A programming language based on game semantics

Case for Support to accompany EPS(RP)

J.R. Longley

2 Proposed research and its context

A. Background

A great deal of research in theoretical computer science has sought to put computer programming on a firmer mathematical basis, in order to increase the reliability of programs. This includes both the problem of designing *languages* with good properties (such as a safe type system ensuring that programs do not derail because of type errors), and that being able to prove that individual *programs* do what they are supposed to. The latter, in particular, is acknowledged to be a very hard problem in general, but current work (including the author's own EPSRC-funded research) is reaching the point where at least the foundational and logical aspects of the problem can now be addressed, even for complex combinations of language features found in many modern programming languages.[1]

Whether we are concerned with language design or program verification, there are, broadly speaking, two basic strategies we can adopt. Either we can start with an existing language or collection of language constructs and try to find some mathematical theory to fit it, or we can start with some well-understood mathematical ideas and structures and design a programming language around these. The first strategy will ensure that we stay close to real-world computing practice, while the second will help to ensure that the mathematics remains clean and tractable. In practice, a combination of the two approaches seems desirable — one should try to address the kinds of language features that programmers actually use, but some modifications to existing languages might be desirable in order to smooth off rough edges that are troublesome from the logical point of view.

A case in point here is the development of the Standard ML programming language, starting in the mid-seventies. Part of the explicit intention of the designers was to design a language with a sound mathematical basis, in at least three senses. First, the *definition* of the language was to be mathematically rigorous and complete. Secondly, the language as a whole was to have good properties, such as type safety and the "no dangling references" condition. Thirdly, it was hoped that the nature of the language would be such as to facilitate the verification of programs written in it. The design of ML was guided in part by the mathematical ideas from denotational semantics which were then available (particularly with regard to the "functional core" of the language), and in part by the demands of programming practice (e.g. "imperative" features such as references and input/output were added).

Since then, the scope of denotational semantics has widened considerably. Thanks to more recent developments such as *game semantics*, we are now able to give clean mathematical models for languages with exceptions, state, input/output, subtyping, higher order polymorphism and more. Indeed, in certain respects the richness of structure possessed by these models now surpasses what is needed to describe current programming languages. We make essential use of some of these theoretical developments in our current EPSRC-funded project "A proof system for correct program development", in order to provide simple but powerful *logics* for reasoning about ML programs. Even so, our experience suggests that, with hindsight, the treatment of the imperative features in particular was not quite as amenable to formal reasoning as one might have hoped, so that when dealing with these features, we and others have found it preferable to work with a certain syntactically circumscribed *fragment* of ML (albeit one that covers almost all programs written in practice). Indeed, it seems to be a general characteristic of attempts to reason about programs in a real-world language that they tend to content themselves with a well-behaved subset of the language.

We believe that in order to make further progress, the time has now come to attempt the design of a full-scale, usable programming language with a sound mathematical basis, somewhat in the same spirit as ML but taking advantage of the more powerful technology from denotational semantics that is now available, as well as taking account of other recent developments and trends in language design. Broadly speaking, we have in mind a strongly typed, object oriented language with powerful polymorphism and a consistently higher order flavour, bringing together "the best of Java and the best of Standard ML", but breaking new ground in the way these features are combined and especially in its treatment of mutable and non-mutable data.

As we see it, the primary benefits of such a project would be threefold. First, we expect the rich mathematical structure of our semantic models to suggest innovative and powerful language features (particularly with regard to the type system) which will contribute to the stock of ideas and techniques in language design. Secondly, by virtue of its clean mathematical

[1] Another aspect of the problem is to provide adequate automation, tool support and user interfaces to make formal program development practicable — this is the focus of a slightly different line of current research.

underpinnings, our language will be particularly well-suited to program verification (indeed, our intention is to design the language hand-in-hand with a suitable logic for reasoning about it), and will thus provide a vehicle for ongoing research in this area. Thirdly, we also hope that our language will be found by many programmers to be a clean and beautiful one for a variety of applications, as has been the case with Standard ML.

