A proof system for correct program development

Case for Support to accompany EPS(RP)

M.P. Fourman J.D. Fleuriot J.R. Longley

2 Proposed research and its context

A. Background

Much research in computer science, ever since its inception, has been devoted the problem: "How can we be sure that a computer program is correct?" The general problem is extremely difficult, and the enormous variety of computer software in use demands a corresponding variety of approaches: e.g. structured design methods [YC86], automated testing [Ber91] and model checking [GL94]. Another possibility—in some sense the most idealistic—is the formal development of programs with mathematical proofs of correctness claims. If this ideal is ever to become a reality, it is widely agreed that certain basic requirements must be met:

- The language in which programs are written should itself have a mathematically rigorous definition.
- A logic is required for expressing and proving properties of programs.
- Machine tools are needed to support the construction of proofs, and to check their validity.

In our view, whether the ideal of provably correct software is ultimately achievable hinges mainly on whether a sufficient level of automation can be achieved in the construction of proofs.

Regarding the first of the above requirements, the definition of Standard ML [MTHM97], developed mainly in the eighties, showed how a rigorous formal foundation could be given for a full-scale programming language. One of the intentions of the designers was precisely to provide a suitable platform for formal program development. The uptake of ML was at first modest, but recently there have been some notable commercial successes and real-world applications (e.g. [EHM+99, Buh95]). Moreover, aspects of ML such as strong typing and the exceptions system have significantly influenced the design of languages such as Java [GJS96], and it seems likely that future systems languages will incorporate many of these features [Mac00].

Regarding the second requirement, even before the definition of ML had fully taken shape, the LCF system [GMW78] provided a program logic for a rather restricted fragment of the language. Subsequent research has sought to build on the definition in order to support formal reasoning about programs. Most notably, the Extended ML project [KST97] resulted in a formal language for specifying program properties, but the complexity of this language prohibited the development of useful proof rules. A different approach has been pursued by Elsa Gunter *et al* [GV94], who have formalized the definition of ML within the HOL theorem prover; this has proved useful for metatheoretical studies, but seems too low-level for practicable program correctness proofs.

At the same time, the programming language features embodied in ML have attracted a great deal of attention from the theoretical computer science community. Here, much research has been devoted to the study of abstract mathematical structures that can be used to give semantics for these features; the long-term motivation has been that a deeper mathematical understanding should shed light on the kinds of logic required for reasoning about programs.

The problem of finding clean and expressive program logics has been a major motivation behind our theoretical work to date (see e.g. [PF92, FT95, Lon95, LP97]). Recently we realized that in terms of the underlying theory, we now have all the pieces we need in order to achieve this goal for a very substantial language, including the full power of a higher-order functional language, plus (we believe) most everyday uses of exceptions, references and input/output. This puts us in a significantly better position than ever before to attack the program verification problem.

Finally, regarding the third of our basic requirements, the LCF project itself showed how one can provide machine support for rigorously checked formal proofs, whilst allowing unlimited scope for automation in their construction. However, the problem of providing adequate automation, so as to render software verification feasible in practice, has not yet been satisfactorily overcome. A substantial body of work from the AI community is relevant here, but relatively little has been done to bring this to bear on program verification—partly, we believe, because of the lack of a good program logic hitherto on which to base such an effort.

The proposed research will apply recent developments in semantics to provide a powerful proof system for a large part of ML, including almost all the features used commonly in programming practice. We will develop a prototype implementation of our proof system using the Isabelle theorem prover [Pau94]; investigate the applicability of a variety of automation techniques, including proof planning using the λ-Clam system [RSG98]; evaluate our proof system via some non-trivial

programming examples; and finally use this system as the basis of our own customized theorem prover. This will result in a tool suitable for use by the program specification and verification communities, providing a platform for research on formal development methodologies.

To summarize this in terms of the basic requirements mentioned above: The definition of Standard ML has already demonstrated how the first requirement may be met. In this project we will show that the second requirement can now be largely fulfilled, by building on insights from recent theoretical work. We will also make some partial progress with the third requirement, using the fruits of recent work from the AI theorem-proving community.

Many of our ideas and methods are in principle applicable to a wide range of programming languages, but there are a number of reasons why Standard ML is a natural testbed for our ideas. Firstly, the semantic techniques we will employ are at present more fully developed for functional languages than for other genres (e.g. object-oriented languages). Secondly, the rigorous formal definition of ML gives us a very solid foundation on which to build. Thirdly, the trend in much current work on next-generation systems language design is towards languages combining higher-order type systems with imperative features [HLP98, Mac00], and ML provides a good illustration of this combination. Towards the end of the project we will assess what needs to be done to apply ideas to other languages (see Section D).

