ACM Workshop on Strategic Directions in Computing Research, MIT, June 14–15, 1996.

Strategic Directions for Research on Programming Languages

Chris Hankin
Imperial College, London

Hanne Riis Nielson
University of Aarhus

Jens Palsberg
Purdue University

Abstract

Programming language research covers a wide range of principles spanning a spectrum from pure to applied and employing methods and techniques ranging from theory to systems. In this working group we have concentrated on five central aspects of this: semantics, type systems, program analysis, program transformation, and implementation; other aspects of programming language research are covered by other groups. This report summarizes the open problems and challenges uncovered by the group.

1 Introduction

"A good programming language is a conceptual universe for thinking about programming"

Alan Perlis
NATO Conference on Software Engineering Techniques
Rome, 1969

Programming language research covers a broad spectrum from systems work through to theory. On the theoretical side programming language research has its roots in the work of Schönfinkel, Curry, Kleene, and Church in the 1920s and 1930s. On the practical side, it has been influenced by the developments in hardware spanning from the von Neuman architecture of the 1950s to the heterogeneous distributed networks of today. Also developments in software engineering with its requirements of feasibility, reliability and performance have had great impact on the area. A feature of programming language research that gives it much of its vibrancy is that individual researchers have the opportunity to work across broad bands of this spectrum.

The area is mainly concerned with the discovery of principles and generic techniques; it is not, with a few notable exceptions, about the invention of new languages. Many of the results of the area are generally applicable to a wide variety of language paradigms including procedural languages, functional and

logic languages, object-oriented languages etc. Many common programming language features have emerged from this area; some of the more well-known examples are:

- *Procedures*: inspired by the notion of λ-abstraction and pervasive in modern programming languages.
- *Recursion*: primitive recursion (for loops) and general recursion (while loops) were introduced in Recursion Theory but are pervasive in modern high-level languages.
- *Types*: developed in the context of pure calculi such as the λ-calculus and combinatory logic but extended by programming language researchers and now pervasive.
- *Exceptions*: introduced in PL/1 but extensively studied and formalised in ML [15]; similar concepts have later appeared in C++.
- *Monitors*: introduced by Hoare in the 1970s [12] and now forming the basis for the concurrency features of Java.

In addition to language features, programming language research has also produced important techniques. Let us mention the five areas that will receive special attention in this report:

- *Semantics*: Formal semantics is concerned with the description of program meanings by operational, denotational or axiomatic specifications. [16, 17]. It improves our understanding of new as well as well-known programming constructs and it provides a yardstick for implementation and a foundation for analysis and verification techniques and program transformation. Over the years different techniques have been developed to handle the different programming paradigms and the different applications.
- *Type systems*: A type is a collection of values which share a common structure, operations and other properties. A type system is a specification of how types are assigned to values [4]. Type safety – the prevention of certain classes of programming errors – is a desirable property of any programming language; many of the recently published standards for safety critical software insist upon the use of strongly typed programming languages. Over the years different type systems have been developed and found their ways into commercially successful languages.
- *Program analysis*: Program analysis is concerned with the problem of statically predicting properties of the dynamic executions of programs [1, 8, 14]. Traditionally, program analysis has been used extensively to enable various optimizations and transformations in compilers; among the newer applications is the validation of software to reduce the likelihood of malicious behaviour. Over the years a wide variety of techniques have been developed to handle the different analysis problems and different programming paradigms.

- *Program transformation*: The goal of program transformation is to modify some representation of the program to change some of its properties whilst preserving others. For example, most useful program transformations preserve the input/output semantics of the program but might radically change the program's complexity. The transformation of programs is an important technique in the development of reliable and efficient software [3, 13]. Techniques have been developed for different stages of the software development process: when developing programs from specifications, when specialising existing programs to specific contexts and, in particular, in optimizing compilers.

- *Implementation*: This area concerns compilation [1] and run-time support. Current concerns in compiler technology include compilation for distributed systems, optimisations across module boundaries and correctness issues. Memory management, particularly on complex cache architectures, is a critical concern in run-time support. This area is increasingly concerned with the development of generic tools rather than tailored solutions.

Programming language research in the 1990s is slowly changing from what it was in the 1970s and 1980s: the traditional uni-processor machines are being replaced by heterogeneous and physically distributed computer networks, the traditional line based user interfaces are being replaced by graphical user interfaces, multi-media applications are the rule rather than the exception, etc. There is a growing need for programming languages to adapt to this fundamental change and thereby for programming language research to address these problems more thoroughly: We need to understand the semantics of the language features developed for programming these systems; we need to understand what type security means in this setting; we need to develop program analyses that statically predict the behaviour of these systems; and we need to develop implementation techniques that utilise the vast amount of parallelism present in these networks.

