Research on Semantically Based Program-Design Environments: The Ergo Project in 1988

Peter Lee, Frank Pfenning, John Reynolds, Gene Rollins, and Dana Scott

March 1988
CMU-CS-88-118

Department of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

Abstract

This report describes the current state of the Ergo Project starting with a discussion of background to the research area, presenting a review of past activities, and then detailing thoughts and proposals for future work. Principally, the research of the Ergo Project is aimed at improving the ability of programmers to develop and maintain provably correct, adaptable, and efficient software. The report argues that an approach to software development is needed in which the results of the development process embody in a formal way the wealth of design information that is lost in current development practices. The project pursues a *Design-Based Approach* in which not only are the specifications of problems and the implementations of their solutions recorded, but also the design decisions that link them are concretely represented and manipulated. Specifically, we recommend a methodology of *inferential programming* and explain how work over the last three years has resulted in valuable general insights into the nature of program design. Future research comprises taking steps towards the creation of an integrated *Experimental Program-Design Environment* that provides specifically for the *retention*, *formalization*, and *reuse* of design information. Such an environment would support inferential programming as well as other design-based approaches to program development, and it will be adaptable to a wide range of languages, development styles and proof systems.

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

Contents

Semantically Based Program-Design Environments: The Ergo Project in 1988

1. Introduction.

The research of the Ergo Project is aimed at improving the ability of programmers to develop and maintain provably correct, adaptable, and efficient software. Historically, programs have had both to describe executions and to provide information to the programmers who develop, analyze, and maintain them. The goal of *good performance* during execution usually conflicts with the human goals in program development of establishing *clarity and structure*, allowing *adaptability and reuse*, and easing *analysis and verification*. Current programming practice forces an unfortunate trade-off between these two types of goals and does not allow one to represent in the resulting program most of the crucial design decisions made during development. What is needed is an approach to software development in which the results of the development process embody, in a formal way, the wealth of design information that is now lost. To this end, we are pursuing a *Design-Based Approach* (DBA) in which not only are the specifications of problems and the implementations of their solutions recorded, but also the design decisions that link them are concretely represented and manipulated.

Specifically, our research centers around exploration into *inferential programming*, which is our term for any DBA founded on formal methods and theoretical underpinnings (for the initial statement of our project goals see [SS83], and for earlier, relevant references see [CS77], [BD77], [BP77], [Tur82]). The Ergo Project's research in inferential programming over the last three years has resulted in valuable general insights into the nature of program design. For example, we have carried out, on paper, large formal derivations of several complex programs, in one case deriving an important extension of an algorithm in response to more general requirements (see [EP87] and the fuller description in the next section).

These insights and experiences have been critically important in guiding our development of a reasonably large body of theoretical knowledge and software support for language-generic program manipulation and analysis. We believe this has placed the Ergo Project in a strong position to make significant advances in the area of design-based approaches to software development. Hence, we are now taking first steps toward the creation of an integrated *Experimental Program-Design Environment* (EPDE) that provides specifically for the *retention*, *formalization*, and *reuse* of design information. This environment will support inferential programming, and will be adaptable to a wide range of languages, development styles and proof systems.

In this report we describe the current state of research in the Ergo Project. First, we review the progress of our past work. Then, we present our thoughts and proposals for our research strategy for the next three years.

2. Background and Accomplishments.

During the process of software development, a programmer makes many decisions. Conceptually, these decisions constitute a body of *design information* that describes how an implementation is related to the original specification or set of requirements. It should be clear that—represented in some form—this information is necessary for adaptation, analysis, verification, and other activities important for the software lifecycle. Unfortunately, descriptions of design decisions are not generally represented explicitly in programs, either as formal structures or informal comments, and thus are usually lost or forgotten.

An undesirable consequence of this loss of information is that programmers are forced to rediscover for themselves much design information whenever a program is to be analyzed and adapted. This is a principal cause of the high cost of maintaining and adapting software—especially when the programmers involved are not the original developers. Also, at the present time very little of the explicit information now associated with one program can be directly applied to the development of other programs that have similar requirements. One of the reasons for this is the lack of detailed structures that would make reuse possible. Finally, the absence of design information makes it inherently difficult (if not impossible) to verify that programs meet their specifications—again because there are no links between the relevant features of the specification and the final program. Just as with adaptation, *a posteriori* program verification requires the rediscovery of much of this lost design information. The primary need is to capture design information at the appropriate points *during* program development, and then to show how to use this information in an effective way.

The DBA meets this need, and thus has been the focus of research for many existing groups (*e.g.*, [Bjo87], [BP77], [BD77], [CS77], [Dar78], [EP87], [Fea87], [Hog81], [Mas86], [Mol84], [SJW86], [Sch86], [Tur82], [Wil83]). In what follows, we describe more fully the DBA, outline the potential benefits of this research, and describe more fully the experience gained so far in the Ergo Project. Then, in Section 3 we delineate the specific proposed research for this project, which is centered around exploration into inferential programming.

2.1. The design-based approach.

Program development always involves making two transitions: first, from an *informal* description of requirements to a *formal* specification of *what* the problem is, and second, from a specification to an implementation, detailing *how* to solve the problem. These transitions are conceptually distinct, even if they are not distinct in process. Both transitions are made even if a specification is not explicitly expressed. Although both transitions are problematic, our research effort is concentrated on the what-to-how transition. Thus, when we say "design" we refer to the development of implementations that are consistent with specifications.

How do implementations differ from specifications? First, they differ in their level of *concreteness*; specifications can be more abstract, since they can avoid giving an algorithm of *how*

to solve a problem. But, secondly, they also differ in their level of *complexity*; implementations carry interdependencies among their parts resulting from optimizations made for efficiency gains. Modularity and structure interfere with good performance. Specifications can have more comprehensible structure (*i.e.*, less interconnection among their parts) since performance is not an issue.

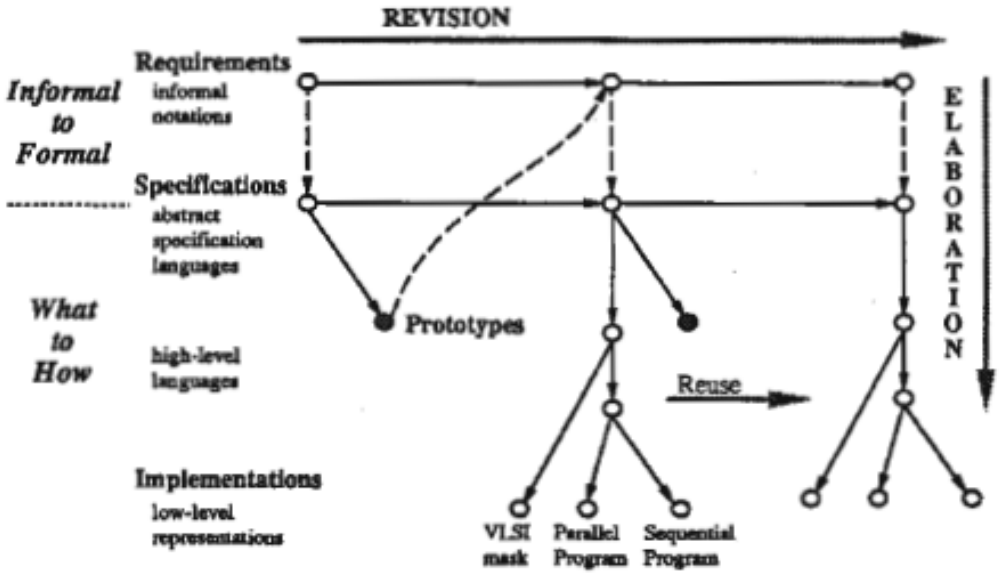

Figure 1: The design-based approach

As Figure 1 shows, there are several different levels of language in the DBA. Requirements are expressed informally, and often contain conflicting objectives which must be resolved in formalizing the problem statement. Specifications express formally what the problem is, and generally avoid commitment as to how to implement a solution, *e.g.*, they may be non-effective. Depending upon the specific development methodology chosen, however, specifications could be implementable, even serving as prototypes. In the transition from what to how, commitments are introduced; specifications are transformed into programs in a semantically clean language, these programs are further transformed to improve their efficiency, and ultimately they may be transformed into lower-level representations, such as well-known sequential implementation languages, parallel languages, or even VLSI masks.

Figure 1 also illustrates the *structure* of the program design problem as viewed in the DBA. There is no intention here to dictate a particular *design process*, or methodology; a programmer might start at any node in the diagram and work towards other nodes. We will briefly describe two classes of design process: program derivation and rapid prototyping. Although these approaches seem very different, in practice they would probably be used in some combination.

Generally speaking, *program derivation* involves formulating a specification in advance, and applying program derivation steps to obtain an implementation consistent with the specification. More specifically, we identify two classes of program derivation steps: *elaboration* and *revision*.

In a revision step, the *functionality* of a program is changed in reaction to changing requirements or in response to a better understanding of the requirements. *Elaboration* involves filling in details of design-decision steps; for example, it can be a process of making commitments (*e.g.*, to particular data-structure representations or to an order of computation) and then exploiting them for efficiency gains.

In a *rapid prototyping* paradigm, a programmer might forgo the initial formal specification and simply implement prototypes directly from an informal understanding of the requirements. Prototypes will have expediently chosen commitments that may not be appropriate for all implementations. These commitments could then be retracted to obtain more abstract specifications from which one could derive alternative implementations. Or, implementations could be derived directly from prototypes.

Eventually, it may be necessary to adapt an implemented software system to new requirements. It is expected that in the DBA such adaptation may be accomplished by *design reuse*. For example, the changes required to adapt a program might be confined to its abstract specification, and then, using the existing derivation by *analogy*, a new implementation would be derived.

There are several potential *benefits* of the DBA, and inferential programming in particular. In the first place, software maintenance under changing requirements will be simpler and conducted at a more conceptual level without compromising program correctness. In the second place, programs will be *correct by construction* with completely formal program derivation. The importance of this cannot be overemphasized, given the complexity of *a posteriori* program verification. In the third place, a record of design decisions will serve as explicit documentation, and abstractions of such records will serve as a basis for reuse of past design experience. And finally—but perhaps most important of all—adaptability and efficiency will coexist, rather than conflict as they do now. However attractive these may be, the *cost* associated with the DBA comes from the need to retain design information. Approaches being developed in our current research and elsewhere in the community indicate that the possibility exists for understanding the nature of design information—how it can be represented, and how it can be applied on an increasingly sophisticated scale. The management of design information, however, remains as a serious question in the development of a DBA.

