for Proofs and Programs*, pages 28–45, Aussois, France, December 1996. Springer-Verlag LNCS 1512.
6. Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.
7. Bruno Barras and Benjamin Werner. Coq in Coq. ⟨`http://pauillac.inria.fr/~barras/coqincoq.ps.gz`⟩.
8. Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal Verification of a C Compiler Front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.
9. Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2008.
10. Marius Cornea, John Harrison, and Ping Tak Peter Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.
11. Gilles Dowek and Robert Boyer. Towards checking proof checkers. In *Informal Proceedings of the Nijmegen Workshop on Types for Proofs and Programs*, 1993.
12. Torkel Franzén. *Gödel's Theorem, An Incomplete Guide to Its Use and Abuse*. A K Peters, 2005.
13. Georges Gonthier. A computer-checked proof of the Four Colour Theorem. ⟨`http://research.microsoft.com/~gonthier/4colproof.pdf`⟩, 2006.
14. Mike Gordon and Tom Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.
15. John Harrison. A Machine-Checked Theory of Floating Point Arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics: TPHOLs '99*, volume 1690 of *LNCS*, 1999.
16. John Harrison. *The HOL Light manual (1.1)*, 2000.

17. John Harrison. Floating-Point Verification using Theorem Proving. In *Proceedings of SFM 2006, the 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 2965 of *LNCS*, 2006.
18. John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Proceedings of the Third International Joint Conference IJCAR 2006*, volume 4130 of *LNCS*, pages 177–191, Seattle, WA, 2006. Springer.
19. John Harrison. Formalizing an Analytic Proof of the Prime Number Theorem (extended abstract). In Richard Boulton, Joe Hurd, and Konrad Slind, editors, *Tools and Techniques for Verification of System Infrastructure*, pages 17–22, London, 2008. The Royal Society.
20. Joe Hurd. OpenTheory, 2009. ⟨`http://www.gilith.com/software/opentheory/`⟩.
21. Joe Hurd. OpenTheory: Package Management for Higher Order Logic Theories, 2009. Slides for a talk at the Galois Tech Seminar, ⟨`http://www.gilith.com/research/talks/galois2009-opentheory.pdf`⟩.
22. Cezary Kaliszyk and Freek Wiedijk. Certified Computer Algebra on top of an Interactive Theorem Prover. In M. Kauers, M. Kerber, R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants, Calculemus 2007*, volume 4573 of *LNCS*, 2007.
23. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, 2000.
24. Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik*, 38, 1931.
25. Xavier Leroy. Formal Certification of a Compiler Back-end, or: Programming a Compiler with a Proof Assistant. In *POPL'06*, Charleston, South Carolina, USA, 2006.
26. Michael Naixin Li, Yashwant K. Malaiya, and Jason Denton. Estimating The Number of Defects: A Simple and Intuitive Approach. In *Proc. 7th Int'l Symposium on Software Reliability Engineering (ISSRE)*, 1998.
27. Lemma 1 Ltd. *ProofPower – Description*. Lemma 1 Ltd., 2000.
28. William McCune and Olga Shumsy. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, 2000.
29. James McKinna and Robert Pollack. Some lambda calculus and type theory formalized. *Journal of Automated Reasoning*, 23, 1999.
30. Michał Muzalewski. *An Outline of PC Mizar*. Fondation Philippe le Hodey, Brussels, 1993.
31. Peter G. Neumann. The Risks Digest, forum on risks to the public in computers and related systems, 2009. ⟨`http://catless.ncl.ac.uk/risks/`⟩.

32. Tobias Nipkow, Larry Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
33. Michael Norrish and Konrad Slind. *The HOL system, Description*, 2007.
34. Steven Obua and Sebastian Skalberg. Importing HOL into Isabelle/HOL. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 298–302. Springer, 2006.
35. Russell O'Connor. A monadic, functional implementation of real numbers. *Mathematical Structures in Computer Science*, 17:129–159, 2006.
36. Scott Owens. A Sound Semantics for OCaml light. In Sophia Drossopoulou, editor, *Programming Languages and Systems, 17th European Symposium on Programming, ESOP 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary*, volume 4960 of *LNCS*. Springer, 2008.
37. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Berlin, Heidelberg, New York, 1992. Springer.
38. Randy Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.
39. Randy Pollack. How to Believe a Machine-Checked Proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, Oxford, 1998.
40. Joakim von Wright. The Formal Verification of a Proof Checker. Technical report, SRI, 1994.
41. Freek Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNCS*. Springer, 2006. With a foreword by Dana S. Scott.
42. Freek Wiedijk. Formal Proof – Getting Started. *Notices of the AMS*, 55(11):1408–1414, 2008.
43. Freek Wiedijk. Formalizing 100 Theorems, 2009. ⟨`http://www.cs.ru.nl/~freek/100/`⟩.

10) **Requested Budget.**

PhD student for 4 years			
a) appointment (incl. benchfee)	standard amount	=	€ 195,424
b) additional travel budget	(*detailed below*)	=	€ 7,600 +
Total		=	k€ **203**

Budget for visiting Harrison

return ticket Netherlands/USA	€ 800		
living costs abroad for one month	€ 3,000 +		
1 × visiting Harrison	€ 3,800		
2 × visiting Harrison	2 × € 3,800	=	€ 7,600 +
Total		=	€ 7,600

(The motivation for two visits that will allow the PhD student to work with John Harrison is given in Section 7 on page 12.)