B. Programme and methodology

In this project, we will work out the details of a program logic by drawing on recent theoretical research, and provide machine tools to support reasoning in this logic. We first mention some general features of the logic we have in mind, drawing attention to other related work. A slightly fuller explanation of the more theoretical aspects of our proposal [Lon00] is available online at `www.dcs.ed.ac.uk/home/jrl/ml-logic.ps`.

- Firstly, our approach is *axiomatic* in the spirit of LCF or Extended ML—that is, we work with clean high-level axioms for the behaviour of programs, rather than with the operational semantics as in [GV94]. Moreover, terms of our logic are just ML expressions—no translation from ML into a logical language is needed.[1] These features allow programmers to reason relatively directly at a level they are familiar with. An axiomatic approach also means that our logic is less tied to the specifics of the operational definition, and hence more easily transferable to other languages.
- Like the Extended ML project, we take seriously the challenge of working with a *full-scale* programming language rather than a toy theoretical language. We have been influenced by Extended ML in many respects, and will benefit from the experience of Sannella and his co-workers in the intricacies and pitfalls of the ML definition.
- We make crucial use of recent insights from *denotational semantics* (this is the main difference from the Extended ML approach). In this respect, we build largely on theoretical foundations laid by Longley [Lon98, Lon99b, Lon99a, LP97] in previous EPSRC-funded research.[2] Certain well-understood denotational models will provide the technology for showing the soundness of our proof rules; but more importantly, they will provide conceptual guidance in the design of our logic. In particular, the fragments of ML that we treat will be determined by what we know how to model well. In our view, this use of semantics is the key to obtaining a clean and manageable proof system, and to avoiding the explosion in complexity that beset the Extended ML project.
- A key feature of our approach is that the meaning of the logic can be explained in terms of *purely operational* concepts, such as observational equivalence of program fragments; this means that the logic will be intelligible to programmers with no knowledge of the semantic underpinnings. We achieve this by using denotational models that enjoy an exceptionally tight fit with the programming language, via the concept of *logical full abstraction* developed in [LP97]. As far as we know, this particular emphasis is unique to our approach.
- Our approach is *stratified* in that we will actually carry out our programme for three different sublanguages of ML, of increasing size and difficulty: a functional fragment $\mathcal{L}_1$; a fragment $\mathcal{L}_2$ with control features (certain uses of exceptions); and a fragment $\mathcal{L}_3$ with imperative features (certain uses of state and input/output). The choice of these sublanguages is not arbitrary, but corresponds to a series of three mathematically well-understood models that all fit well into a uniform semantic framework.[3] (Of course, part of our task will be to give careful syntactic definitions of these sublanguages.) The idea is that the three logics $K(\mathcal{L}_i)$ that we obtain from them will fit together in a pleasant way; this means, for instance, that we can use $K(\mathcal{L}_1)$ to reason about the purely functional parts of a program, resorting to the more delicate machinery of $K(\mathcal{L}_2)$ or $K(\mathcal{L}_3)$ only for those parts of the program that involve control or imperative features. Likewise, once we have proved that a program involving imperative features (e.g. a memoization operation) behaves in a purely functional way, we can revert to $K(\mathcal{L}_1)$ for reasoning about it.

[1] These aspects of our proposal have roots in the logic of the LAMBDA 3.2 theorem prover [FF90], and the ideas of Fourman and Phoa [PF92].

[2] EPSRC grants GR/L89532 "Notions of computability for general datatypes" and GR/J84205 "Frameworks for programming language semantics and logic."

[3] Namely, the category of PERs on the van Oosten/Longley combinatory algebra [Oos97, Lon98].

As a somewhat separate issue, we will also add some support for data abstraction (`abstype`s and opaque signature constraints). Thus, we believe that we will cover most of the features of ML occurring commonly in programming practice. Of course, with future advances in semantic understanding it may be possible to handle even larger fragments of the language, though this is likely to yield diminishing returns beyond a certain point.