2.2. Accomplishments of the Ergo Project since 1985.

We feel that during the past three years, our understanding of the DBA and the issues concerning its realization has increased considerably. We have accomplished research in three areas: *generic tool components for environment prototyping*, *program derivation techniques*, and *applications of inferential programming*. We were able to make this progress by combining theoretical research with program derivation methodologies, and by implementing experimental environments, thereby enabling us to put the methodologies into practice. In particular the *Ergo Support System* that we have developed is a collection of generic tool components that assists in the exploration of ideas in the DBA by enabling the rapid prototyping of experimental applications. It comprises

tools to support language implementation, program analysis and transformation, user interaction, and reasoning about programs.

Our research in *program-derivation techniques* covers the spectrum from high-level algorithm development methodologies to lower-level implementation optimizations. This work has been driven by concrete examples, and has provided invaluable input and feedback to the tool development. We studied *staging transformations*, including precomputation and frequency reduction, through compiler development examples [JS86a]. We presented techniques for deriving imperative programs that reuse storage in simple applicative programs [JS86b]. We explored design reuse from two different perspectives: abstract data-type specialization [Sch86] and the use of analogy in program development [DS87]. We also explored variations of inferential programming, including the transformation of statements about the semantics of the desired program [EP87] and deriving programs through proof transformation in a constructive logic [Pfe87a].

2.2.1. Generic tool components for environment prototyping.

We have long recognized the importance of providing an engineering framework for experimentation in the DBA. We believe that such a framework must support experimentation with languages *and* program development styles. We also believe that new forms of expression are required for a realization of design-based programming. This belief is based on our previous experimental research which indicates that the DBA imposes new requirements on programming languages due to the formal links that are maintained between specifications and implementations in the DBA. Therefore, experimentation with new and different languages and language concepts should be an integral part of research in the DBA. Moreover, the DBA admits many different concrete program design methodologies. We have strong evidence (in the form of examples) that different development styles will be appropriate for different problem types and problem domains. Thus, during the last three years of work on the Ergo Project, we have devoted considerable effort to designing and implementing a set of highly-integrated generic tools, collectively referred to as the *Ergo Support System* (ESS).

The ESS provides a rapid prototyping environment for researchers in design-based program development, program manipulation, and programming language theory. These tools allow us to build experimental environments with relatively modest programming effort. We learn from using these environments, and this then provides insights that spur the development of improved experimental systems. This view of the ESS is illustrated in part in Figure 2. Ideally, the ESS should support rapid prototyping of all aspects of programming languages pertinent to program-design environments. That is, one should be able to easily specify syntax, semantics, and transformation rules for a language and then obtain an environment tailored to manipulating programs in that language. Our goals are even broader, since we believe that logical languages and proof systems are essential to the development of an inferential programming environment (any DBA based on theoretical foundations), and that we must therefore be able to implement these languages as well.

We are still short of this broad goal, but have made significant progress towards its realization.

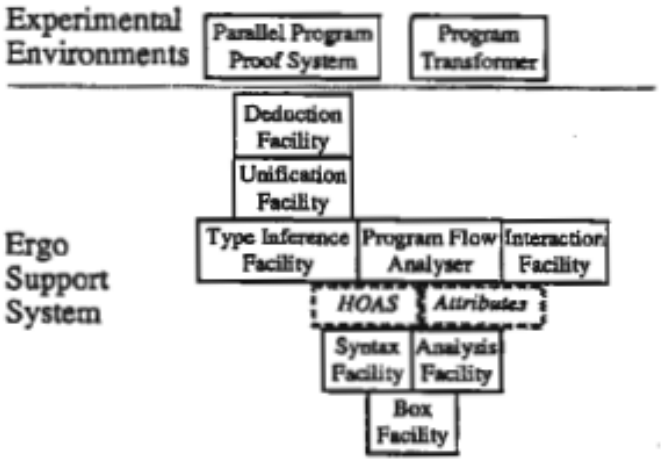

Figure 2: The Ergo Support System

Advances were made on the conceptual level in designing abstract interfaces between tools, answering the requirements of a language prototyping system for inferential programming. More specifically we designed *higher-order abstract syntax* (HOAS) as the central representation for formal objects, and we designed a new data type providing a coherent caching scheme for attributes and attribute sharing across many generations of a program or specification. Prototype implementations provided design feedback; further, bootstrapping of tools (building tools using our own tools) provided a higher degree of integration and flexibility of the system components.

Currently, the ESS provides tools for specifying the syntax and static semantics of languages in a simple, modular, and user-oriented fashion. Rules for transforming and manipulating syntactic objects can be specified, but we have only achieved partial integration of the former and latter sets of tools. We believe the integration of rules and the user interface descriptions with the other ESS tools lies in the near future, while full description and use of language semantics in the ESS is subject to further research.

The Ergo Support System tools. The ESS comprises tools to support language implementation, program analysis and transformation, user interaction, and reasoning about programs. We have developed tools and facilities to aid work in each of these areas. We briefly describe here what each of these facilities does.

The *Syntax Facility* [DPRS88,Die88] generates components for dealing with the syntactic structure of languages. From a language grammar it generates parsers, unparsers, and other useful syntactic manipulation components for use in other tools and application programs. A unique aspect of this facility is the highly readable and intuitively appealing metalanguage used for specifying the grammars.

The *Analysis Facility* [MN86a,Nor87b] provides for general-purpose program analysis by means of attribute grammars. As is well known, these grammars provide a convenient and effective framework for specifying many kinds of semantic analyses; for instance, it is possible to specify data-flow analysis. The Analysis Facility accepts attribute grammars specified in a

simple metalanguage closely related to the metalanguage used for syntax specification. From this it generates attribute evaluators. The generated evaluators are demand-driven, which reduces the amount of recomputation required when attribute values are changed. Furthermore, they employ a unique strategy for ensuring the consistency of the attribute values over the many incarnations of a program that might arise during its derivation.

The *Type Inference Facility* takes a specification of the semantic type signature of a language, and produces an attribute evaluator for performing type inference and type checking of programs in the language. Type inference can be performed up to the degree of polymorphism present in ML; partial type inference and type checking is possible for explicitly polymorphic languages in which full type inference is undecidable.

The *Program Flow Analyzer* (PFA) [Nor87a], is a tool for constructing application components that obtain control and data-flow information about programs. For example, a data-flow component called *Last Use* was developed for analyzing a program to determine for any variable reference if that reference is the last use of that variable. The PFA works from a specification of the partial semantics of a programming language, thus allowing a flow component to be specified independently of any specific programming language.

The *Higher-Order Unification Facility* [Ell87a,Ell87c,Ell87b] is a language-generic system for matching, unifying, and rewriting programs, formulas and other syntactic objects. This facility is unique, in that the matching and unification algorithms are not restricted to first order. The generalization to higher-order unification allows for very succinct formulation of transformation rules or rewrite equations in the presence of object language binding constructs.

The *Interaction Facility* [Fre86] provides high-level support for developing interactive program manipulation systems using keyboard, mouse and windows. The high degree of integration of this X-windows-based facility with the other components of the Ergo Support System allows for much faster prototyping of environments without sacrificing flexibility in the style of the user interface.

The *Deduction Facility* reads and analyzes a description of a logical language and deductive system and provides support for interactive and semi-automatic proof development in that system. The logical systems accepted by the Deduction Facility include first- and higher-order classical and intuitionistic logic, modal logics, Huet's Calculus of Constructions, the type theory underlying PRL and many others. Other uses include general type inference and Hoare-style reasoning about programs. Much of the Deduction Facility has been completed and the full implementation will be ready soon.

The *Box Facility* [Pfe86] provides automated configuration management for the ESS. For example, the Box Facility automatically activates the Syntax Facility when an attempt is made to execute a parser whose associated grammar specification has changed since the time the parser was last generated. Many of our tools create other tools from high-level specifications. The Box Facility handles this by allowing an arbitrary nesting of tool-generating tools.

Abstract interfaces. An important aspect of the tooling effort for the ESS is use of *component technology* in its development. We have designed several types of interfaces, both internal

abstract data types and external specification languages, which allow us to develop tools independently and yet achieve a high degree of integration. This greatly increases the usefulness of the ESS tools for prototyping, and furthermore offers the possibility of sharing of tools with other research groups. In the following discussion we give descriptions of the most important abstract data types we have developed for use in internal abstract interfaces.

Internal interfaces. *Higher-order abstract syntax*. (HOAS) [PE87] provides a new standard internal representation for programs, formulas, and other syntactic objects in the ESS. It has a first-order interface which is used for parsing and unparsing, and a higher-order interface which is used for transformation and manipulation and soon for various kinds of semantic analysis. The higher-order interface extends the usual notion of abstract syntax by allowing name binding constructs to be encoded in the typed -calculus. This allows many of the techniques used for analyzing -expressions to become applicable to the analysis of binding information in programs and formulas. For example, the Higher-Order Unification Facility provides language-generic higher-order unification functions for programs based on this representation.

Presently, only the Higher-Order Unification Facility uses the higher-order interface. New versions of the Analysis Facility and the Type Inference Facility which are also based on the higher-order interface are under development. First-order interfaces are defined through context-free grammars, and higher-order interfaces through the scope declarations, described below. Currently, the parsers generate first-order abstract syntax which is then converted by stylized LISP code to HOAS when necessary.

Attributes provide a standard means of annotating abstract syntax terms and subterms with semantic information. As in standard attribute grammars, the values of attributes associated with a particular term are defined by functions on the term's context, its subterms, and the term itself. The data type of attribute in the ESS [Pfe87b] also incorporates novel schemes which address issues of attribute consistency, attribute caching, and demand-driven attribute evaluation in a program-transformation environment where a large number of local program transformations often leave many attribute values unchanged. In addition to attribute grammars, type signatures and program flow signatures employ the data type of attribute. It is noteworthy that formal specification of this attribute interface requires dependent function types which are unavailable in most languages.

Boxes [Pfe86] provide a framework for specifying the functional relationship between the various functions and tools used to generate an environment. Conceptually, a box is a collection of functions mapping between abstract data types (*i.e.*, abstract interfaces). For example, the Syntax Facility consists of two higher-order functions: one, the parser generator, maps a grammar to a function from strings to abstract syntax trees, and the other, the unparser generator, maps a grammar to a function from abstract syntax trees to strings. At present, the ESS still assumes a file-oriented development environment. Hence, boxes are annotated with information about the files comprising a box.

External interfaces. In addition to internal interfaces, we have developed external interfaces in the form of formal specification languages. Ease of use, generality, and rapid adaptability were the overriding concerns in their design. These goals were achieved via "bootstrapping"

(*i.e.*, by building the second generation of these data types with our own tools). In the following paragraph we briefly describe this bootstrapping process for some of the external interfaces.