We believe that the history tells us that programming language research is in good shape to take up this challenge.

The views of programming language research expressed in this report are inevitably coloured by the constitution of the working group; there were other working groups covering similar topics, notably Object-Oriented Programming, Software Engineering and Programming Languages, and Human Computer Interaction. A recent publication which contains more detailed discussion of some of the research themes identified below is [10].

2 Strategic Directions

This section presents five strategic directions identified by the working group. Our aim was to identify some general themes which will provide the impetus for new research over the next few years. Progress will be achieved by successful results in a number of technical areas (some of) which are identified in

the next section. The strategic directions represent a mix of continuing, long-term research going on within the field, and more recent directions sparked by changes in the larger world of computing. In the past, the programming of single computers and computers in local-area networks has been the dominant programming task. In the coming years, programs will be needed for global computing, domain-specific computing, embedded systems, large-scale systems, and more. These application areas almost certainly require new programming concepts. For example the heterogeneous, distributed and dynamic nature of global computing networks raise issues to do with configuration, coordination, security, and exploitation of interprocessor parallelism. In the context of embedded systems, performance predictability and fault tolerance pose new challenges.

Each of the five strategic directions is addressed by many of the theme areas in programming language research which we discuss below. The alphanumeric codes identify relevant research topics in these themes, which are described more fully in the next section.

1. *Distributed Computing*: One of the recurrent themes throughout the position statements is distributed computing. This poses new challenges in all areas of programming language research: how do we design languages for such systems that have firm semantic foundations, support the safe programming of distributed systems and utilise the vast amount of parallelism offered by the networks. Persistence and support for transactions are examples of areas of increasing importance. A common feature of distributed systems, which is also of independent interest, is mixed language working – "integration of programming paradigms" is an important theme. A number of the topics described in the next section contribute to this direction: S-1, S-3, S-4, T-4, PA-4, PT-4, I-1, and I-4.

2. *Incrementality, Modularity and Abstraction*: Software systems are long-lived and must survive many modifications in order to prove useful over their intended life span. The primary linguistic mechanisms for managing complexity are modularity (separating a system into parts) and abstraction (hiding details that are only relevant to the internal structure of each part). A challenge for future language design is to support modularity and abstraction in a manner that allows incremental changes to be made as easily as possible. Object-oriented concepts have much to offer and are the topic of much on-going investigation. Topics contributing to this direction include: S-3, T-1, PA-2, PT-2, I-2, and I-5.

3. *Generic Formalisms, Integration of Tools and Techniques*: Many of the formalisms that we work with have been developed in the context of specific problem domains; there is the need to develop these further to provide the basis for generic formalisms. At the more systems-oriented end of the spectrum, there is the need to integrate the tools and techniques that are being produced by individual research activities into "real" systems. Part

of the work in this direction also involves developing a better understanding of the relationship between different approaches. We have identified the following topics as contributing: S-2, S-3, T-2, T-3, PA-1, PA-2, PT-1, PT-3, and I-5.

4. *Correctness, Efficiency, Engineering, and Pragmatics*: The goal of programming language research must be to define languages and techniques which improve the quality of software. The notion of *quality* is multifaceted but must include programmer productivity, verifiability, reliability, maintainability, and efficiency. This theme has assumed greater importance following a number of well-publicised disasters, such as the Pentium chip and Ariane-5. A number of topics address this direction: T-3, T-4, PA-3, PT-2, PT-3, I-3, and I-4.

5. *Education and Technology Transfer*: Although only few of the topics mention this aspect, the validity of all programming language research hinges on our ability to educate others and transfer our technology into practical systems. For example, we can consider TCL to be a failure of education and technology transfer because the language does not even have a syntax suitable for presentation by a grammar, and Java a success, since the industrial group responsible for its design used research developments of the past decade to great advantage. The following topics explicitly address this direction: PT-1, and PT-2.

3 Research Areas

Programming language research is built on a bedrock of principles, ranging widely along the pure–applied and theory–systems spectra. In this working group we have chosen to focus on five aspects that play a central rôle in programming language research:

1. *Semantics*
2. *Type systems*
3. *Program analysis*
4. *Program transformation*
5. *Implementation*

None of these subareas can exist without the others, and indeed, the various subareas rely upon each other. For instance, program analysis has a strong theoretical basis: we trade precision for computability and thus cannot expect precise answers to our questions, but it is important for the practical applications that the answers are semantically sound and feasible to implement. Program transformation likewise has strong theoretical foundations and is often enabled by program analysis: it is crucial that the transformations preserve the semantics of the programs and transformations are often guided by the

results of program analyses. Type systems ensure that the execution of well-typed programs do not lead to certain run-time errors. Implementations are required to be faithfull to the semantics and rely on types, program analysis and transformation.