- Our logics will be *sound* and *relatively complete*.[4] Soundness ensures that all provable theorems are true under the operational interpretation; completeness ensures that (in some sense) no axioms are missing from our system. Absolute completeness is too much to hope for in view of Gödel's theorem, but our axiomatizations will be, in a precise sense, as complete as first-order Peano arithmetic.[5]

  This means that we do not sacrifice any logical power by going for a high-level axiomatic approach. This contrasts interestingly with the LOOP project on Java program verification [JvdBH+98], to which our proposal is similar in spirit in many other respects. In LOOP, one uses a combination of axiomatic reasoning (Hoare-style logic) with reasoning *about* the semantics via a translation from Java to higher-order logic. By dispensing with the latter, we believe we will achieve much greater abstraction and ease of reasoning; the trade-off is that we do not yet have the semantic technology to apply our approach to an object-oriented language.

On a more technical level, starting from a sublanguage $\mathcal{L}$ of Standard ML, we build a many-sorted first-order classical logic $K(\mathcal{L})$ whose sorts are types of $\mathcal{L}$ and whose terms are terms of $\mathcal{L}$, with atomic predicates for equality and termination. We then give an operational interpretation for formulae of $K(\mathcal{L})$: for instance, equality is interpreted as observational equivalence relative to $\mathcal{L}$, and variables are taken as ranging over terms of $\mathcal{L}$. Finally, we axiomatize our logic by making use of logically fully abstract denotational models of $K(\mathcal{L})$ (see [LP97]). The model we have in mind is nothing other than the closed term model for $\mathcal{L}$ modulo observational equivalence, but the point is that we have other more semantic characterizations of this model which give us a better handle on the logic. The relative completeness is achieved by a technique of bootstrapping from lower to higher types; this exploits the existence of *universal* types (i.e. types within which all other types can be represented as retracts) for the languages in question.

How will we know that our proof systems are sound? Once we have formulated the proof systems in detail, these will have the status of precise mathematical claims involving the definition of ML. In principle, one could imagine a machine-checked verification of these properties based on something like Gunter's encoding of the definition, but we will leave this as an ambitious possibility for future work. In the meantime, we will content ourselves with producing careful statements of our claims, together with clear outlines of the proofs. Whether our claims are valid in every detail will have the status of a working scientific hypothesis, to be subjected to the scrutiny of experts in the manner of [Kah93].

Next we outline the stages of our programme. Some overlap between the various stages will be possible once the basics are in place (see the Diagrammatic Workplan).

I. Design of logic and Isabelle prototype

We will begin by identifying those language features we wish to study, demarcating the corresponding sublanguages of ML, working out the details of the logical axioms, and embodying all this in a prototype implementation using the Isabelle generic theorem prover [Pau94]. Our preliminary work here has shown that Isabelle is a very suitable tool for this purpose, owing to the flexibility of its syntax-encoding mechanisms, and the ability to experiment cheaply with different sets of axioms.

We have also discovered that some language features (e.g. record types, and certain details of ML syntax) are awkward to encode faithfully in Isabelle.[6] We emphasize, however, that these difficulties arise purely from certain restrictions imposed by working with Isabelle, rather than from any inherent theoretical difficulty in devising a logic for the features concerned. The intention is that later on we will build our own customized theorem prover, which will iron out these difficulties and incorporate language features missing from our prototype (see IV below).

Along with our Isabelle implementation we will produce user documentation, together with documents explaining the underlying theory and justifying the soundness and completeness claims.

I.1. The functional fragment We will first consider a functional fragment $\mathcal{L}_1$; this is the fragment that we expect it to be easiest to reason about. We have already devoted about two man-months' work to this part of the programme, and now have a prototype implementation in Isabelle for most of the functional fragment. With our present system, the user may write Isabelle theory files incorporating function declarations in ML syntax; the system then generates a defining axiom for each function (see [Lon00] for an example). So far, we have encoded the ML syntax and logical axioms for: the basic types

[4] At present we know how to achieve relative completeness only for monomorphic formulae—in the proposed project we will try to extend this to polymorphic formulae.

[5] If required, PA can be replaced here by stronger theories, e.g. ZF set theory.

[6] This is partly because we are adopting a *shallow* embedding wherein ML types are represented by Isabelle types—this means, for instance, that the Isabelle typechecker does ML type inference for us.

`unit`, `bool`, `int` and `'a list`; product and (higher order) function types; `val` and `fun` declarations with general pattern matching (including overlapping and non-exhaustive patterns); polymorphism (including equality types); simultaneous and mutually recursive declarations; `fn` and `case` expressions; `while` expressions; infixes; explicit type constraints (in expressions and patterns); and layered patterns. Our efforts so far provide encouraging evidence that our approach does indeed scale up to the intricacies of a real programming language.