Concrete grammars, externally, are strings whose concrete syntax is defined through a metagrammar. This provided great flexibility (unparsing of grammars, adaptability of the grammar syntax) which was used extensively during the development of ESS tools. *Attribute grammars* are doubly bootstrapped; their syntax is defined through a metagrammar, and their semantics is defined through a metaattribute grammar. *Scope declarations* are currently provided in an *ad hoc* manner through LISP functions which define the higher-order view of abstract syntax. As in an earlier prototype, we plan to allow for bootstrapping by defining scope declaration with context-free grammars, attribute grammars, and higher-order matching. *Type signatures* are treated simply as a list of language constructs (abstract syntax operators and constants) annotated with their (usually polymorphic) types. *Program and data flow-annotations* are drawn from an evolving library of program and data-flow idioms. They are easily specified as properties of abstract-syntax operators.

2.2.2. Techniques for program derivation.

Our approach to building program design environments is largely experimental. However, this does not imply that we believe all foundational problems of the DBA have been solved. This is a long-term problem, but we now know how to make incremental progress towards an integrated program design environment. Our past research in this area has focused on the discovery of program-development paradigms and the gauging of the area of their applicability. Our results cover the spectrum from high-level algorithm development methodologies to lower-level implementation optimizations. In our work, the transformation techniques are always illustrated with concrete examples, and the experiences with them have thus provided invaluable input and feedback into the tool development.

In addition to the individual results, the two main lessons we learned from this research are:

> Program derivation can provide deep insight into the nature of algorithms and implementations.
>
> Only a variety of languages and methodologies will be able to cope with the variety of problem types and abstraction levels which arise during software development.

In the following, we briefly summarize what we consider our main contributions in this area of program-derivation techniques.

(1) *Staging transformations for compiler development* [JS86a]. Computations can generally be separated into stages, which are distinguished from one another by either frequency of execution or availability of data. Precomputation and frequency reduction involve moving computation among a collection of stages so that work is done as early as possible (so less time is required

in later steps) and as infrequently as possible (to reduce overall time). By means of examples, several general transformation techniques for carrying out precomputation transformations were presented. The techniques are illustrated by deriving fragments of simple compilers from interpreters, including an example of Prolog compilation, but the techniques are applicable in a broad range of circumstances.

(2) *Abstract data types and reuse* [Sch86]. We consider a view of programming experience as a network of programs that are generalizations and specializations on one another and that are interconnected by appropriate derivation fragments. This view is supported with a number of examples which illustrate the important role of abstract data-type boundaries in program derivation. A key observation is that code is an insufficient representation of the results of a design process and that design information must be explicitly represented.

(3) *Use of analogy in program development* [DS87]. We have illustrated how abstractions on program derivations can support a rich space of generalizations and analogies. It is argued that neither code nor abstracted code alone can serve as a basis for reuse, but that program-design information is essential for reuse of past design experience.

(4) *Transformations on destructive data types* [JS86b]. We present techniques for deriving, from simple applicative programs, efficient implementations that use destructive data operations and that can reuse heap-allocated storage. These techniques rely on simple propagation of non-interference assertions; reasoning about the global state of storage is not required for any of the examples presented.

(5) *Transformation of statements about the semantics of the desired program* [EP87]. We consider the specification to be a formula describing properties of the *meaning* of the desired program. The specification is then transformed (through logical inference) until it can be recognized as the meaning of a program. Using this methodology we developed program derivations for several complex algorithms for higher-order unification and its special cases. These derivations share insights and commitment steps, thus forming a family of derivations. To our knowledge, this is the largest program derivation effort to date. This paradigm can also be used to give a new interpretation to the fold/unfold method for program transformation.

(6) *Deriving programs through theorem proving and proof transformation in a constructive logic* [Pfe87a]. The use of the proofs-as-programs paradigm for algorithm design and program development is outlined. We argue that proving theorems in a constructive logic is in practice not sufficient to specify algorithms, but that the algorithm developer must have some tools for general proof transformation in order to formally develop efficient and complex algorithms.

2.2.3. Applications of inferential programming.

An example of a major application of our philosophy of inferential programming to program and algorithm development is the derivation—on paper—of algorithms for first-order and higher-order unification [EP87]. We have used these derivations in an essential way as a basis for an implementation in LISP as part of the ESS. Actually doing the formal derivation of Huet's com-

plex algorithm for higher-order unification led us to several substantial efficiency improvements and extensions. For example, we were able to incorporate products and polymorphism into the unification algorithm by conducting a second derivation guided by our original one. These extensions were necessary for our applications based on higher-order unification; furthermore, they will be required for the mechanization of the derivation itself. Such mechanization will complete a "bootstrapping" process—the formal derivation of parts of the ESS by applying the tools provided by the ESS itself. Not only are we improving our system, but we are learning what support the system has to provide for substantial derivations.

Since we regard this example as both a very significant illustration and as a sound motivation for further work, we would like to give more detail here on how we arrived at the present result. During early development of the ESS, we, like others, had used abstract syntax trees as the central data type for representing programs, derivations, and whatever other syntactic objects we needed to manipulate (grammars, attribute grammers, *etc*.). It quickly became clear that this representation did not allow us to specify even basic transformation rules in a simple and complete way. The well-known problem of the correct treatment of *bound variables* (or *local variables*) was the crux of the matter. A paper by Huet & Lang [HL78] on second-order matching and its uses seemed to provide a solution. Thus, we decided to base our central data type on the simply-typed -calculus (which incorporates bound variables in a natural way) rather than on the more usual first-order terms. Eventually this led to the notion of Higher-Order Abstract Syntax (HOAS) [PE88]. For use in the ESS, matching and substitution then require an implementation of Huet's second-order matching algorithm. Care must be taken, however, because second-order matching may possibly generate many different substitutions and can have exponential complexity even for simple examples. Even so, a first, language-specific prototype was put together to test the viability of Huet's ideas, and it was extremely successful.

The next logical step was to produce a language-independent implementation that could be used for any language as long as its binding constructs were made explicit in a uniform way through HOAS. Much to our chagrin, this second implementation proved quite impractical and also exposed some fundamental weaknesses in Huet's approach: namely, that a given pattern could only match functions with a *fixed* number of arguments. At this point we realized that we did not have a deep enough understanding of Huet's algorithm to generalize it or make nontrivial efficiency improvements. Thus, we embarked on the project of *formally deriving* the algorithm for full higher-order unification. As of yet, no implemented system is capable of handling a derivation of this order of magnitude and complexity, so it was to be a paper exercise—but we kept the goal of eventual mechanization clearly in mind.

Actually, there were at least three different, but closely related specifications we wanted to derive: (1) deciding higher-order unifiability, (2) enumerating a set of minimal and complete unifiers, and (3) higher-order matching (with one term being constant). We started with a simple one-line specification of the set of unifiers and, after some arduous work, arrived at a set of non-effective recursive equations. These could then be specialized to solve our three specifications. Moreover, this derivation suggested some further transformations leading to dramatic efficiency improvements that would reduce the algorithm's complexity from exponential to linear

or quadratic time in many common second-order cases. We also found that the proper way of handling the multiple-argument problem was through the use of products and polymorphism, rather than through the inclusion of lists as we had assumed earlier. We changed the specification again, and, guided by the previous derivations, we derived the extended version which had eluded us up to then. (The derivations are fully described in [EP87].)

The final set of recursive equations from our derivation was the basis of a Common LISP implementation which is now an integral part of the ESS. The formal derivation is now a part of our documentation [EP87] and should enable us to incorporate further improvements without compromising its correctness by first deriving an updated algorithm and then systematically (but, alas, still by hand) changing the LISP program. As in the past, we should also be able to adjust to changing requirements if they arise through our experience with the current implementation. What remains to be done is of course the *mechanization* of the derivation. This would be an important milestone: we would then have derived part of the improved ESS with the tools provided by the ESS. At present, we are prevented from accomplishing this only by the lack of a facility for maintaining the large accumulation of theorems and subderivations necessary for this complex algorithm derivation. We believe that a general, persistent object-base system might provide a solution here, but are not confident that such a system will be available soon. Hence, we are actively thinking of both general solutions to this difficulty and ways of working around the problem in the short term.

3. Future Research.

We now present our thoughts and proposals for our research strategy for the next three years. Section 3.1 gives a brief overview of the research plans, and then more detailed explanations are presented in the three following sections: 3.2 for Foundations, 3.3 for Applications, and 3.4 for Engineering. Each of these sections is composed of three parts which lay out a problem statement, our general approach, and more specific research plan. The purpose is to give summaries of the important technical issues as we perceive them, descriptions of our ideas on how best to address these issues, and finally details of how we want to implement these ideas.

3.1. Overview of the Ergo research strategy.

Research into the DBA necessarily encompasses many areas of both theoretical and experimental work in computer science. Such research always involves risk; however, we assert, this risk can be managed by adopting a strategy based on the construction of an *engineering framework* which we call an *Experimental Program-Design Environment* (EPDE). Such a framework would support not only continuing *experimentation* with various aspects and components of an EPDE as they are developed, but also the *rapid integration* of new theoretical results. The research proposed here therefore centers around the construction and use of just such a framework. Our intention is that this will serve as the basis for the development of an integrated environment

that supports program design, analysis and maintenance, rapid prototyping of language systems, and logic applications such as reasoning systems.

In the following subsections, as we said above, we present proposals for research on three major fronts: *Foundations*, *Applications*, and *Engineering*. This separation of topics does not imply any isolation or compartmentalization of thinking, however. We claim not only that there are strong ties between these three major research areas, but also that we are proposing a practical mechanism permitting essential feedback from one area to the other. We use work in foundations to provide a sound basis for formal ideas, but in addition this provides insights into the interplay between reasoning about programs and program derivations that suggest new ideas in our engineering projects. In the other direction the tools and environments we have been working on produce common, language-generic facilities. These are used not only as prototyping tools, but they are also employed to test logical and semantical ideas while they are being developed. We feel that application of inferential programming makes use of the above in both an experimental and foundational exploration into the nature of the program-design process. This exploration is based both on our previous work in inferential programming (resulting in many nontrivial derivations) and on a paper "database" of derivation ideas with which we propose to experiment.

Section 3.2 on *Foundations* is divided into three subsections concerning (1) axiomatic language description and program development, (2) languages for inferential programming, and (3) the transformation of semantic specifications. In each separate subsection, we have provided a statement of the problems to be solved, a description of our general approach to the research on these problems, and a more specific research plan. The theme of the first subsection is what we call the *proofs-as-programs paradigm*, but we take care not to isolate this work from thinking on standard programming languages. The theme of the second subsection is the development of wide-spectrum systems of languages by means of semantic modules, and their use in managing the passage between specifications and implementations. Finally, the theme of the third section is the derivation of program-transformation systems using denotational semantic definitions of programming languages expressed in suitable higher-order logical languages.