We see this project as bringing together two strands of our research to date: our interest in natural *notions of computability* and in languages and models that embody them, and our current experience of working with the details of a full-scale programming language. A central idea behind our approach is that the semantic perspective and conceptual guidance afforded by the first of these strands is the key to avoiding a blow-up of complexity in the second. This philosophy will also be crucial in our approach to the currently proposed project: the essential simplicity of the semantic models will ensure that the various features of the programming language fit together in a clean and principled way.

Ours will not be the first attempt to combine aspects of Java and ML. Most notably, the OCAML language [**?**] offers a version of "ML with objects" which has achieved considerable popularity among functional programmers in recent years. The distinctiveness of our approach, however, lies in its semantic basis, which both provides a rigorous definition of the language and helps to ensure that it has good logical properties. (At present, OCAML does not have a formal semantics, and it is unclear whether it lends itself to program verification in the way we envisage.) We will regard it as a positive sign if some aspects of our language turn out to resemble OCAML; however, we believe that our language will go considerably further in its combination of object oriented and higher order features. (Below we will mention some particular aspects of OCAML from which we will draw inspiration.)

B. Programme and methodology

Creating a new programming language is obviously an massive undertaking, and in the current project we intend merely to carry out some exploratory research resulting in a prototype definition of the language and a crude "interpreter" for it, perhaps with a view to a more serious implementation effort at a later date. Our approach will be based on state-of-the-art developments in semantics, type theory and language design — most specifically, on a particular *game model* for computation which has proved very useful in our current work on logics for ML, but whose potential usefulness appears to extend far beyond ML and to embrace many aspects of object-oriented languages such as inheritance and dynamic binding, for instance. Our intention will be to allow the design of our language to grow naturally out of the structure available in this model.

Using our game model, we will design and provide a formal mathematical semantics for a prototype version of the language, together with a logic for reasoning about programs. In order to achieve this, some part of the project will also be taken up with consolidating and extending our theoretical understanding of the model. Finally, our game semantics will be naturally *executable*, so we will be able to obtain, almost for free, an "animation" of our language definition. This will allow us (and others) to experiment with the use of the various language features, which in turn will provide valuable feedback allowing us to improve the language design.

We now give a more detailed description of the main phases of the project. We will describe the phases in logical order, though we envisage them as being undertaken largely in parallel as there will be much interaction between them.

I. Semantic foundations

We have already identified the game model we intend to use as the semantic basis for the project. An intuitive and somewhat informal description was given in [**?**]; a more precise account, along with a sketch of how our model can be used to interpret various languages (in particular an object oriented language with inheritance and dynamic binding), will be forthcoming as part of our present EPSRC-funded project.

In order to provide justification for the ensuing discussion, it is necessary here to go into a few technical details concerning what we already know about our model. The reader unfamiliar with the mathematics here will still be able to make sense of the rest of the proposal, though may have to take some of our claims on trust.

Our preferred construction of our model starts from a concretely constructed untyped λ-*algebra* of strategies. This λ-algebra exists in two versions: a *linear* λ-algebra L and an *intuitionistic* one I, the latter being obtained from the former by means of a "!" modality. By means of a simple categorical construction which adds splittings for projections in these algebras, we obtain two rich categories of games: a symmetric monoidal closed category $\mathcal{K}(L)$ and a cartesian closed category $\mathcal{K}(I)$, related by a co-Kleisli adjunction in the usual way.[2] The fact that our categories are constructed from untyped λ-algebras in this way is significant, as the existence of easily understood *universal types* in these categories gives us a good handle on many aspects of their structure and allows us to prove several results very easily (this approach is discussed in detail in [**?**]).

[2] Our $\mathcal{K}(L)$ is essentially Lamarche's category of sequential games, though our ! modality is richer than his, and in some sense combines the power of Lamarche's "backtracking exponential" and Abramsky's "copying" one. Our $\mathcal{K}(I)$ appears to be very close to one of the categories of games constructed e.g. in [**?**] (namely the one with no innocence or well-bracketing constraints on strategies), though our $\mathcal{K}(L)$ is quite different from their corresponding linear category.