Besides consolidating what we have done so far, we will add `let` and `local` constructs[7] and `datatype` declarations. With regard to the latter, the ideas of [SP82, Pit94], in combination with properties of universal types, lead to good axioms even for mixed-variance recursive types.

I.2. Control features There is a class of toy languages—all equivalent from our point of view—which embody a notion of functional programming with control (e.g. PCF+`catch`; μPCF; PCF with first-order `callcc`). These languages suffice for modelling exceptions in Java, for instance, and from a semantic point of view they are now very well understood (see e.g. [CCF94, OS97, Lai98, Lon98]).

We will define a language $\mathcal{L}_2$ by isolating a (syntactically defined) class of "safe" uses of exceptions in ML corresponding to the above languages. Thus, the known semantic models will guide us in choosing a sublanguage with pleasant logical properties, which will integrate smoothly with $\mathcal{L}_1$ and $\mathcal{L}_3$. Our ideas for a logic of exceptions are outlined in [Lon00].

I.3. Imperative features State and input/output are staple features of systems languages, and some use of state is essential if functional languages are to offer competitive efficiency. However, even apparently simple functional languages with state can display very complex behaviour unless the use of state is restricted somehow (see e.g. [Rey78, OPTT99, PS98]).

Our intention is to work with a suitable class of safe uses of first-order references (also of input/output), somewhat analogous to the safe uses of exceptions mentioned above—this will ensure that we avoid unpleasant extrusion effects, for instance. Some of the theory here still needs to be worked out in detail. In the first instance, we will isolate a language $\mathcal{L}_3$ that corresponds a particular semantic model we have constructed (see [Lon00] for more details). We are confident that this at least will work, though it is not yet clear how much programming power it will give us (e.g. how far it will allow us to treat equality of references or parameter passing by reference). We will also consider whether other approaches to languages with state allow us to do better, e.g. [AHM98, PS98, OPTT99].

Finally, we will add the logical principles needed to relate the logics $K(\mathcal{L}_1), K(\mathcal{L}_2), K(\mathcal{L}_3)$ to each other, allowing all three logics to be subsumed in one big proof system if desired.

I.4. Data abstraction We will also consider extending our fragments of ML by adding data abstraction, as given by ML `abstype`s or opaque signature constraints. This kind of extension is somewhat orthogonal to the sequence of languages $\mathcal{L}_1 \subset \mathcal{L}_2 \subset \mathcal{L}_3$ outlined above.

At present, we know how to extend our approach smoothly to a certain (semantically motivated) class of abstract types, provisionally called *translucent* datatypes.[8] Examples of translucent abstract types include all the usual implementations of stacks, queues, sets, multisets and lookup tables, and the abstract type of theorems in an LCF-style theorem prover. One problem, at present, is that in order to prove anything about an abstract type, one would first need a theorem saying that it is translucent. We may be able to do better than this by extending our theoretical understanding of `abstype`s; however, our primary goal in the present project is to see how much can be achieved with the currently understood theory.

II. Automation and proof heuristics

The general problem of making interactive theorem proving tractable for large examples is a very difficult one, involving cognitive as well as logical issues, but all are agreed that a high degree of automation is an essential component. We will develop various forms of automated assistance:

II.1. Decision procedures and related tools Many standard techniques have already been used in conjunction with interactive provers such as PVS, HOL and Isabelle (for example, arithmetical decision procedures [CLS96], term rewriting [Nip89], model checking [Sha96] and BDDs [Gor00]). We will adapt and make use of these techniques where appropriate (particularly the first two). There is also scope for developing new decision procedures specific to our logic.

II.2. Proof planning A more novel aspect of our proposal is to investigate the use of *proof planning* from AI [Bun91] for reasoning about programs. There are specific reasons why we think proof planning is particularly well-suited to program verification. Our experience of verifying small programs by hand is that almost everything is proved by induction, and most

[7] A makeshift is required to give the appearance of capturing `let`-polymorphism in Isabelle. See also IV below.

[8] Other people have used this term with a different meaning, e.g. [HL94].

of the difficulty lies in choosing just the right strength of induction hypothesis—it often takes two or three attempts before this is found. (This corresponds to the problem of choosing the right loop invariants when verifying imperative programs.) It is precisely in this area of inductive proof—and of constructing a correct proof out of previous failed attempts—that work on proof planning has specialized.

Fleuriot has recently proposed an EPSRC-funded project [Fle00] to develop a generic proof planning system by integrating Isabelle with the λ-Clam proof planner [RSG98], developed at Edinburgh by Bundy, Richardson *et al.* The present project will complement this by applying this system to a particular target logic and testing it on some hard application problems.