The various subareas are useful in identifying the future topics for programming language research. The following topics were abstracted from the position statements prepared by the participants of the working group.

3.1 Semantics

S-1: Language Design.

One example of an area in which there is current interest in language design is coordination languages [5, 6]. Coordination languages have been developed for programming systems in which multiple, heterogeneous agents cooperate on the execution of a particular task. The recent emergence of this class of languages has coincided with the emergence of the WWW; the latter provides a fertile area for applications. As yet, there is no clear consensus about a suitable set of language primitives for coordination. Like in other areas of language design, there is the, often unrealised, expectation that studies in semantics will lead to improved language designs.

S-2: Foundations.

Work on programming language semantics have produced a number of different formalisms. Operational semantics provide an abstract implementation-oriented account of program meaning, denotational semantics give a more abstract mathematical account and axiomatic semantics focus on partial correctness issues (see [16, 17] for a thorough discussion). In Operational Semantics there is the choice between big-step Natural Semantics or small-step Structural Operational Semantics. In Denotational Semantics there is a choice about which mathematical universe is used for program meanings (for example, metric spaces or domains). These choices are sometimes a matter of taste rather than because of deep foundational differences. It is important that effort is devoted to understanding the capabilities and limitations of the different approaches so that informed choices can be made.

S-3: Paradigm Integration.

Mixed paradigm languages are attractive because they allow the programmer to exploit the different strengths of the paradigms in addressing different aspects of a problem. A problem that arises, particularly if the mixing is *ad hoc*, is that there can be unwelcome interactions between the component sub-languages. Examples of successful integrations include functional and logic languages [11] and functional languages with imperative features [17]. The emergence of concurrent and distributed applications raises new issues and opportunities for successful integrations.

S-4: New Directions.

The last few years have seen the development of a number of new areas of computing which pose interesting challenges for semantics. These include Artificial Neural Networks, Embedded Systems, Graphical User Interfaces, Global

Computing. Some of these may require essentially new approaches and theoretical tools. For example, some of the issues that have to be addressed in Global Computing include: type systems, security, reliability, modularity, resource management. Another challenge is the "executable content in messages" which is supported to a limited degree by Java. There has been very little work so far on semantics-related issues that arise in these new areas.

3.2 Type Systems

T-1: Types for Objects.

Many type systems for object-oriented languages have been defined [4, 9]. Their relative merits should be investigated, aiming at building consensus on what features a type system should include, what the proper role of type inference is, and how these ideas can be integrated with module systems. Specific issues include the integration of subtyping and polymorphism, the current lack of principal types in implicit types systems with subtyping, and the typing of binary methods.

T-2: Type-based Compilation.

Types can be used by an optimizing compiler to make programs run faster and consume less space. Known ideas on typed intermediate languages, typed pointer data structures, etc., should be extended and applied to the compilation of both conventional and modern languages. It should be further investigated how the type-based program analyses compare with more traditional program analyses, and how we can integrate type-based analysis and compilation. Specific issues include side effects, exceptions, and modules.

T-3: Type-aware Programming Environments.

The editors, version managers, etc., for typed languages can be improved by making them aware of the type system. During programming, type information might be computed incrementally and used, for example, to warn about errors and to help the programmer understand the program. The type system itself might be made changeable by the programmer, leading to customizable tools.

T-4: New Applications.

The safety and security issues inherent in distributed programming [2] may be partially solved using type systems. Known ideas on structural type matching, type systems for concurrency, and types for security should be extended and complemented. On a different front, there is a continuum between type systems and program analysis. Refined type systems are increasingly being used to capture program properties that have been the preserve of program analysis. One example is the property of deadlock-freedom in process calculi. Such type systems are usually undecidable, so we have to work with approximate types. This is an area that will be of growing importance.

3.3 Program Analysis

PA-1: Unification of Different Techniques.

Program analyses have been developed for a variety of programming languages (e.g. imperative, functional, logic, object-oriented) using different techniques (e.g. flow based, semantics based, inference based, and abstract interpretation based) and with different semantic foundations (e.g. flow charts, operational semantics, denotational semantics). The techniques have been developed and further specialised by different sub-communities that, to some extent have continued in different directions. There is a critical need to understand the relative merits of the different approaches and to what extent they improve each other.