Intuitively, one can think of the linear category $\mathcal{K}(L)$ as a world for modelling potentially *mutable*, non-copyable data such as (non-cloneable) objects: a computation can only "use" the value of an object once, since if the state of the object changes it is effectively replaced by the new value, and it is no longer possible to get at the old value. (This connection between computing with state and the ideas of linear logic has been observed before, but the connection seems to be more precise in our model than in previous work.) The intuitionistic category $\mathcal{K}(I)$, on the other hand, can be thought of as a world of *non-mutable* (or else copyable) data, such as values of type `int` or functional programs, together with programs involving certain *local* uses of exceptions and non-persistent state. As a first approximation, one can think of $\mathcal{K}(L)$ as "Java world" and $\mathcal{K}(I)$ as "ML world" — indeed, it will be shown in [**?**] how these categories can respectively be used to model Java-like and ML-like languages.[3] [4] Moreover, thanks to the existence of universal types, one can easily arrange that the languages precisely match the model in the sense that all (effective) strategies are expressible by a program of the language.

Our basic idea is to design a programming language combining Java-like and ML-like features, in which a fundamental distinction is enforced between *mutable* and *non-mutable types*. The design of the language will be inspired by the structure of our models, with the two kinds of types corresponding to objects in $\mathcal{K}(L)$ and $\mathcal{K}(I)$ respectively. Both of these categories have a rich mathematical structure, including higher-order types, and this will be used to suggest a type system and term calculus for the programming language itself. The presence of a natural notion of *subtyping* in our models will allow us to incorporate a familiar notion of inheritance in the language, and perhaps subtypes of other kinds as well. Moreover, the functors between $\mathcal{K}(L)$ and $\mathcal{K}(I)$ will allow for mutable and non-mutable data to interact in various ways. (Some particular features which we expect the language to include will be mentioned below.)

One further aspect of our models deserve mention, since it will vastly increase the scope of what we will be able to include in our language. Both our λ-algebras come equipped with a *universal projection*, which means the corresponding categories contain an object that plays the role of a "type of types". This means that they can model very strong type theories such as $F\omega$, and even logically inconsistent systems such as Martin-Löf's System U. In terms of the programming language, this not only allows for very strong polymorphism if we want it, but also opens up the possibility that many constructs for programming-in-the-large (such as ML-style *functors* acting on structures or on classes) might arise from from essentially the same mechanisms as those of the "core" language (indeed, for simplicity we might even choose not to make any syntactic distinction between "large" and "small" levels of the language). This will set us free to design and explore some potentially very powerful features, secure in the knowledge that type-safety will be assured by our adherence to the semantic model. It is in this area that we expect to make the most innovative and far-reaching contributions to programming language design.

Besides some consolidation of the theoretical ideas outlined above, one task in this project will be to investigate more closely what kinds of type constructors are available in our models, and hence what types we can put into our programming language. For instance, it seems that in $\mathcal{K}(L)$ there are two kinds of "product type" — one which can be used when the two components are truly independent (*i.e.* do not share state), and one which allows for the possibility of interference between the components. Several type systems have recently been proposed for keeping track of of which parts of the heap are independent in this sense. This will also be an important concern for us since we are designing a language with verification in mind. We will therefore start by investigating to what extent our categories provide models for these type systems.

II. A prototype language definition

Having investigated what is possible in terms of programming language design, the next stage will be to crystallize a prototype definition for our programming language. Our purpose is to come up with a provisional definition which we can release in order to elicit feedback from the community, and which could form the basis of a more definitive version at a later date. The particular priorities we shall have in mind are as follows:

- Our language should admit a denotational interpretation in our semantic model. Indeed, our denotational semantics will serve as the official definition of the language. We will resist the temptation to add any features that we cannot interpret in the model. This is crucial to our approach to program verification, since it will ensure that we retain a reasonable logical grasp of how programs will behave and how the various parts of the language interact.
- Our language should be clean and beautiful. This may seem a rather subjective criterion, but a more objective test would be whether it is possible to state simple and useful properties that are universally applicable to all programs in the language. Examples might include type safety properties, a "no dangling references" property as in ML, and a theorem asserting that data of non-mutable type is indeed non-mutable. In our view, cleanness and beauty are not merely a matter of aesthetics but are of real practical benefit for the avoidance of needless complexity in the understanding and (especially) verification of program behaviour. As mentioned earlier, this ideal will be achieved

[3] In particular, we are able to interpret class definitions so that inheritance and dynamic binding are handled correctly, though we will not give the details here.