Section 3.3 on *Applications* proposes a continuing development for current Ergo work on making derivations of larger programs. These derivations are extensive enough for use in building parts of the Ergo system itself. Several other direct applications are also contemplated. The experience in doing these derivations will be used in investigations into design reuse, and also for improving many aspects of the system.

Section 3.4 on *Engineering* discusses not only the continued development of the Ergo system (which is essential for all the other activities), but it also presents a research plan focusing on high-level components, abstract interfaces, the uses of higher-order abstract syntax and abstract data-type specification, and the understanding of a program-design record. We are claiming that we have already made progress on these problems and that their development will have wide utility.

3.2. Foundations.

Theoretical and experimental research on the DBA must go hand in hand in order to achieve real progress. The word "foundations" for us means investigating *the interplay between reasoning about programs and program development*. We believe that *program correctness* and *language design* are the major issues of the foundational work for inferential programming. In particular, we will focus on the following topics: (1) axiomatic language description and its use in program development (including a reexamination of the *proofs-as-programs paradigm* and (generalized) *Hoare logics* in this context); (2) semantic language description and its relation to program development (with concentration on the design of a flexible semantic foundation for a *wide-spectrum system* of languages); (3) denotational semantics and its use in deriving programs or justifying program transformation rules (with an emphasis on the choice of an appropriate logical basis). The *Deduction Facility*, a high-level tool in the ESS, is the bridge spanning the gap between foundational and experimental research. It has been explicitly designed to support rapid prototyping of systems for semantically and axiomatically based reasoning about programs.

3.2.1. Axiomatic language description and program development.

This area of research in the Ergo Project is concerned with exploring how the axiomatic description of language can be exploited in the context of a program derivation system. Axiomatic language descriptions are given through inference systems for properties of programs, the most prominent of which are (1) various constructive type theories (proving properties of programs written in an appropriate -calculus) and (2) Hoare-logic systems (axiomatizing imperative languages). Currently, these two approaches have little in common, as constructive type theories mainly employ the *proofs-as-programs paradigm* (PPP), in which a program is extracted from a completed proof of the specification, whereas Hoare logics are used to verify previously given imperative programs.

Problem statement. We believe that the three main issues that need to be considered are: (1) the practicality of the PPP, (2) the usability of proof transformations in program development, and (3) the possibility of establishing a hybrid approach that allows the manipulation of *both* proofs *and* programs.

Practicality of proofs as programs. There are many questions to be decided to determine whether the PPP is more than a theoretical curiosity. Too pure an application of the idea will not have practical applications in real software development or in generating efficient implementations. We have to discover how much of a proof must be developed interactively (with the programmer making crucial decisions based on his own knowledge), and how much can be synthesized during the development of proofs based on properties required by the specification. In particular, it has to be shown whether the PPP is applicable to more practical languages than the -calculus. Even when practicality is demonstrated, we still have to consider whether highly formal proofs erect a barrier that inhibits intuitive understanding of an algorithm or program. When this is the case, further structuring will have to be designed, not only for the sake of the

human user, but also to allow the building of libraries of proofs that can be reused in a creative way in new combinations.

Proof transformations. If proofs correspond to verified programs, proof transformations correspond to program transformations. Much more work is needed to show what the basic proof-transformation techniques are. We also need to explain how we are to exploit the additional information about the program which can be found in its correctness proof. In particular, we do not know fully yet what the appropriate metalanguage is for writing programs that carry out proof transformations—something that is badly needed for the treatment of any large scale problem. Experience with logical deduction systems shows that an interactive mode of operation with machine-based tools is extremely helpful. We do not know yet, however, how we are to interact with a system that allows proof transformations, since experience to date has only been carried out on a very informal level. We also have to investigate whether proof transformations apply to Hoare logics as well as to constructive type theories which are closer to functional languages.

Integration of approaches. Is it possible to integrate different methodologies in the DBA into a design-based environment which allows many different development strategies? We believe so, but this is something that has to be demonstrated explicitly and tested on significant examples. In particular, we have to asses how large the gap is between traditional program derivation and the PPP and how we will be able to bridge this gap.

General approach. Our goal in this research is not only to understand the theoretical foundations for the use of axiomatic language descriptions in program development but simultaneously to prototype environments for relevant experimentation. As explained in Section 2, we have already found considerable value in using experimentation on paper, and the ESS has been explicitly designed and implemented to support rapid prototyping of such environments. We also believe that the tools necessary for the relatively small-scale experiments we plan in this area can be created quickly from the broad base of the existing ESS. In particular, we feel that testing the PPP is an excellent and relatively well understood starting point for investigating the issues we have outlined in the problem statement above.

We will continue to work on augmenting the scope of the PPP by introducing *proof transformations* in order to capture a larger class of program development steps (see [Pfe87a] for some preliminary results). We also propose to pursue another generalization that puts proofs and programs on a more equal footing and allows direct manipulation of programs while maintaining proofs. This could open up the technology to a potentially much larger class of users, since the program developer can choose to deal primarily with programs, which are more familiar objects than proofs. Below we elaborate on these approaches to using axiomatic language description in program development. In the paradigm of *pure program synthesis*, algorithms are developed by proving a theorem in a suitable logical system. The program is extracted as an afterthought implicit in the proof. If the proof is complete, the algorithm is verified. More rapid prototyping of algorithms is possible, since unverified programs can be extracted from incomplete proofs. The understanding of this technology is relatively advanced (embodied, for example, in the NuPrl proof-development system, see [BC85] and [Con86]).

Turning to *program transformations* and *proof transformations*, we note that a common misconception is to think of a constructive proof as a *design record* for the extracted program. This is only a weak analogy, since the proof only contains certain types of steps which form a small subset of elaborative steps: namely, those connecting logical specification with implementation, but not those connecting one implementation with another. Proof transformations achieve these more general steps. In approaching the connection between proof and programs from the side of programs, we note that most program transformations actually transform more than just programs. Rather, most techniques use programs *and* assertions about them (*e.g.*, loop invariants, strictness and termination properties) and proceed to transform those more complex objects. We take this idea a step further and, in essence, transform a program and its associated correctness proof into a new program with a new correctness proof. We claim that, since a correctness proof contains much information about a program, this information can be exploited to guide the transformation process. This is similar to the way assertions are used to permit some program transformations to be used in certain appropriate contexts.

An important principle of program transformations is to keep the set of basic rules small and prove them correct once and for all. The same principle applies here, and examples and preliminary experiments (see [Pfe87a]) indicate that four basic proof-transformation techniques suffice to generate a large number of practical algorithms. They can be named as follows (with analogous program-transformation techniques mentioned in parentheses): (1) *Reduction* (partial evaluation, unfold, application), (2) *Expansion* (fold, abstraction), (3) *Choice-function introduction* (steps which go from specifications to executable code), and (4) *Choice-function elimination* (steps which abstract from executable code to specification). But, there are two other considerations that should be mentioned as part of our approach.

First, *lateral steps* during program development, which are quite common in practice, change the specification. This happens, for example, when a program is to be adapted to the solution of a different, but related problem. The central issue concerning lateral steps is that of *design-information reuse*. In the PPP this is provided for very directly: lemmas (theorems, previously proved or accepted facts from former developments) can be used directly in the way a mathematician uses them. This often leads to inefficiencies, but proof transformation techniques can help to eliminate these inefficiencies. Metatheorems can also be used as proof transformers for lateral steps in algorithm design if the metatheory is sufficiently formal and powerful. The system we are developing is intended to make this easier to do in a formal way.

The second consideration again relates programs and proofs. Typically, proofs come first in the PPP and the program is extracted from it. Making the programs explicit in the proofs (as in Hoare logic and some type theories) opens up the possibility of having both programs and proofs *partially instantiated*. This blurs the distinction between various paradigms and will allow some writing of programs in a style similar to that familiar to present-day programmers, without leaving the security of verified program development and formalized design records. The explicit incorporation of proofs, in particular, allows us to place conventional development methodologies in a continuum. At the extremes we have *program verification* (*i.e.*, program given, find proof) and *program synthesis* (*i.e.*, find proof, extract program). In between these

we can distinguish various styles of *hybrid development* (*i.e.*, program partially given, proof partially given, complete both). This discussion always assumes that the program specifications are *given*. If one is interested in *generating* specifications, there is yet another dimension to the picture allowing the specifications we are trying to satisfy to be only partially instantiated in various respects. The implications of this broader view have to be explored further, but we believe we have established the context to do so.

Research plan. Our planned work involves investigating the foundations of axiomatic language description and its use in program development, carrying out examples both on paper and through prototyping simple environments for exploring the practical implications of our research. Different classes of languages, however, entail different styles of axiomatic description. Thus, we propose to follow three principal lines of investigation.

Functional program development through proof transformation. We plan to begin by rederiving—in the first place mainly on paper—a number of existing program derivation examples using this paradigm. We have already achieved this for several algorithms, Warshall's algorithm being our most complex example to date. [Pfe87a] Simultaneously we will investigate proof-transformation techniques and their properties, such as confluence, termination, or applicability. We expect the set of proof transformation rules to be relatively small and similar in structure to a fold/unfold or specialization system in that it derives its power through aggregation of small, local steps. Once we have developed an understanding of the transformation techniques, we need a metalanguage for expressing proof transformations and proof transformation strategies. This involves continued experiments with Prolog [MN86b] [MN87] and LF [HHP87]. We will also engage in the design of a system which allows "hybrid development" where programs and their correctness proofs are both incomplete at the outset of program development. We plan to carry out some selected examples in our own Deduction Facility prototype. We further plan to continue our work on transformation techniques and to expand on the examples of program derivations in this paradigm. A major project will be to derive some seven known algorithms for first-order unification. We are currently working on this set of examples using transformation of semantic specifications (see Section 3.2.3).

Generalized Hoare logics for imperative and parallel languages and their relation to program development. Here, too, we plan to start by deriving a number of well known imperative programs through transformation of programs together with their Hoare logic proofs. We hope to identify patterns of proof transformations in this setting. We plan to carry out a detailed comparison with other approaches to program transformation on imperative languages (for example [SJW86][JS86b][RS82][Mas86]). In particular, we will investigate where the additional information about the program (in the proof) enables or simplifies the recognition and application of transformations. We will also continue to investigate semantically based formulations of generalized Hoare logics for a larger class of programming languages, in particular for parallel languages (see [Bro85], [Bro86], and [Bro87]) and languages with powerful procedure mechanisms (see [Rey82]). We plan to apply transformation techniques to this larger class of generalized Hoare logics. In particular we will explore how proof transformations can be used on languages with parallel constructs and higher-order procedures that are axiomatized through

generalized Hoare logics. As a first step, a reimplementation of an existing Hoare-logic prover prototype using the Deduction Facility will enable us to carry out program derivation examples in the ESS. We also plan to depart from pure verification by allowing various degrees of instantiation of program and proof. Again, experimentation using the Deduction Facility should give us some insight into the practical ramifications of allowing such a development style.