PA-2: Realisation of Program Analyses.

Program analyses have been implemented in a number of optimizing compilers but the re-usability of these implementations seem to be rather limited. In particular, the automatic generation of program analysers is still lagging far behind the technology developed for front-ends (parsers) and back-ends (code generators) [1]. Although the theoretical foundations of a particular technique may seem well-understood there is still a long way to a generally useful analysis tool.

PA-3: Choice of Program Analyses.

Different analyses may claim to solve the same problem but may do it in different ways (even when using the same underlying framework) and may yield different precision. Sometimes a coarse analysis is acceptable if we get the answer fast and at other times we need the precision that a slower analysis often will give. We need criteria for selecting an analysis with the "right" balance between precision and cost.

PA-4: New Applications.

Traditionally program analysis has been applied in compiler construction but the technology has much more to offer. It is largely unexplored how to analyse real-time systems and distributed systems. This is relevant for safe systems on heterogeneous networks with thousands of workstations.

3.4 Program Transformation

PT-1: Automatic Tools.

The development of programs is to a large extent manual work. The development of semantics-based program manipulation tools offers the prospect of changing that: programs can be considered as data and thereby be automatically manipulated. Progress has been made in various areas such as partial evaluation and unfold/fold transformation, and in the application of techniques such as program slicing and staging transformations. But we need to push the state-of-the-art so that semantics-based transformation tools can play a full rôle in the software development process.

PT-2: Algorithm Development and Design.

Program transformations have been used to integrate algorithm design and program development. This has already given rise to a better understanding of known algorithms and to the discovery of new algorithms [13]. A next step

would be to use program transformation technology to gradually develop efficient implementations using the best known data structures.

PT-3: Foundations.

Program transformations allow us to replace certain programs with others. In order for this to be sound it is crucial that the transformations preserve the semantics of the original programs. Moreover, we must have strategies to direct the application of the transformation rules to achieve goals such as efficiency improvement, greater degree of parallelism, better modularity, etc.

PT-4: New Applications.

Traditionally, program transformations have been used to improve the efficiency of programs to be run on a single machine. When programming complex parallel and distributed systems we may need tools for automatically mapping programs onto the architecture. Sophisticated transformations are then required to enhance the performance of the overall system; there may even be a need to apply such transformations dynamically thereby adapting the programs to a constantly changing environment.

3.5 Implementation

I-1: Compilation for Distributed Systems.

Programming languages with support for parallelism and distribution increase the demands for incremental compiler technology: occasionally decisions has to be taken dynamically about where to store data, when to move data from one storage area to another, on which machine to perform specific computations, etc. In general, the language support will be for coarse-grained parallelism, whereas the exploitation of fine-grained, instruction level parallelism is left to the compiler itself. Techniques have been developed for single processors but the adaption to heterogeneous processor architectures is a complex task.

I-2: Optimization.

Modularity is often associated with separate compilation, and separate compilation has traditionally been a barrier for aggressive optimizations, which often require global program analysis. Feasible global optimization might be achieved by compiling each module into a central persistent repository, and let an optimizer roam it whenever there are cycles to spare. The delaying of some optimization and code generation until link time and run time has shown promise and deserves further attention. Finally, it is important to get a better prediction of the cost/benefit of various optimizations.

I-3: Compiler Correctness.

While techniques for proving the correctness of non-optimizing compilers basically are well understood, it is not so clear how to prove the correctness of realistic optimizing compilers. Ideally one would want to prove each of the optimizations correct and then combine the results to a correctness theorem for the complete compiler. However, this is complicated by the fact that the optimizations often introduce new constructs in the intermediate language and this have to be reflected in the correctness predicates.

I-4: Memory Management.

New garbage collection techniques, interacting with representation optimizations can improve locality and latency properties of automatic memory management. Also, write-allocate caches and fast block moves would speed heap allocators and garbage collectors, but these are rare in current hardware. As garbage collection becomes customary, one can hope that memory system design will be adapted to optimize garbage collection. The increasing ratio of processor speed to memory speed means that cache performance will dominate all other back-end factors; for complex cache architectures it is far from clear how to achieve good cache performance.

I-5: Generic Tools.

Tools for developing compiler front-ends (scanners and parsers) are well-known and widely used. The remaining part of the compiler has not been equally successfully automated: tools based on attribute grammars have been for semantic analysis and only very recently tools for the optimization phase have been developed. For the back-end very recent tools for code selection based on pattern matching seem promising and tools based on graph coloring have been successful for register allocation. Further development and extension of these tools to languages and systems running on heterogeneous networks is needed.