[4] Of the standard features of OO languages, it seems that only *pointer equality* poses problems for our models. There are various slightly messy ways in which we can circumvent this problem if we are unable to find an elegant solution.

partly by the fact that we are designing our language around a mathematically elegant structure, but we will also pay close attention to the simplicity of each construct and to keeping the number of primitive concepts to a minimum.

- We will favour the use of very strong typing wherever possible. Types are a way of expressing certain properties of programs, and we strongly believe that the richer the type system (i.e. the more program properties it encapsulates), the easier the task of program verification will then become. Of course, a richer type system will typically also mean that more type annotations are necessary in programs; however, since our emphasis is on trying to provide support for verifiable programs, we will be willing to accept some additional verbosity in the language. Our strategy will be to start with an explicitly typed version of the language, and then allow the programmer to omit certain information where we see opportunities for type inference.
- We will try to stay close to the syntax of existing languages such as Java, Standard ML and OCAML wherever possible. This will facilitate the learning of our language and the the task of porting existing programs from these languages.

The following list of features gives a rough indication of what we expect our language to contain. (However, we have already identified many ways in which some of these features could be merged, so we expect the eventual form of the language to be significantly smaller than this list might suggest.)

- A repertoire of non-mutable datatypes similar to those of functional ML, including inductive types and higher-order types. Certain localized uses of state and exceptions will also be permitted in the non-mutable fragment (the semantic model provides guidance as to what the appropriate syntactic constraints should be).
- The usual constructs of an Algol-like imperative language with block structures and lexical scoping.
- An object system with classes, interfaces, single inheritance and dynamic binding, similar to that of Java, but with a more consistently higher-order character than Java or even OCAML. For example, consider a method invocation expression `o.m(t)`, where `o` is a target object, `m` a method and `t` a parameter. In Java this expression as a whole is a value, but in our language we also expect `o.m`, `m(t)` and `m` to be first-class values of some higher type. Moreover, we expect to be able to enforce type distinctions between mutable and non-mutable values of these kinds: for instance, a method call `m(t)`, whilst it will typically be applied to a mutable object `o`, might *itself* be either mutable or non-mutable, in the sense that its behaviour may or may not depend on state other than the internal state of `o`.
- A powerful polymorphic type system, incorporating but going well beyond the system of generic types currently being added to Java. For instance, we can, if we wish, allow types to be passed as parameters as in System F; moreover, such parameters could be constrained to be subtypes of existing types as in Pierce's $F_{\leq}$.
- ML-style pattern matching in function and method declarations, perhaps augmented by disjunctive patterns and guarded patterns as in OCAML.
- An exception mechanism similar to that of Java, where each method or function is explicitly annotated with the set of exceptions it could in principle throw. This approach to exceptions fits in well with our semantic model.
- A module system somewhat akin to that of ML (including higher-order functors). It is clearly possible to use the class system as a module system to some extent, although there may be arguments in favour of a separate module system.

However, our language will not include threads or any form of concurrency, since our game models embody an inherently sequential notion of computation. We will probably also not attempt to include *dependent types*, even though our model does support them and they would certainly be in accord with our objectives, since this would probably introduce considerable additional complication with little novel contribution on our part.[5]

For any critical design decisions we encounter, we will consult with colleagues in LFCS who are specialists in type theory, language design, language implementation and software engineering. In terms of concrete deliverables, this phase of the project will result in a formal syntax for our language, a complete set of typing rules, and a denotational semantics specifying the dynamic behaviour of programs.

III. Operational and axiomatic semantics

Since our language is to be inspired by a particular semantics model, it is natural to opt for a definition based on denotational semantics. However, in order to give some insight into how an implementation might work, we will also provide an equivalent operational semantics, and a justification of the equivalence.[6]

In order to support verification of programs in our language, it is desirable to provide a program logic or axiomatic semantics. We believe that it will not be too difficult to do this based on the denotational semantics, using the ideas

[5] Dependently typed programming languages constitute a major research area in themselves — see e.g. [**?**].