III. Examples and case studies

We will evaluate our approach via examples of programs formally specified and verified using our proof system. It is difficult to anticipate the scale of the examples that will be feasible—this will depend largely on the success of the automated proof mechanisms in practice—but we believe we will be in a position to tackle more substantial examples than have previously been possible. Early case studies may include ML implementations of[9]

- the Fast Fourier Transform (FFT) algorithm;
- the RSA public-key cryptosystem;
- Ordered Binary Decision Diagrams (OBDDs);
- (more ambitiously) a calculator for exact real arithmetic.

A good source of further examples will be the "pearls" published in the Journal of Functional Programming. We will focus particularly on verifying efficient programs involving clever optimizations whose correctness is not obvious.

We emphasize that the success of our approach should be gauged not only by the size and complexity of the programs we can verify, but by the ease with which we can do it. Interactive theorem proving has resulted in some impressive achievements (e.g. in hardware verification), but it seems fair to admit that many of these have been accomplished more by heroic perseverance than by generally tractable methods. Our aim will be to investigate what can be done on a modest timescale, and moreover what we can expect others to reproduce.[10]

IV. Customized system

As mentioned earlier, it is difficult to encode certain features of ML faithfully within the existing Isabelle system. Some of the difficulties are merely matters of surface syntax, but others seem more far-reaching and reflect deeper meta-logical issues (e.g. `let`-polymorphism;[11] the modules system). Once our prototype has stabilized, we plan to modify Isabelle to create our own theorem proving system, customized to our logic for ML and not subject to these restrictions. The new system will be smoothly compatible with the ML definition, and will be generally more friendly to ML programmers (for instance, one will be able to load in an ML source file directly as a "theory file" rather than writing special Isabelle theory files).

IV.1. The modules system The main intellectual challenge in this part of the project will be to incorporate (as much as possible of) the ML modules system in a principled way. Part of the issue here is to design a modules system for *theories* within the theorem prover that closely reflects the modules system for programs—existing work on Extended ML and on modules in OBJ [GWM+00] may be relevant here. There are also specific challenges associated with opaque signature constraints (see I.4 above) and ML functors—again, our approach will be to see how much we can achieve with our current semantic understanding.

IV.2. Implementation In this project, we intend merely to start what we hope may become an ongoing endeavour. Building a theorem prover is a major undertaking, and we plan to re-use available components wherever we can. In particular, we expect that much of the existing Isabelle code will serve our purposes well, and that the automated tools developed earlier in the project will slot in easily. We also intend to borrow a parser and type-checker from the ML Kit Compiler [BRTT93], and to take advantage of Aspinall's Proof General interface [Asp99]. If appropriate, we may use parts of the PROSPER proof environment toolkit [DCN+00].

We do not aim for a tool engineered to the level of a commercial product, but for a reasonably clean and robust system suitable for distribution and use in the research community. Fourman's experience as architect of the LAMBDA theorem prover will be invaluable here; we are also eager to profit from the experiences of other colleagues who have implemented theorem provers (e.g. in Edinburgh and Cambridge).

[9] Some of these examples have previously been verified as abstract algorithms, but we will be verifying actual code.

[10] Case studies will thus furnish ideal material for student projects at M.Sc. and final-year undergraduate level.

[11] The inconsistency of `let`-polymorphism with HOL-style logic is not a problem for us, since formulae are not terms in our logic.

Once the basics are in place, we will be able to run large samples of code through our system (without necessarily proving anything about it), in order to assess our claim that we are handling most of the commonly used features of Standard ML. This may highlight strategic areas for further research.

C. Relevance to beneficiaries

In the short term, our project will benefit workers in the formal methods and software verification communities. Software verification has been talked about for decades, and there is an enormous literature on methodologies for formal software development, but many of these ideas have never really been put to the test—largely because of the absence of appropriate tools (both conceptual and mechanical). Our work will therefore fill a significant gap, providing a platform for case studies and opening up new directions in formal methods research. Like the HOL, LAMBDA and PVS theorem provers, we hope that our tools will also enjoy some level of use within industry for safety-critical applications.

We also expect our work to have a bearing on the next generation of systems languages. There is a significant move amongst language designers towards the adoption of more ML-like features (e.g. strong typing, exceptions), and the application of such languages to systems programming [HLP98, Mac00]. Our work will therefore be an important step towards bringing formal techniques to bear on real-world software. In the long term, we expect our basic strategy to be transferable to languages of any genre (e.g. object-oriented languages).