Appendix: Group Members

Our working group on programming languages was part of the Workshop on Strategic Directions in Computing Research held at MIT, June 14–15, 1996 to celebrate the 50th anniversary of the ACM. This appendix lists the members of our group. Most of the members prepared a position statement before the meeting; these statements provide background information for the issues mentioned in this report.

The following people attended the discussions at the workshop: Alex Aiken (UC Berkeley), Kim Bruce (Williams College), Luca Cardelli (DEC SRC), Paolo Ciancarini (University of Bologna), William Clinger (Northeastern University), Charles Consel (University of Rennes), Patrick Cousot (École Normale Supérieure), Robert Glück (DIKU), Chris Hankin (Imperial College, London – Co-chair), Robert Harper (CMU), Suresh Jagannathan (NEC Research Institute), Simon Peyton Jones (University of Glasgow), Peter Lee (CMU), David MacQueen (Bell Laboratories), José Meseguer (SRI), Daniel Le Métayer (Irisa/Inria Campus de Beaulieu), Albert Meyer (MIT), John Mitchell (Stanford University), Ugo Montanari (University of Pisa), Flemming Nielson (University of Aarhus), Hanne Riis Nielson (University of Aarhus – Co-chair), Martin Odersky (University of Karlsruhe), Bob Paige (NYU), Jens Palsberg (MIT – Co-chair), Alberto Pettorossi (University of Roma II), Uday Reddy (University of Illinois at Urbana-Champaign), Tom Reps (University of Wisconsin), John Reynolds (CMU), Jon Riecke (Bell Laboratories), Barbara Ryder (Rutgers University), David Schmidt (Kansas State University), Olin Shivers (MIT), Scott Smith (Johns Hopkins University), Dennis Volpano (Naval Postgraduate

School, California), Reinhard Wilhelm (University of Saarland).

The following submitted position statements but were unable to attend the meeting: Jaco de Bakker (CWI), John Darlington (Imperial College, London), Susan Graham (UC Berkeley), Michael Hanus (RWTH Aachen).

References

[1] Aho A. V., Sethi R. and Ullman J. D., *Compilers: Principles, Techniques and Tools*, Addison Wesley, 1986.

[2] Bank J., *Java security*, available via http://www-swiss.ai.mit.edu/ jbank/javapaper/javapaper.html, 1995.

[3] Burstall R. and Darlington J., *A transformation system for developing recursive programs*, Journal of the ACM, 24(1), 44-67, 1977.

[4] Cardelli L. and Wegner P., *On understanding types, data abstraction and polymorphism*, ACM Computing Surveys, 17(4), 471-522, 1985.

[5] Carriero N. and Gelernter D., *Coordination languages and their significance*, Communications of the ACM, 35(2), 97-107, 1992.

[6] Ciancarini P. and Hankin C. (eds), *Coordination Languages and Models*, LNCS 1061, Springer Verlag, 1996.

[7] Cohen J., *Garbage collection of linked data structures*, ACM Computing Surveys, 13(3), 341-367, 1981.

[8] Cousot P. and Cousot R., *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL), ACM Press, 238-252, 1977.

[9] Gunter C. and Mitchell J., *Theoretical Aspects of Object-Oriented Programming*, MIT Press, 1994.

[10] Hankin C. and Nielson H. R. (eds), *Symposium on Models of Programming Languages and Computation*, ACM Computing Surveys, 28(2), 293-359, 1996.

[11] Hanus M., *The integration of functions into logic programming: ¿From theory to practice*, Journal of Logic Programming, 19 and 20, 583-628, 1994.

[12] Hoare C. A. R., *Monitors: an operating system structuring concept*, Communications of the ACM, 17(10), 549–557.

[13] Jones N. D., Gomard C. K. and Sestoft P., *Partial Evaluation and Automatic Program Generation*, Prentice-Hall, 1993.

[14] Jones N. D. and Nielson F., *Abstract interpretation: A semantic-based tool for program analysis*, in volume 4 of *Handbook of Logic in Computer Science*, Abramsky S., Gabbay D. and Maibaum T. (eds), Clarendon Press, 1995.

[15] Milner R., Tofte M. and Harper R., *The definition of Standard ML*, MIT Press, 1990.

[16] Nielson F. and Nielson H. R., *Semantics with Applications, a Formal Introduction*, Wiley, 1992.

[17] Tennent R. D., *Semantics of Programming Languages*, Prentice-Hall International, 1991.