[6] We believe that standard techniques will suffice for this. Since the language will be fairly large, we may content ourselves with a proof outline concentrating on the interesting cases rather than a complete rigorous proof.

and methods that we are currently applying to program logics for fragments of Standard ML. We will spend some time investigating any particular challenges posed by our language in order to assess the feasibility of this approach, but we do not expect to develop a complete program logic in full detail.

IV. Animation and prototype implementation

An interesting aspect of game semantics is that, in contrast to many other kinds of denotational semantics, it is *executable*: the denotation of a program will be a strategy of some kind which can be "played" or executed on a machine. We have already developed an "implementation" in ML[7] of our λ-algebras L and I, and shown how this can be used as a workbench for game semantics in the context of ML programs. This program, known as Stratagem, is available from the author's web page along with examples illustrating its use.

The denotational semantics for our language will essentially define a function from programs to strategies, with one inductive clause per syntactic construct of the language. We can therefore directly recast such a definition as an ML function from abstract syntax trees to executable strategies, with one ML clause per syntactic construct, and thus obtain, almost for free, a prototype implementation or "animation" of the language. This implementation will not be particularly efficient, but at least it will allow us (and others) to get a feel for the language and experiment with the effects of various choices, at the cost of very little implementation effort. It will also enable us to explore possible programming idioms that make use of the novel features we have introduced into the language.

We will devote a part of the project to developing and experimenting with an animation of the language in this way. This work will also have spin-offs for of other uses of the Stratagem system (e.g. execution tracing for ML).

In the longer term, a natural goal would be to build a compiler for our language, compiling down to some intermediate language such as JVM. We do not expect to have time to complete even a prototype compiler within the current project, but we hope to demonstrate the feasibility of such an undertaking by writing small-scale compilers for particular subsets of the language that raise interesting implementation issues.

One important question is whether we allow the design of our language to be constrained by any particular target intermediate language we have in mind. Experience within the Mobile Resource Guarantees project at Edinburgh [**?**] (and also the MLj project [**?**]) has shown that certain advanced language features such as ML functors do not admit compilation to JVM in a natural way, and other intermediate languages in current use are likely to pose similar problems. There is a tension here: clearly if we can compile to JVM then a great deal of existing infrastructure will be available to us (e.g. programs in our language can probably make use of existing Java APIs), and this would encourage the use of our language and promote wider dissemination of our ideas. On the other hand, the spirit of the project is to explore ideas in language design for the long-term future, and also to let the design of our language be driven by the mathematics rather than by more accidental pragmatic concerns. In this project we will opt for the purity and idealism of the latter approach: we will not insist that our language be naturally compilable to any existing intermediate language (though a substantial subset of it probably will be). We will, however, devote some time to a preliminary investigation of what needs to be done in order to support efficient implementations of our language.

Manpower and work plan

All parts of the project will be supervised by Longley, who will contribute an average of five hours a week to the project. It is expected that he will contribute actively to part I of the project in particular. The core of the project, consisting of parts I, II and III, will be undertaken by one PhD student, working full-time for three years. The student will also devote some time to part IV, but we will employ a research assistant for 1 year to bear the brunt of the implementation effort in this part of the project. (This could be done by a recent graduate without research experience but with some knowledge of formal semantics from their undergraduate course.) [ASK MONIKA ABOUT ALL THIS.]

Details of the timescale for the various phases of the project are given in the accompanying Diagrammatic Workplan.