The Deduction Facility. The Deduction Facility will be a high-level tool in the Ergo Support System that allows definition of a logical language and inference rules and creates an interactive environment for deduction in the given logic. We list it here with the above research plans, since it will provide the basis for some of the experimentation with proofs and programs. Also we think that the experience on the above problems will influence in essential ways the further development and redesign of the tool. In particular, proof system designers will interact closely with software developers. The concrete plans for the continued implementation of the Deduction Facility are given in Section 3.4

3.2.2. Languages for inferential programming.

The elaboration of specifications into implementations spans many levels of abstraction. Languages for the DBA must effectively support expression at all of these levels. Such support is often provided by a *wide-spectrum language* containing the high-level abstraction mechanisms necessary for comprehensible specification, as well as the lower-level, computationally committed mechanisms necessary for efficient implementation. To some extent, wide-spectrum languages such as SETL [DGL*79], Paddle [Wil83], CIP-L [Mol84], and REFINE [SKW85] have been successful in allowing the expression of both formal specifications and efficient implementations. However, the extent to which these languages can support a DBA, in particular inferential programming, is uncertain.

Problem statement. We see several general problems with the idea of wide-spectrum languages. First, insofar as a wide-spectrum language is a single language, it is inflexible with respect to the kinds of abstraction mechanisms it provides for high-level specification. Applicability to a broad range of problems can be enhanced by adding new constructs which either raise the possible level of abstraction, or provide alternative specification paradigms. However, such extension would incur a great expense in both design and implementation effort. Unfortunately, we think it is unlikely that any *particular* set of constructs can be appropriate for all applications, since new domains of application will give rise to new language requirements. Also, new advances in language design and implementation will make available increasingly abstract language mechanisms. It seems to us that the inflexibility of wide-spectrum languages will stand in the way of allowing any program-design environment to adapt to new problems and incorporate new technology as it is developed.

The second problem is that reasoning about specifications and programs in a wide-spectrum language is, at present, theoretically intractable. In most cases, there is no definition of the language semantics which is useful for formal reasoning. Even if formal definitions were to be constructed, it would be likely that they would be, at best, unwieldy, and that the underlying

semantic concepts and primitives used to describe the language constructs at one abstraction level will bear little or no useful relation to the concepts used to describe other levels. For example, the primitives typically used to describe the semantics of logic programming constructs [vEK76] are hardly related to those used for functional programming [MP82] or pattern-matching mechanisms [Ten78]. It seems us, therefore, that reasoning about wide-spectrum programs which may in fact contain such useful, yet semantically disparate, constructs would present major difficulties. This also often precludes reasoning about the correctness of the transformation rules, since the rules may also involve seemingly unrelated (at least on the semantical level) language constructs. This problem has been addressed in the design of CIP-L, which is based on an incrementally developed spectrum of semantic definitions. However, CIP-L is still based on a particular set of language features and abstraction mechanisms, and is also committed to particular implementation targets.

As we explain below, the approach we are going to take in avoiding the problems of wide-spectrum languages is to change the thrust to emphasize *systems* of languages. In thinking through the implications of adopting this point of view, we have identified three categories of issues for research in the area of wide-spectrum systems: (1) the engineering of languages in wide-spectrum systems, (2) the development of techniques for building a semantic framework for wide-spectrum systems, and (3) the organization and management of such systems.

In *language engineering* we must cope with the need to design and implement all of these diverse languages. In particular, since the languages are related, we have to decide how much of the implementation effort can or should be automated. More specifically, in the context of the Ergo Project, we have to determine just which tools should be provided by the ESS. Clearly, to be able to provide a *semantic foundation*, what is needed is something that provides a basis for semantically correct reasoning in all of the languages involved in the wide-spectrum system. We have to find out how such a semantic framework can be developed and how transformation rules can be extracted from experimental derivations in such frameworks. These problems are closely related to some of those discussed in Section 3.2.1. But, with respect to wide-spectrum systems, we additionally have to understand how such developments can be conducted incrementally and in what ways such frameworks can be utilized in language design and implementation. As regards *organization and management of wide-spectrum systems*, we need to ask how useful wide-spectrum systems are going to be developed and maintained over long periods of time. Can such activities be made accessible to program designers, instead of only specialists in semantics and compiler generation? If they cannot, our project will have failed in an crucial way. We have to learn how to break the problems up into feasible subprojects to avoid this kind of failure.

Aside from the large questions concerning wide-spectrum systems, we also see several major technical challenges in this research area. A key challenge will be to understand the result of the enrichment and restriction process (explained further in the next paragraphs) in terms of transformations of higher-level language features to lower-level ones, *and* also in terms of the structure of the design record. The development of the design record (see Sections 3.3 and 3.4) will be both influenced by, and provide guidance for, these efforts in language design. The organization provided by the Ergo Project will enable this necessary and close cooperation between these research efforts. We also see several other significant challenges: How abstract

can languages based on our design approach be? Will we always be bound to algorithmic languages? Can the enrichment/restriction process be formulated in a useful way for all, or sufficiently many, kinds of language constructs? How will the transformation rules be extracted from this process? How can the semantic information be utilized in the representation of the design record? We believe that the coordinated efforts in the research areas described in this proposal will be very effective in addressing these challenges.

General approach. In order to avoid the problems we have discussed concerning wide-spectrum languages, we need an alternative approach which involves the incremental design and development of a wide-spectrum *system* of languages, based on a modularly constructed and incrementally developed semantic framework. Languages are designed and added to this spectrum through a process that both extends the semantic framework *and* provides the transformation rules necessary for reasoning and derivation in the newly extended system. For program derivation, the main difference between such a wide-spectrum *system* and a wide-spectrum *language* is this: in a wide-spectrum system, specifications and programs will be written entirely in a particular language of the system. Elaboration will involve both the transformation of programs/specifications within a particular language, as well as the translation to the next lower-level language in the spectrum. We believe this approach will provide several advantages.

First, reasoning will be easier because each language and its formal semantics (as expressed in the system) will be smaller and less complicated. Still, the entire spectrum of abstraction mechanisms will be available during derivation, because the languages will have been fully related by their incremental development into the system, leaving the mixing of language levels to the program derivation where they are under careful control. As discussed in Section 3.2.3, this will also make it easier to reason about the correctness—or even derivation—of the transformation rules. Finally, incremental development will provide a level of *language-genericity* permitting the avoidance of excessive commitment to a particular set of language mechanisms. We believe that this will be especially important in reducing risk in our development of the DBA.

We also plan to show how the *design* of wide-spectrum systems can be carried out incrementally, by developing the idea of *semantic specification modules*. Such modules might be possible in the context of an *action-based* approach to formal semantic description (for example, consider the *abstract semantic algebras* of Mosses [Mos82], the *action semantics* of Mosses and Watt [MW86], and the *high-level semantics* of Lee and Pleban [Lee87], [PL88]; other approaches include the *local formalisms* of Wile [Wil86]). We and others believe that a synthesis of known techniques will allow the formulation of formal descriptions of programming languages by combining what we call *semantic modules*, each of which contain a description of some particular feature or type of construct of languages together with rules for combination with other modules.

As a simple example, a module might be created that encapsulates the notion of **goto**. This could then be combined (semiautomatically) with other modules according to its combining rules, in effect guiding the derivation of a new language description. A new, formally related language allowing the **goto** construct would result. Furthermore, the resulting derivation would serve as the basis for analysis and derivation of transformation rules relating the new language to the old.

Preliminary steps toward such modularity appear in [Lee87], [Wil86], and [Mos87]. We believe that this modularity will greatly enhance the usefulness of formal methods during language design in much the same way that modularity in programming languages is useful in programming. Even without the modularity, however, action-based semantic descriptions provide a basis for reasoning about programs. Furthermore, it has been demonstrated that efficient implementations of languages can often be automatically derived from action-based descriptions of their semantics [LP87]. Hence, we believe that the risk in our approach is relatively low.

Semantic modules are intuitively appealing since they can often be understood in terms of language concepts and mechanisms. Hence, their use becomes more accessible to program designers, not just to specialists in semantics and compiler generation. More importantly, this idea will permit the development of libraries of semantic modules which will serve as a basis for a *language designer's workbench*. Such a workbench will provide for the construction, testing, and management of semantic modules, as well as their combination into formally specified languages. Dependencies between the modules composing a language or wide-spectrum system can thus be maintained, through facilities for carefully controlled inspection and modification of semantic specifications. Finally, the workbench provides tools for exploiting the semantic framework in the formal derivation of transformation rules, as described in Section 3.2.3.

The design of a wide-spectrum system, then, can proceed as follows. First, the semantics of a "mid-spectra" language, say, a simple first-order imperative language IMP, is written in some modular form of an action-based semantics. A spectrum of languages is then created by a process of *enrichment and restriction*, accomplished via the addition and removal of semantic modules. For example, the next higher-level language in a spectrum might be obtained by enriching the IMP semantics with a module for higher-order functions, and restricting it by removing a module for assignment, thus resulting in an ML-like language [Mil85]. The next lower-level language, then, might come from an enrichment with **goto**, and restriction to simpler data types. Eventually, this process, properly carried out, leads to a spectrum extending from a low-level language to highly abstract specification language. Of course, many choices of enrichments and restrictions are possible, leading to many different wide-spectrum systems. This enrichment/restriction process is depicted in Figure 3.

At least two advantages will be realized from this approach. First, it will be possible to define semantic modules and methods for connecting them. This will greatly facilitate our efforts in the incremental design and development of wide-spectrum systems. Secondly, it will be possible to obtain rapidly some efficient prototypes for many of the languages, and this will improve the development of the ESS and the EPDE. For implementation purposes, we expect that previously developed compiler generation techniques, and new, Ergo-developed facilities for flow analysis [Nor87a] and partial evaluation will be important.

Research plan. Our proposed research, following the approach outlined above, blends experimentation with investigations into semantic foundations. As a starting point for the development of semantic modules, we will implement an action-based semantics system based on Lee's MESS system [LP87] in the ESS. This will provide initial rapid language prototyping capabilities, as well as an engineering base on which to build further support for language design. Integration of

Figure 3: Example development of a possible wide-spectrum system.

MESS concepts into the ESS will also provide direct support for experimentation with semantics transformations. Then, using this experimental base, we will develop techniques for constructing semantic modules, and make first steps in formulating the enrichment/restriction process. We plan to base these developments on an extension of the high-level semantics method supported by MESS. We believe this extension will be a significant advancement in formal semantic description. In particular, we believe this will lead to the development of a semantic foundation for wide-spectrum systems. When a version of the enrichment and restriction process has been formulated, we will implement it in the semantics system to obtain the first version of the Language Designer's Workbench.