Finally, our project will bring together two different research communities (theoretical computer science and AI), and we expect this to be enriching for both. On the one hand, it will give an indication of how far current semantic theories address the demands of programming practice; on the other hand, it will offer proof planning techniques some challenging test problems, which will stimulate further refinements of these techniques.

D. Dissemination and exploitation

As usual in our research community, the main modes of dissemination will be refereed journal papers, conferences, informal workshops and personal visits to other research establishments. In addition, our software (Isabelle version and customized version), together with supporting documentation and examples, will be made available to colleagues via the Internet.

Our contacts with industry (e.g. Microsoft Research; Bell Labs; Compaq Research) will provide a rich source of potential applications. Towards the end of the project, we intend to host a workshop for industrial and academic colleagues in formal methods, safety critical systems and software verification, both to offer instruction in the use of our tools to verify programs, and to discuss how to render our technology applicable to industrial or commercial projects.

E. Justification of resources

Manpower Longley will work full-time on the project as a postdoctoral research associate for the whole three years. Funding for Longley is requested on the AR1A to start at spinal point 9 with annual increments. Fleuriot will contribute an average of three hours a week for the last two years of the project. Prof. Fourman will contribute an average of three hours a week to the project, and conduct termly project review meetings. We also seek funding for 15% of a computing officer (AD3.3) to provide infrastructure support for development and use of our machine tools, and 10% of a secretary (CN3.3) to support publication and dissemination of our results.

Travel We plan to attend an average of two international conferences a year, e.g. LICS, MFPS, POPL, ICALP, ETAPS, MFCS, CSL, TACS, CTCS, CADE, CAV, TPHOLs. We request travel and subsistence support for seven one-week visits to European centres (e.g. Aarhus, INRIA), and two 1-2 week visits to a number of centres within the US (e.g. CMU, SRI/Stanford, Pennsylvania, Cornell). These visits will be made in conjunction with overseas conference travel. We also request support for two 2-3 day visits per year within the UK (e.g. Birmingham, Cambridge, Oxford, QMW).

Equipment A workstation, maintained over the project period, is requested to support the development of our proof system and its evaluation via examples. A portable computer is requested for demonstrating our system at conferences and on other visits. We also request an appropriate contribution to consumables, shared networking, and fileserver provision.

References

[AHM98] S. Abramsky, K. Honda, and G. McCusker. A fully abstract game semantics for general references. In *Proc. 13th Annual Symposium on Logic in Computer Science*. IEEE, 1998.

[Asp99] D. Aspinall. Proof General: A generic tool for proof development. Presented at ETAPS 2000; available from `http://zermelo.dcs.ed.ac.uk/~proofgen/`, 1999.

[Ber91] A. Bertolino. An overview of automated software testing. *Journal of Systems Software*, 15:133–138, 1991.

[BRTT93] L. Birkedal, N. Rothwell, M. Tofte, and D.N. Turner. The ML Kit (version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[Buh95] J. Buhler. The Fox project: A language-structured approach to networking software. *ACM Crossroads*, 2(1), 1995. Available electronically via `www.acm.org/crossroads/`.

[Bun91] A. Bundy. A science of reasoning. In *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[CCF94] R. Cartwright, P.-L. Curien, and M. Felleisen. Fully abstract semantics for observably sequential languages. *Information and Computation*, 111(2):297–401, 1994.

[CLS96] David Cyrluk, Patrick Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction—CADE-13*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[DCN+00] L. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, and T. Melham. The PROSPER toolkit. In *Proc. 6th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems, Berlin*. Springer LNCS 1785, 2000.

[EHM+99] P.H. Eidorff, F. Henglein, C. Mossin, H. Niss, M.H. Sorensen, and M. Tofte. Annodomini: From type theory to year 2000 conversion tool. In *ACM POPL Symposium*, 1999.

[FF90] S. Finn and M.P. Fourman. *Logic Manual for the Lambda System, version 3.2*. Abstract Hardware Ltd., 1990.

[Fle00] J.D. Fleuriot. A generic approach to proof planning (case for support). EPSRC grant proposal GR/N37414, 2000.

[FT95] M. Fourman and H. Thielecke. A proposed categorical semantics for ML modules. In *Category Theory in Computer Science*. Springer LNCS 953, 1995.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. Programming Languages and Systems*, 16(3):843–871, 1994.

[GMW78] M. Gordon, R. Milner, and C. Wadsworth. *LCF: A Mechanised Logic of Computation*. Springer, 1978.