C. Relevance to beneficiaries

We expect our work to have an impact in three main areas:

Language design. Even if we proceed no further with the development of our language, our work will have contributed some innovative ideas on programming language design, particularly regarding higher-order and polymorphic extensions of object-oriented features, and the enforcement of type distinctions between non-mutable and (potentially) mutable data. History shows that ideas in language design developed within more theoretical communities can and do find their way into mainstream computing practice — witness the influence of ML on the design of Java (e.g. the exceptions mechanism), or the current incorporation into Java of the system of generic types proposed by Abadi and Wadler. In the

[7] Our implementation requires ML of New Jersey since it makes essential use of continuations.

long term, such developments in language design (particularly those relating to type systems) make for a cleaner, more disciplined approach to programming, and hence ultimately for more robust, reliable software. Furthermore, strong type systems make more information available to a compiler and increase the opportunities for program optimization; we would certainly expect our mutable and non-mutable types to be of value here.

Program verification. The problem of rendering feasible the verification of realistic programs remains a huge challenge, and in our view one important step towards achieving it is to start with a mathematically clean language that lends itself to program verification. Recent progress in semantics has shown that very substantial "safe fragments" of existing languages are suitable for this purpose, but this is somewhat satisfactory as there remains a worry about how safe programs will behave in interaction with unsafe ones. Our project will, perhaps for the first time, demonstrate the viability of a realistic programming language based entirely on well-understood semantic principles; we will also go some way to providing a program logic for the language. We therefore believe our language will offer a suitable framework for further research in program verification. In terms of research impact, most of this benefit will be realized even if we do not develop a full-scale implementation (much valuable research in this area has been based on theoretical languages that no one has implemented). However, we also hope that, if implemented, our language might actually become the medium for substantial formal software development projects in the future.

Programming applications. Finally, if we eventually decide to proceed with a more serious implementation of our language, we are hopeful that it will prove attractive to many programmers as a state-of-the-art language based on theoretically sound principles, much as ML has been for the past couple of decades. (Indeed, our ideas have been influenced to some extent by our own views of the sort of language we would most like to use, based on our programming experience in Java and ML.) Like ML, our language will be well suited to fast prototyping of research applications, and could enjoy some level of use in teaching. Furthermore, we expect that certain familiar programming tasks will become simpler in our language: for instance, certain common programming idioms in Java such as factory, listener and visitor patterns could be more naturally expressed using higher-order constructs.

D. Dissemination and exploitation

As usual in our research community, the main modes of dissemination will be refereed journal papers, conferences, informal workshops personal visits to other research establishments, and the publication of other materials via the Internet. In particular, the definition of our language, the animations and prototype implementations we shall produce, and any programs of particular interest that we develop, will be made available at an early stage to interested colleagues via the World Wide Web. The definition of the language will be submitted for publication to a refereed journal once we deem it to have reached a sufficiently stable form. A unified presentation of all the results of the project will appear in the student's PhD thesis.

At the end of the project we will take a decision on whether to proceed with the development of a usable implementation and a commitment to supporting it. The decision will be based on feedback from colleagues and our own views concerning the level of success of the project. If we do develop such an implementation, the opportunities for dissemination of our ideas will be taken to a new level. Our implementation would be made available under a free software licence, and we would offer general support to encourage its use in research and teaching.