At this point we will be able to develop and experiment with small wide-spectrum systems and methods for extracting transformations from the enrichment/restriction process. In close association with the research in semantics transformation (see Section 3.2.3), methods will then be developed for extracting transformation rules from wide-spectrum systems. These rules will be used in program derivations based on the languages in the system. With sufficient experimentation along these lines, we expect to be able to develop a spectrum suitable for use in the larger derivations described in Section 3.3. Continued research and experimentation should involve use of the semantic framework in the representation of the design record and the incorporation of the Language Designer's Workbench into the EPDE.

3.2.3. Transformation of semantic specifications.

The overall aim of this phase of the work of the Ergo Project is to establish not only a rigorous but also a flexible and usable foundation for inferential programming that makes explicit and formal use of programming-language semantics.

Problem statement. As we see the matter, an attack on the handling of semantic specification requires a formal deduction system supporting (1) the theories involved in algorithm development, (2) the semantic domains for programming languages, (3) the syntax of programming languages, and (4) the semantic functions relating syntax to semantics. Unlike current program derivation and verification systems and formalisms, such a system must necessarily allow for explicit reasoning about termination properties, as well as a broad class of implementation languages (*e.g.*, call-by-name and call-by-value functional, imperative, logic) rather than a single one. It must also permit the systematic *derivation* of program transformation rules as semantic equivalences.

To achieve these specific aims there also a number of general questions that have to be studied. We have yet to answer, for instance, the question of what a good meta-language for the formalization of a language and its underlying semantic domains should be. For instance, can we extend a system like LCF [GMW79] to encompass higher-order logic? Can we use higher-order unification to mechanize (forward and backward) applications of inference rules and to construct derived inference rules as in Isabelle [Pau86]? We can also ask how we can shield the user from the details of the semantic level, especially when targeting towards imperative programming languages, and whether we can synthesize useful systems of program transformation rules within our system. There are several logic-based systems for reasoning about programs, but to make them more widely useful we have to determine whether we can remedy the inability of systems like NuPrl [BC85] [Con86] and the Calculus of Constructions [CH85] to reason easily about programs that do not always terminate. We also have to find out if we can provide a useful link between the seemingly opposed approaches of program verification and program synthesis.

General approach. For the purposes of the present discussion, we regard a *specification* as being expressed as a predicate in the language formalizing the semantic domains underlying the programming language. An *implementation* is a program whose meaning (in one of the semantic domains) satisfies the specification. *Program derivation* means to apply *deduction steps* to the specification with the goal to arrive at a description of a function in the semantic domain that can be shown to be the meaning of a program.

We think that a useful meta-language would be a higher-order logic with implicit polymorphism (as in ML) supporting predomains, and more generally reasoning that is more like familiar set-theoretical reasoning in mathematics. This may be compared with LCF, which is a first-order logic with implicit polymorphism allowing only domains and continuous functions. It may take some time to find a convenient framework that is both formal and sufficiently general. Such a generalization is important for a number of reasons. For example, using the framework of predomains, we can directly represent many of the the mathematical types we want to work with. This makes them more intuitive and also easier to mechanize especially for reasoning about meanings of programs in call-by-value and imperative languages. Then, in higher-order logic, we can express and prove higher-level facts such as the validity of various induction laws. In LCF, data-type inductions must be simulated by complex fixed point inductions each time they are used, resulting in cumbersome reasoning. There are other meta-language concerns as well. The programmer or language designer should be insulated from the low-level semantic

concepts by allowing the introduction of definitions and new notations for them. Experience with NuPrl [Con86] and the Environment for Formal Systems [Gri87] suggests that this is a simple but very effective device which we hope to exploit further.

Following Prolog [MN86b,MN87] and Isabelle [Pau86], we plan to use *higher-order unification* instead of pattern-matching as the mechanism for applying inference rules. This means that the (forwards or backwards) application of an inference rule may partially instantiate the theorem it is proving. This idea merges the apparently opposing approaches of theorem proving (program verification) on the one hand, from theorem discovery (program synthesis) on the other. The Deduction Facility (see Section 3.4) is explicitly designed to support higher-order unification as an inference engine and will be the basis of our implementation.

Research plan. Our research on this approach to program derivation will be very strongly driven by examples. We plan to continue our work on the *derivation* of existing program transformation systems using the denotational semantic definition of the programming language. This has been done already for the fold/unfold system for a simple functional language. Other examples will include at least functional and imperative programming languages. We also plan to assemble a casebook of examples for derivations using explicit transformation of semantic specifications. The first major example is a derivation of a complex higher-order unification algorithm (see [EP87]). Work on a genealogy of first-order unification algorithms using this paradigm is in progress.

Our experience suggests certain features of the meta-language as outlined above. We plan to design a meta-language for the description of language semantics and for reasoning about the objects in the semantic domain including these features. A prototype implementation of this meta-language in the ESS using the Deduction Facility should provide us with essential feedback as to its adequacy.

3.3. Application of inferential programming.

Larger programs require a great deal of information management as well as the use of tools for viewing larger amounts of information in a sensible way. We will gain experience in this area by developing some larger-scale applications using inferential programming methodologies. Along the way we will develop additional tools, data structures, and techniques to handle these larger programs.

Problem Statement. There are three orthogonal axes along which we can measure our progress in the application of the methods of inferential programming: (1) the degree of formality of the derivation, (2) the complexity of the resulting algorithms and programs, and (3) ease of reuse of design information. Exploration of each of these dimensions presents its own problems. Indirectly, these criteria will also give us a measurement of the maturity of our tools and of our understanding of the paradigms in the DBA. Thus, we regard these investigations—and the design of support tools—as an integral and important part of our research effort over the next few years.

How formal is the derivation? Program derivations can be located on a continuum between "fully formal and implemented on the computer" and "completely informal". One point in between is illustrated by our derivation of unification algorithms [EP87]. That work was carried out completely on paper. It was rigorous in carrying out major design steps, but completely left out many intervening small steps. As the tools in the ESS mature and higher-level components and more complete languages become available, we hope to make substantial progress towards increasingly formal algorithm derivations. This will expose new issues which are conveniently ignored in less formal derivations, and will provide us with feedback about the adequacy of our tools.

How complex are the specification and the resulting algorithms and programs? Small examples with simple specifications and moderately complex algorithms do not pose a major challenge. Some of our examples below have a potentially large scope with the possibility of highly optimized implementations of very complex algorithms. We hope as time progresses that the complexity of the generated code will increase, allowing more efficient implementations. At the same time we hope to *reduce* the complexity of the specifications—the simpler the specification, the fewer commitments will have been made at the outset and the fewer errors are likely. Thus, over time we are exploring the space of languages which connect specifications and implementations "from the inside out": starting with high-level languages we are working towards more powerful specification languages on the one side, and towards lower-level languages which allow more tailoring to the target architecture on the other.

How easy is it to reuse design information? Devising suitable representations for the design information that links a comprehensible specification with derived low-level efficient programs is critical for providing computer support for carrying out non-trivial program derivations. Convenience of design information reuse is related to the degree to which there is explicit sharing of structure among a set of related derivations.

General Approach. Before and during the current research contract the Ergo Project has completed the following significant derivations: parsing algorithms [Sch80], graph algorithms [RS82], compiler generation [JS86a], abstract data type derivations [Sch86], use of destructive data types [JS86b], and first-order and higher-order unification [EP87]. The main purposes of these examples was to illustrate program derivation techniques and, in some cases, to gain a deeper understanding of an algorithm for an implementation. We can find commonality among these derivations which will guide us in the design of data structures for representing derivations. We will attempt to mechanize these derivations completely, although this is a major challenge. The goal will be to demonstrate that program derivation can serve as a practical tool in software development which goes beyond informal algorithm explanation.

In particular, we have concrete plans for deriving programs in three major problem classes, which we now explain. Naturally, others may arise in the course of our research. The three classes are: (1) *Rederivation of past examples*. The use of previously developed informal derivations will allow us to focus on the problems of formalizing derivations, since the necessary design insights have already been made. We can study the array of existing derivations and find commonality to exploit in developing interactive derivation tools, data structures for representing

derivations, and derivation techniques. (2) *Bootstrap of the* ESS. As mentioned earlier, derivation of selected components of the ESS will provide more adaptable tools and good inferential programming experience. We expect that first-order unification will be the first example of an ESS component bootstrapped completely with our own tools. (3) *Derivation of theorem provers*. The goal is to derive theorem-proving procedures from the specification of the logical system (language and inference rules). The groundwork has been laid through derivations of first- and higher-order unification algorithms.

Turning to the problem of the nature of the *design record* (or account of the design decisions), we assert that the most fundamental purposes of the design record are (1) the support of reuse of design information, through formal representation of designs and design decisions, and (2) the support of explanation of programs, through both formal and informal representations of design processes. As we describe in Section 3.4, we will use the insights gained from our extensive experience in formal derivations to design an *implementation* of the design record. This engineering-based approach is, we believe, the most practical and direct way to provide a basis for experimentation and research into the *formal* nature of the design record.

Research Plan. Specifically, we have the following plans in this research area. In the first place, we will carry out the formalization of the various derivations mentioned above. This should allow us to identify the salient features of the derivation process, thereby providing us with a basis for the design of prototype design record implementations. We expect that experimentation and evolution of such implementations will also lead to a more formal understanding of the design record. Next, in order to augment the pool of derivation examples from which we can draw insights, we will survey derivational methodologies and incorporate our findings into the same evolutionary process. Finally, continued development of a prototype implementation of the design record will be carried out to accommodate revision and elaboration design steps, thus allowing flexibility with respect to temporal ordering of user actions. In addition, we will work to incorporate abstraction and generalization mechanisms for design records. It should also be noted that in conventional software development methodologies, support for reuse is largely limited to fully committed code (*e.g.*, subroutines, modules, or packages from a code library). The addition of polymorphism and higher-order functions to programming languages carries this reusability significantly further. However, the DBA can provide a high degree of support for structured reuse and adaptation. We have already begun studying how to generalize concrete derivations and apply existing derivations by analogy to new programs [DS87], and we plan to continue this line of investigation as well.

3.4. Engineering of tools and environments.

The Ergo project has already devoted considerable effort to designing and implementing a set of highly-integrated generic tools, collectively referred to as the Ergo Support System (ESS), which is intended to provide a rapid prototyping environment for researchers in design-based program development, program manipulation, and programming language theory. Ideally, the ESS should support rapid prototyping of *all* aspects of programming languages pertinent to program-design

environments. That is, it should be easy to specify syntax, semantics, and transformation rules for a language and obtain an environment tailored to manipulating programs in that language. Our goals in Ergo are even broader, since we believe that logical languages and proof systems are essential to an inferential programming environment, and that we must be able to implement these as well.