[Gor00] M. Gordon. Reachability programming in HOL98 using BDDs. Submitted to TPHOLs, 2000.

[GV94] E.L. Gunter and M. VanInwegen. HOL-ML. In Jeffery Joyce and Carl Seger, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 780 of *Lecture Notes in Computer Science*, pages 61–73. Springer-Verlag, February 1994.

[GWM+00] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. *Introducing OBJ*. Kluwer, 2000.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st POPL Symposium*, pages 123–137. ACM Press, 1994.

[HLP98] Robert Harper, Peter Lee, and Frank Pfenning. The Fox project: Advanced language technology for extensible systems. Technical Report CMU-CS-98-107, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1998. (Also published as Fox Memorandum CMU-CS-FOX-98-02).

[JvdBH+98] B. Jacobs, J. van den Berg, M. Huisman, U. Hensel, and H. Tews. Reasoning about Java (preliminary report). In *OOPSLA'98*, pages 329–340. ACM Press, 1998.

[Kah93] S. Kahrs. Mistakes and ambiguities in the definition of Standard ML. Technical Report ECS-LFCS-93-257, Department of Computer Science, University of Edinburgh, 1993.

[KST97] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theor. Comp. Sci.*, 173, 1997.

[Lai98] J. Laird. *A Semantic Analysis of Control*. PhD thesis, University of Edinburgh, 1998. Examined March 1999.

[Lon95] J.R. Longley. *Realizability Toposes and Language Semantics*. PhD thesis, University of Edinburgh, 1995. Available as ECS-LFCS-95-332.

[Lon98] J.R. Longley. The sequentially realizable functionals. Technical Report ECS-LFCS-98-402, Department of Computer Science, University of Edinburgh, 1998. To appear in *Annals of Pure and Applied Logic*.

[Lon99a] J.R. Longley. Matching typed and untyped realizability. In *Proc. Workshop on Realizability, Trento*. Elsevier Electronic Notes in Theoretical Computer Science, 1999. To appear.

[Lon99b] J.R. Longley. When is a functional program not a functional program? In *Proc. 4th International Conference on Functional Programming, Paris*, pages 1–7. ACM Press, 1999.

[Lon00] J.R. Longley. A sound and relatively complete program logic for most of Standard ML. Draft document, available at `www.dcs.ed.ac.uk/home/jrl/ml-logic.ps`, 2000.

[LP97] J.R. Longley and G.D. Plotkin. Logical full abstraction and PCF. In J. Ginzburg et al., editor, *Tbilisi Symposium on Language, Logic and Computation*, pages 333–352. SiLLI/CSLI, 1997.

[Mac00] D.B. MacQueen. Speculation on a modern systems language. Talk given at University of Edinburgh, March 2000.

[MTHM97] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML: revised 1997*. MIT Press, 1997.

[Nip89] T. Nipkow. Term rewriting and beyond—theorem proving in Isabelle. *Formal Aspects of Computing*, 1:320–338, 1989.

[Oos97] J. van Oosten. A combinatory algebra for sequential functionals of finite type. Technical Report 996, University of Utrecht, 1997. To appear in Proc. Logic Colloquium, Leeds.

[OPTT99] P.W. O'Hearn, A.J. Power, M. Takeyama, and R.D. Tennent. Syntactic control of interference revisited. *Theoretical Computer Science*, 228:211–252, 1999.

[OS97] C.-H.L. Ong and C.A. Stewart. A Curry-Howard foundation for functional computation with control. In *Proc. Symposium on Principles of Programming Languages*, pages 215–227. ACM Press, 1997.

[Pau94] L.C. Paulson. *Isabelle: A generic theorem prover*. Springer LNCS 828, 1994.

[PF92] W.K.-S. Phoa and M.P. Fourman. A proposed categorical semantics for Pure ML. In W. Kuich, editor, *Automata, Languages and Programming*. Springer LNCS 623, 1992.

[Pit94] A.M. Pitts. A coinduction principle for recursively defined domains. *Theoretical Computer Science*, 124, 1994.

[PS98] A. Pitts and I. Stark. Operational reasoning for functions with local state. In A. Gordon and A. Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.

[Rey78] J.C. Reynolds. Syntactic control of interference. In *Proc. 5th POPL Symposium*, pages 39–46. ACM, 1978.

[RSG98] J. Richardson, A. Smaill, and I. Green. System description: Proof planning in higher-order logic with λ-Clam. In *Automated Deduction – CADE-15*. Springer LNAI 1421, 1998.