E. Justification of resources

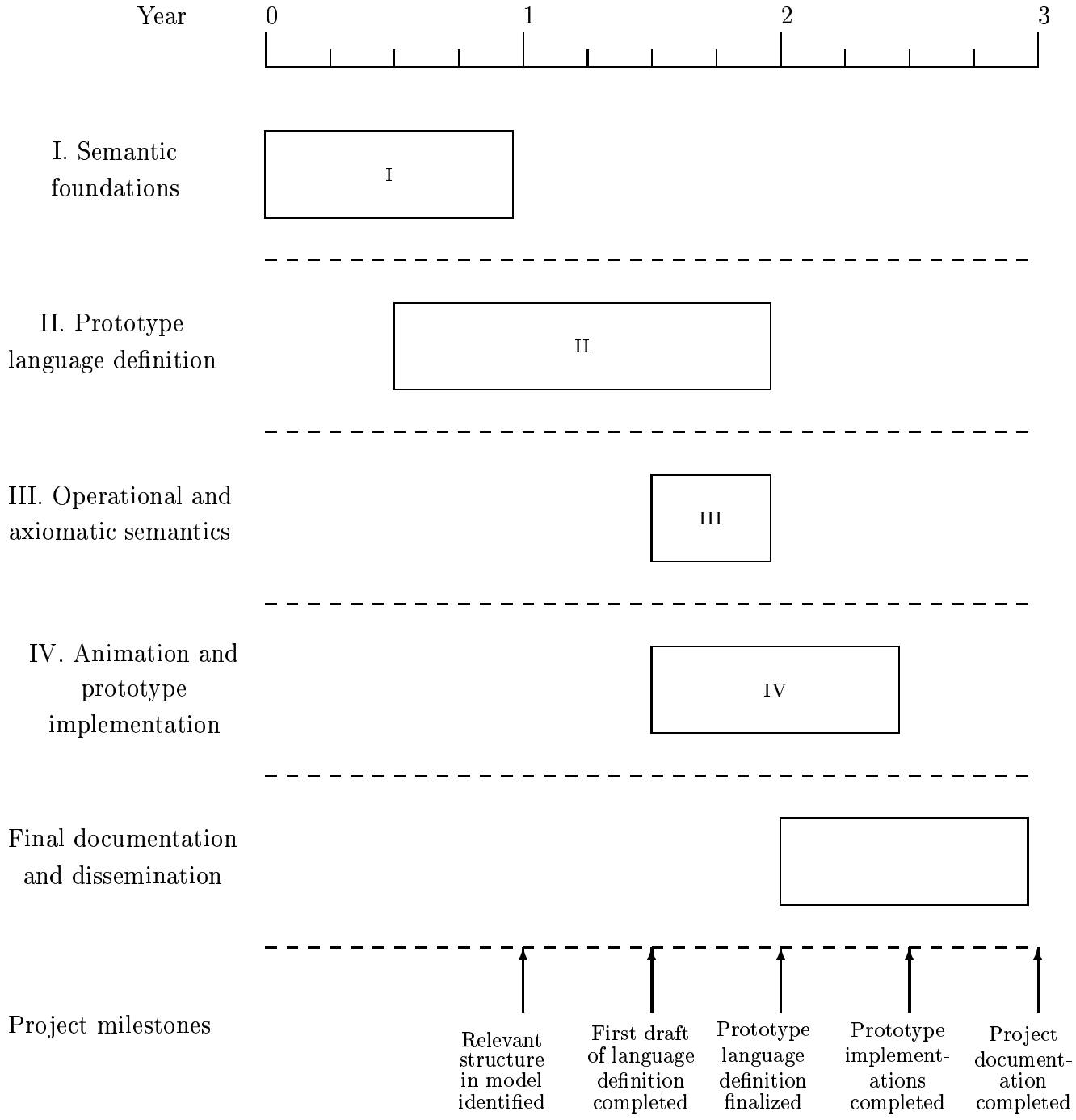
Manpower. We request funding for one PhD student for three years, in order to work full-time the project as described in section B above. We also request funding for a research associate on the AR1A scale for one year, in order to work on the implementation aspects of the project in the later stages, as described in part IV of section B. In addition, we seek funding for 10% of a computing officer (AD3.3) to provide infrastructure support for development and use of our prototype implementations, 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. We request travel and subsistence support for five one-week visits to European centres (e.g. Aarhus, INRIA), and one 1-2 week visits to a number of centres within the US (e.g. CMU, SRI/Stanford). 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).

Equipment. A workstation, maintained over the project period, is requested for the use of the PhD student, to support work on prototype implementations of our language, experiments with programs in our language, the production of documents reporting the results of our research, and interaction with colleagues via email and the Web. A second workstation, maintained over one year, is requested for the use of the research associate, to support the intensive implementation work involved in part IV of the project. We also request an appropriate contribution to consumables, shared networking, and fileserver provision.

A programming language based on game semantics

Diagrammatic Workplan



Remarks:

- The task numbers (I,II,III,IV) refer to the parts of the project described in Section B of the main proposal.
- Longley will supervise all areas of the project, and will contribute an average of five hours a week.
- The PhD student will work on parts I,II,III of the project, and will produce the documentation for these parts. He will also have some input to part IV.
- The research assistant will do the bulk of the implementation work in part IV of the project, and will produce the documentation for this part.