Problem Statement. New forms of expression are required for a realization of design-based programming. We have long recognized the importance of providing an engineering framework for experimentation in the DBA, and we also believe that such a framework must support experimentation with *both* languages *and* program development styles. This belief is based on our previous experimental research which indicates that the DBA imposes new requirements on programming languages; therefore, experimentation with a variety of languages and language concepts must be an integral part of our research.

Currently, the ESS provides tools for specifying the syntax and static semantics of languages in a simple, modular, and user-oriented fashion. Rules for transforming and manipulating syntactic objects can be specified, but we have only achieved partial integration of the former and latter sets of tools, and much more needs to be done. We believe the integration of rules and the user interface descriptions with the other ESS tools lies in the near future, while full description and use of language semantics in the ESS is subject to further research. There are two main considerations driving the maintenance and further development of the system: (1) high-level support for experimentation, and (2) continued development of the internal structure (in the spirit of component technology) to support the exchange of components with other research groups, to upgrade existing tools, and to allow the rapid prototyping of new tools.

Other research groups are also producing relevant system components that we hope to exploit. We have to give serious thought, however, to the many problems of sharing software. It is particularly clear that an *object-oriented database facility* in an inferential programming environment would serve the role that a file system serves in a conventional system, providing support for structured objects instead of just text files, and contributing to much improved performance of systems. At this juncture we cannot see just which group is going to develop this especially desirable component in a general-purpose form that can be widely shared. Nevertheless, we already are taking preliminary steps to collaborate with other research groups that are actively engaged in building prototypes of such facilities [WA87]. We feel our careful attention to abstract interfaces and component technology will aid us in incorporating object-base technology as it is developed for the research community. In another direction, we feel a *hypertext system* would allow us to incorporate informal explanation into formal derivation objects, thereby enhancing the user interface and giving us a more flexible approach to documentation. Currently the expression of interest in hypertext is no more than a statement of intent on our part, but steps are being taken to develop concrete plans for exchange of tools and components with other research groups that have similar interests.

General Approach. Inasmuch as the DBA admits many different concrete program design methodologies, we have strong evidence (in the form of examples) that different development styles will be appropriate for different problem types and problem domains. An important part

of our general aims is to demonstrate that we can accommodate the different styles within a single support system.

A major aspect of our implementation activities to date has been program development in the spirit of *component technology*. This not only simplifies program maintenance and rapid prototyping of new tools, but also supports the exchange of components with other research groups. The most visible results of our development style are the *abstract data-type specifications* for abstract interfaces between system components and the use of *bootstrapping* of components whenever possible. We feel we have made very substantial advances on the conceptual level in designing *abstract interfaces* between tools and answering the requirements of a language prototyping system for inferential programming. More specifically, we designed *higher-order abstract syntax* [PE88] as the central representation for formal objects, and a new data type providing a coherent caching scheme for attributes and attribute sharing across many generations of a program [Pfe87b]. Simultaneously, prototype implementations provided design feedback, and the bootstrapping of tools (*i.e.*, building tools using our own tools) provided a high degree of integration and flexibility of the system components.

Components will only integrate well if their abstract interfaces are well-designed and precisely described. We hope to evolve current descriptions into a state where they could serve as a basis for a *metalanguage* which would become part of the ESS itself. Current languages for specifying abstract data types do not seem appropriate for concise description of the internal interfaces of the ESS. Thus our first step is to design the meta-abstract data type of data type specifications in which we plan to describe formally the major abstract interfaces in the ESS. This will be a prerequisite for work on a more complete ESS bootstrap. Note that the idealized goal of *bootstrapping* is to implement the whole ESS in itself. Currently, our understanding of inferential programming is advanced enough that only engineering problems stand in the way of achieving this goal for some components, like the first-order and higher-order unification tools. However, further foundational research is required for most others. A fully bootstrapped implementation of first-order unification is planned and would serve a dual role as a major application for the ESS tools *and* as a part of the ESS itself.

Research plan. We will continue to develop our applications in the framework of the ESS. This will guide its further design and development. Abstractions of current prototype applications should develop into new, higher-level components of the ESS.

The Deduction Facility is the first example of such a tool, and we plan to continue its development. This generic proof manipulation tool can be used for natural deduction, type inference, Hoare-style reasoning about programs, and for reasoning in semantic domains. Its design was strongly influenced by Prolog [MN86b,MN87] and LF [HHP87]. It is based on an explicit *metalogic* (LF) rather than an explicit meta-programming language (ML). The metalogic can be given a computational interpretation in the style of Prolog. The implementation of the Deduction Facility will proceed in three stages. The first stage, an implementation of the metalogic, has already been completed. During the second stage we will use the Interaction Facility in the ESS to build an interactive environment for theory definition, proof development and transformation. In the third stage we will implement a version of Prolog which will give

us a powerful language to express tactics for theorem proving and proof transformation.

The second example is expected to be a *language-generic program derivation environment*. From high-level user-oriented definitions (of language, static semantics, types, transformation rules, interaction style, *etc*.), this tool will generate a specialized derivation systems in an otherwise uniform environment. A first prototype will be based on an implementation of a data type representing the program design record. Since we will continue to expand our library of derivations, we will be able to study the kinds of design steps that need to be represented, as well as the design states that programs typically pass through during derivation. We expect that our study of derivations, along with experimentation, will allow us to refine our notions of the design record, leading eventually to its formalization. Further plans include derivation replay and analogy capabilities.

Moreover, we will use abstraction mechanisms to modularize large, complex, and highly interconnected derivations, making them understandable and reusable. Design record abstractions will be based on the sources of modularity inherent in derivations due to the independence of many design decisions. For example, the simplest kind of modularity in a derivation is that present in a single design state, which is generally a program composed of encapsulated modules. This program modularity can be exploited by decomposing a design state along its internal abstraction boundaries, and carrying out sub-derivations independently on the parts. Modularity in the design record will not only make derivations more manageable, but will increase the potential reusability of designs, since each sub-derivation will be independently reuseable. Another simple form of abstraction we will exploit is the aggregation of many small design steps into a single higher-level step. Generalizations of such aggregations will permit the reuse of a design strategy as a single design step. It is clear from our past experience with derivations that many derivation techniques are commonly used even across unrelated applications.

We recognize that the "scaling up" of applications and high-level components will require support for the management of persistent structured objects [Nes86], [BZ87], [Wie86], [ML87]. Our plan is to import and install an *object-oriented database facility* when a suitable one becomes available. As we mentioned earlier, we are taking preliminary steps to collaborate with other research groups involved in the development of such facilities.

In addition to higher-level components, we have plans to develop the following two specific applications.

A model language implementation. We will implement a new experimental language, *Idealized Algol*, that combines a simple imperative language with a procedure mechanism based on the lambda calculus [Rey81]. The type structure of the present version of this language makes extensive use of conjunctive types (in the sense of Coppo and Dezani [CDV81]). As a consequence, the language not only subsumes Algol, but also supports object-oriented programming. We expect that the ESS will substantially reduce the effort required for this implementation; conversely the implementation will provide a practical test of the adequacy of the support system (particularly its concept of abstract syntax). The clean and well-understood semantics of Idealized Algol [Ole82], [Ole85] should also make the language an excellent vehicle for program transformation

and formal proofs. In particular, its programs can be reasoned about in Specification Logic, a partial correctness formalism that embeds Hoare logic, dealing with the imperative aspects of programs, within an intuitionistic logic dealing with -calculus-based procedure mechanism [Rey82], [Ten85].

Investigations of Specification Logic. We will use the Deduction Facility to implement a system for reasoning in Specification Logic. It is clear from examples of proofs in this logic that automated support should greatly reduce tedious details. Moreover, the intuitionistic aspect of the logic suggests that the system design can usefully draw upon ideas in systems such as NuPrl [Con86].

4. Personnel.

The following lists the faculty, students, and staff presently affiliated with the Ergo Project.

Faculty.

Stephen Brookes Stephen.Brookes@cs.cmu.edu (412) 268-8820
Semantics of programming languages
Reasoning about programs

Peter Lee Peter.Lee@cs.cmu.edu (412) 268-3049
Environment prototyping tools
Semantic specification and implementation

Daniel Leivant Daniel.Leivant@cs.cmu.edu (412) 268-8817
Proof system design
Reasoning about computational complexity

Frank Pfenning Frank.Pfenning@cs.cmu.edu (412) 268-6343
Environment prototyping tools
Theorem proving and proof transformation
Program derivation

John Reynolds John.Reynolds@cs.cmu.edu (412) 268-3057
Design of programming languages
Theory of polymorphism
Semantics for conjunctive types

Eugene Rollins Gene.Rollins@cs.cmu.edu (412) 268-5684
Persistent Typed Objects
Environment prototyping tools
Program derivation

Dana Scott Dana.Scott@cs.cmu.edu (412) 268-2566
Semantics of programming languages
Reasoning about programs

Graduate Students.

Penny Anderson Penny.Anderson@cs.cmu.edu
Program derivation - design record

Kenneth Cline Kenneth.Cline@cs.cmu.edu
Type Inference

Anne De Niel Anne.DeNiel@cs.cmu.edu
Partial Evaluation

Scott Dietzen Scott.Dietzen@cs.cmu.edu
Program Derivation - design reuse

Conal Elliott Conal.Elliott@cs.cmu.edu
Transformation of semantic specifications
Program Derivation

Tim Freeman Tim.Freeman@cs.cmu.edu
Program derivation
Environment prototyping tools

Nevin Heintze Nevin.Heintze@cs.cmu.edu
Constraint Logic Programming

Robert Nord Robert.Nord@cs.cmu.edu
Program derivation
Program flow analysis
Environment Prototyping Tools

Technical Staff.

Michael Mills Michael.Mills@cs.cmu.edu
Systems administration

Mary Ann Pike Mary.Ann.Pike@cs.cmu.edu
Technical writing

5. Ergo Papers and Reports.

[Bro85] Stephen Brookes. An axiomatic treatment of a parallel language. In *Proc. IEEE Conference on Logic in Computer Science*, Springer Verlag (LNCS 239), 1985.

[Bro86] Stephen Brookes. A semantically based proof system for deadlock and partial correctness in CSP. In *Proc. IEEE Conference on Logic in Computer Science*, IEEE Press, Boston, June 1986.

[Bro87] Stephen Brookes. Semantically-based axiomatics. In *Proc. IEEE Conference on Logic in Computer Science*, Springer Verlag, Berlin, April 1987.

[Die88] Scott R. Dietzen. *The Ergo Approach to Syntax Manipulation*. Ergo Report In preparation, Carnegie Mellon University, Pittsburgh, 1988.