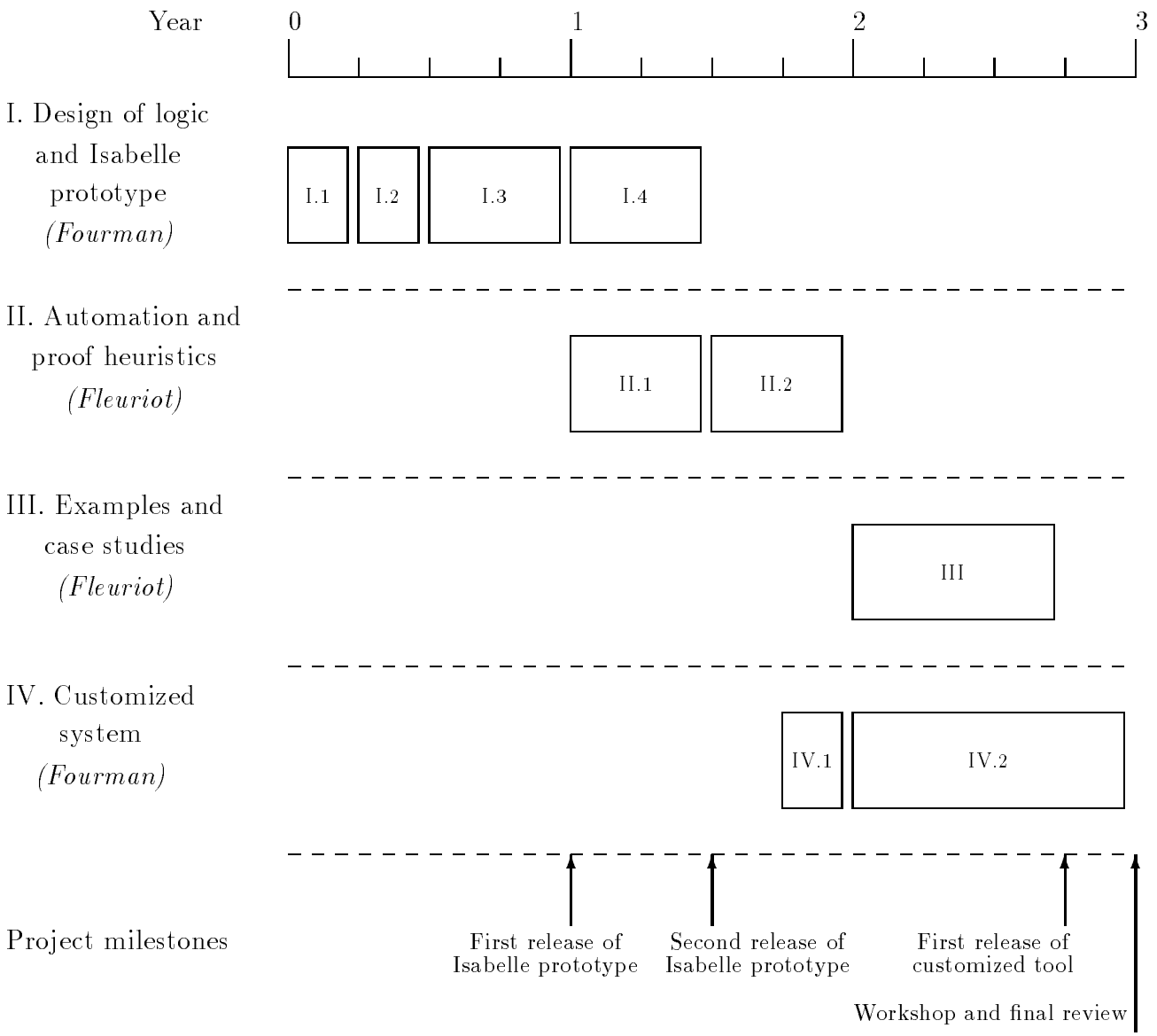
[Sha96] N. Shankar. PVS: Combining specification, proof checking, and model checking. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 257–264, Palo Alto, CA, November 1996. Springer-Verlag.

[SP82] M.B. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11, 1982.

[YC86] E. Yourdon and L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, 1986.

A proof system for correct program development

Diagrammatic Workplan



Remarks: Longley will work on all areas of the project; Fleuriot and Fourman will contribute to parts of the project as indicated above. Where tasks overlap, Longley will divide his time approximately evenly between them.

The task numbers (I.1, I.2 etc.) refer to paragraphs in the main proposal.