[DPRS88] Scott R. Dietzen, Mary Ann Pike, Anne M. Rogers, and William L. Scherlis. *User's Guide to the Ergo Syntax Facility*. Ergo Report In preparation, Carnegie Mellon University, Pittsburgh, 1988.

[DS87] Scott R. Dietzen and William L. Scherlis. Analogy in program development. In J. C. Boudreaux, B. W. Hamill, and R. Jernigan, editors, *The Role of Language in Problem Solving 2*, pages 95–117, North-Holland, 1987.

[Ell87a] Conal Elliott. *Ergo Higher-Order Terms*. Ergo Report 87–041, Carnegie Mellon University, Pittsburgh, December 1987.

[Ell87b] Conal Elliott. *A Guide to Ergo Typed Lambda Calculus, Higher-Order Unification, and Hooking Up an Object Language to Both*. Ergo Report 87–044, Carnegie Mellon University, Pittsburgh, December 1987.

[Ell87c] Conal Elliott. *Some Annotated Examples of Higher-Order Unification in the Ergo Support System*. Ergo Report 87–043, Carnegie Mellon University, Pittsburgh, October 1987.

[EP87] Conal Elliott and Frank Pfenning. *A Family of Program Derivations for Higher-Order Unification*. Ergo Report 87-045, Carnegie Mellon University, Pittsburgh, November 1987.

[Fre86] Tim Freeman. *Object Oriented Interaction Facility*. Ergo Report 86–003, Carnegie Mellon University, Pittsburgh, August 1986.

[JS86a] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Thirteenth Symposium on Principles of Programming Languages*, pages 86–96, ACM, January 1986.

[JS86b] Ulrik Jørring and William L. Scherlis. Deriving and using destructive data types. In *IFIP TC2 Working Conference on Program Specification and Transformation*, North-Holland, 1986.

[Lee87] Peter Lee. *The Automatic Generation of Realistic Compilers From High-Level Semantic Descriptions*. PhD thesis, University of Michigan, May 1987.

[LP87] Peter Lee and Uwe F. Pleban. A realistic compiler generator based on high-level semantics. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich*, pages 284–295, ACM, January 1987.

[MN86a] William Maddox and Robert L. Nord. *The Ergo Analysis Facility*. Ergo Report 86–001, Carnegie Mellon University, Pittsburgh, June 1986.

[Nor87a] Robert L. Nord. *A Framework for Program Flow Analysis*. Ergo Report 87-038, Carnegie Mellon University, Pittsburgh, November 1987.

[Nor87b] Robert L. Nord. *New Ergo Analysis Facility Release Notes*. Ergo Report 87–033, Carnegie Mellon University, Pittsburgh, December 1987.

[PE87] Frank Pfenning and Conal Elliott. *Higher-Order Abstract Syntax*. Ergo Report 87-036, Carnegie Mellon University, Pittsburgh, November 1987.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation*, ACM, June 1988. To appear.

[Pfe86] Frank Pfenning. *The Box Tool*. Ergo Report 86–005, Carnegie Mellon University, Pittsburgh, August 1986.

[Pfe87a] Frank Pfenning. *Program Development through Proof Transformation*. Ergo Report 87-047, Carnegie Mellon University, Pittsburgh, December 1987. Talk give at the *Workshop on Logic and Computation*, June 1987, Pittsburgh.

[Pfe87b] Frank Pfenning. *A View on Attributes in the Ergo Support System*. Ergo Report 87-031, Carnegie Mellon University, Pittsburgh, March 1987.

[PL88] Uwe F. Pleban and Peter Lee. High-level semantics — an integrated approach to programming language semantics and the specification of implementations. In *Proceedings of the Third Workshop on the Mathematical Foundations of Programming Semantics, New Orleans, April 1987*, Springer-Verlag, Berlin, 1988.

[Rey81] John Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*, pages 345–372, North-Holland, 1981.

[Rey82] John Reynolds. Idealized algol and its specification logic. In D. Neel, editor, *Tools and Notions for Program Construction*, pages 121–161, Cambridge University Press, 1982.

[RS82] John H. Reif and William L. Scherlis. *Deriving Efficient Graph Algorithms*. Technical Report CMU-CS-82-155, Carnegie Mellon University, Pittsburgh, December 1982. To appear in TOPLAS.

[SAng] David Steier and Penny Anderson. *Comparing Algorithm Syntheses*. Technical Report, Carnegie Mellon University, Pittsburgh, (forthcoming).

[Sch80] William L. Scherlis. *Expression Procedures and Program Derivation*. PhD thesis, Stanford University, August 1980. Available as Technical Report Stan-CS-80-818.

[Sch84] William L. Scherlis. Software development and inferential programming. In *NATO ASI Series, Vol. F8 Program Transformation and Programming Environments*, Springer-Verlag, 1984.

[Sch86] William L. Scherlis. Abstract data types, specialization and program reuse. In *International Workshop on Advanced Programming Environments*, Springer-Verlag LNCS 244, 1986.

[SS83] William L. Scherlis and Dana S. Scott. First steps towards inferential programming. In R.E.A. Mason, editor, *Information Processing*, pages 199–212, Elsevier Science Publishers, 1983.

6. General Bibliography.

[BC85] Joseph Bates and Robert Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[Bjo87] Dines Bjørner. On the use of formal methods in software development. In *Ninth International Conference on Software Engineering*, IEEE, 1987.

[BP77] Manfred Broy and Peter Pepper. Program development as a formal activity. *IEEE Transactions on Software Engineering*, SE-7(1):44–67, 1977.

[BZ87] Toby Bloom and Stanley B. Zdonik. Issues in the design of object-oriented database programming languages. In Norman Meyrowitz, editor, *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando*, pages 441–451, ACM, December 1987.

[CDV81] Mario Coppo, Maria Dezani-Ciancaglini, and B. Venneri. Functional character of solvable terms. *Zeitschrift für mathematische Logic und Grundlagen der Mathematik*, 27:45–58, 1981.

[CH85] Thierry Coquand and Gérard Huet. Constructions: a higher order proof system for mechanizing mathematics. In *EUROCAL85*, Springer-Verlag LNCS 203, 1985.

[Con86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CS77] K.L. Clark and S. Sickel. Predicate logic: a calculus for the derivation of programs. In *IJCAI5*, 1977.

[Dar78] John Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11(1):1–30, 1978.

[DGL*79] R. B. K. Dewar, A. Grand, S. C. Liu, J. T. Schwartz, and E. Schonberg. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Transactions on Programming Languages and Systems*, 1(1):27–49, July 1979.

[Fea87] Martin S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, to appear 1987.

[GMW79] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[Gri87] Timothy G. Griffin. *An Environment for Formal Systems*. Technical Report 87-846, Department of Computer Science, Cornell University, Ithaca, New York, June 1987.

[HHP87] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, IEEE, June 1987.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[Hog81] C. J. Hogger. Derivation of logic programs. *Journal of the Association for Computing Machinery*, 28(2):372–392, April 1981.

[Mas86] Ian A. Mason. *The Semantics of Destructive Lisp*. Volume 5 of *CSLI Lecture Notes*, Center for the Study of Language and Information, Stanford, 1986.

[Mil85] Robin Milner. The Standard ML core language. *Polymorphism*, II(2), October 1985.

[ML87] Thomas Merrow and Jane Laursen. A pragmatic system for shared persistent objects. In Norman Meyrowitz, editor, *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando*, pages 103–110, ACM, December 1987.

[MN86b] Dale A. Miller and Gopalan Nadathur. Higher-order logic programming. In *Proceedings of the Third International Conference on Logic Programming*, Springer Verlag, July 1986.

[MN87] Dale A. Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *Symposium on Logic Programming, San Francisco*, IEEE, September 1987.

[Mos82] Peter D. Mosses. Abstract semantic algebras! In D. Bjørner, editor, *Formal Description of Programming Concepts II*, pages 63–88, North Holland, Amsterdam, 1982.

[Mos87] Peter D. Mosses. Modularity in action semantics. In *Proceedings of the Workshop on Semantic Issues in Human and Computer Languages, Half Moon Bay, California*, Center for the Study of Language and Information, Stanford University, March 1987.

[MP82] Steven S. Muchnick and Uwe F. Pleban. A semantic comparison of LISP and Scheme. In *Proceedings of the 1982 ACM Conference on LISP and Functional Programming, Stanford*, pages 95–107, ACM, August 1982.

[Mol84] B. Möller. *A survey of the project CIP: Computer-aided, intuition-guided programming*. Technical Report TUM–18406, Institut für Informatik der TU München, Munich, West Germany, 1984.

[MW86] Peter D. Mosses and David A. Watt. The use of action semantics. In *Formal Description of Programming Concepts III*, North Holland, Amsterdam, 1986.

[Nes86] John Nestor. Towards a persistent object base. In *Proceedings of the PE Workshop '86: International Workshop on Advanced Programming Environments*, IFIP Working Group 2.4 on Systems Programming Languages, 1986.

[Ole82] Frank Oles. *A Category-Theoretic Approach to the Semantics of Programming Languages*. PhD thesis, Syracuse University, August 1982.

[Ole85] Frank Oles. Type algebras, functor categories, and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantic*, pages 543–573, Cambridge University Press, 1985.

[Pau86] Lawrence Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[SJW86] Fatima Ali Hussain Stefan Jaehnichen and Matthias Weber. Program development by transformation and refinement. In *International Workshop on Advanced Programming Environments*, Springer-Verlag LNCS 244, 1986.

[SKW85] Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, November 1985.

[Ten78] Robert D. Tennent. Mathematical semantics of SNOBOL4. In *Proceedings of the First Annual ACM Symposium on Principles of Programming Languages, Boston*, pages 95–107, ACM, January 1978.

[Ten85] Robert Tennent. Semantical analysis of specification logic (preliminary report). In R. Parikh, editor, *Tools and Notions for Program Construction*, pages 373–386, Springer-Verlag LNCS 193, Berlin, 1985.

[Tur82] D.A. Turner. Functional programming and proofs of program correctness. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 187–210, Cambridge University Press, 1982.

[vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, 23(4):733–743, October 1976.

[WA87] David S. Wile and Dennis G. Allard. Worlds: an organizing structure for object-bases. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Palo Alto*, pages 16–26, ACM, December 1987.

[Wie86] Douglas Wiebe. A distributed repository for immutable persistent objects. In Norman Meyrowitz, editor, *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages and Applications, Portland*, pages 453–465, ACM, November 1986.

[Wil83] David S. Wile. Program developments: formal explanations of implementations. *Communications of the ACM*, 26(11):902–911, November 1983.

[Wil86] David S. Wile. Local formalisms: widening the spectrum of wide-spectrum languages. In L.G.L.T. Meertens, editor, *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Toelz, FRG*, pages 459–481, North-Holland, November 1986.