Semantically Based Programming Tools (Summary)

William L. Scherlis and *Dana S. Scott*

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213, USA

December 1984

The design and development of semantically based programming tools is a goal whose fulfillment is unquestionably many years off. The vision has been articulated in various forms by many researchers (including us in IFIP 83), but our impression now is that the expression of the vision provides little more than spiritual support and that there is an urgent need to distinguish shorter term goals along the way to mechanized tools.

While there does indeed seem to be some consensus that a notion of program derivation or "meta-program" is required (and there are indeed many informal examples of derivations in the literature), little has been said concerning the formal structure of these objects and how we will need to operate on them. What distinguishes derivations from mere proofs? What are the basic derivation steps? What deductive support is required? How can we assess the value of proposed derivation structures? Also, while much has been said concerning the functionality of the proposed tools, it is safe to say that we are still quite uncertain of their intended behavior and mode of interaction. What will be the correct balance of responsibility between programmer

This research was supported in part by the Office of Naval Research under Contract N0014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DoD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. 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 the Defense Advanced Research Projects Agency or the U.S. Government.

and tool and how will they interact? Can we realistically expect to build tools that will capture enough knowledge that they will become useful in practice?

This brief paper is intended to draw attention to several of the shorter-term issues and problems. Our recent focus has been on ths construction of a system for experimentation, so the bulk of our comments here will be on this aspect. Even though there are substantial theoretical problems still to be overcome, it is possible now to make progress towards experimental systems. The second section of the paper contains some remarks concerning programmer interaction with semantically based tools (what we have been calling inferential programming systems).

1. Towards experimental semantically based programming tools.

While a number of implemented systems have been built for experimenting with interactive theorem proving, program transformation, heuristically guided programming, and the like, we are still far from having a satisfactory framework for experimental inferential programming tools. The ML and LCF systems are perhaps the best examples currently available, but they are not well suited to the large scale syntactic manipulations that will be required. More importantly, ML is not as well able to support, as compared with Lisp, the (relatively) large scale system development we are undertaking.

We evaluated a number of possible architectures for prototype inferential programming systems, and it became immediately clear that the overall system design would continue to elude us until a number of conceptual and engineering questions had been settled. But we did find that in all of our scenarios, certain basic system functions were required above and beyond those provided in conventional programming environments. The Ergo Support System (ESS) is being created in order to provide an environment in which a diversity of implementation paths can be explored—an environment that will support the construction of prototype semantically based tools.

In our present view, the Support System includes four basic components, (1) an object and type management facility we call the *datafile*, (2) a set of *language*

tools, including a syntactic processor for supporting experimentation with programming and logical languages, (3) a basic *deductive facility* for supporting inference and simplification of expressions, and (4) an *interaction and view management* facility that supports communication between the system and human users. As the Support System evolves, other components will be added. (The Support System has been under active development since Spring 1984; it is being implemented in a Common Lisp environment.) The initial applications of the Support System will be experimentation with mechanizing several existing program derivations in order to better understand what kind of deductive support is needed.

While the Support System development will soon provide a useful set of facilities, a longer term goal is the development of metalanguage for controlling the support system and, ultimately, expressing knowledge about programming. Three requirements for the metalanguage are a suitable type framework for organizing objects, language for accessing the support system, and a means of easily expressing deductive knowledge. This deductive metaphor for computation is especially vivid in program manipulation systems, and a useful metalanguage should have a strong deductive flavor. Again, the best existing model for this language and its relationship to the system is ML.

Datafile. A principal source of complexity in large systems is the number and variety of the objects they manipulate and the lack of a uniform mechanism for managing the objects. This will be particularly true for program manipulation systems, in which it will be necessary to manipulate objects such as program text, program specifications, facts about programs, domain-specific facts, proofs of facts, documentation text, compiled code, program transformations, program derivations, program derivation patterns (for representing heuristic knowledge), and even mail messages and window descriptions for displays. Adding to the complexity is the fact that most of these objects, considered abstractly, will need a variety of representations appropriate to different computational needs (and appropriate to different storage media such as files, displays, and run-time storage). Ordinary software engineering considerations

dictate that a uniform means—beyond typing—be developed for managing the many objects, kinds of objects, and representations.

In the Ergo Support System, the *datafile* facility serves in this information management role. The datafile functions as a combined file system and database by providing a means of naming, storing, classifying, and retrieving objects and collections of objects.

A major function of the datafile is classification and search management. The language for naming objects and collections of objects must, in order to permit fast search, be more expressive than that of an ordinary file system. But, in order to retain flexibility, we have avoided strong commitment to particular database organizations. Two basic mechanisms are provided—*classes*, which are special objects that represent sets of objects, and a simple set-theoretic language of *retrieval expressions* (which themselves are also objects) for describing classes. The datafile keeps track of selected retrieval expression objects and their values during incremental updates, thus enabling rapid search of the classes described even when the datafile state has changed.

Besides naming and search support, the datafile has a rudimentary object typing mechanism. The types are simply uninterpreted symbols; thus, there is no commitment in the support system to a particular ontology or organization of types. But this rudimentary typing system does provide special support for multiple representations of objects. Program fragments, for example, are most usually represented internally as annotated abstract syntax trees, but on a user's display or in an externally available file, they are best represented as concrete-syntactic text, and in an internal text file the best structure may be Lisp-like lists. The datafile helps keep track of the multiple instances and representations and maintain consistency as appropriate—allowing programs that use the datafile to manipulate objects without regard to representation.

Syntactic Tools. The major objects of attention in programming tools are, of course, programs. While it would be dangerous to make commitments to any specific languages of programs, support of a generic sort can be provided for manipulation of

programs and other complex objects. The Support System provides sophisticated tools for compiling grammar specifications into parsers, unparsers, formatters, matchers, and so on. These tools provide a uniform means for associating concrete syntax with objects such as programs, specifications, transformations, assertions, and so on. Although syntax may appear to be a trivial problem, it is essential that it be handled gracefully even in prototype semantically based systems, in which so many different languages must be used; the user interface should not be the limiting factor to the success of experimental systems.

Reasoning Facility. The bulk of the computational activity in a semantically based tool is deduction. We are pursuing in our implementation a general approach that allows most of the formal reasoning activity to be described in a uniform framework, including proving assertions about programs, making simplifications to formulas and programs, applying transformations, and carrying out computations on virtual machines. This deductive framework would be the mechanism that carries out the fundamental program derivation steps and interacts with the user to plot strategy. LCF is the major antecedent of our design, though there will be important differences. For example, we are interested in directly manipulating proof objects and abstractions on proof structure.

Our recent efforts have been concentrated on building or collecting an implemented base of decision procedures that will facilitate inference in theories such as simple recursive data types and Presburger arithmetic, and on integrating these with a simple rewriting and pattern matching facility. We are not yet at the point of approaching substantive issues in this aspect of the implementation.

But, on the basis of the informal program derivations that exist in the literature, it can be argued that in most circumstances only a modest inference capability is required for most derivation steps. Of course, there are cases where substantial intellectual jumps must be made; in the first prototype tools these jumps will need to be made by the user.

Interaction Facility. In developing mechanisms by which users can interact with semantically based tools, perhaps most essential to recognize is the overwhelming complexity and size of the objects that will be manipulated. In order to evoke a decision, request advice, or simply inform a user concerning a particular object, such as a theorem to be proved or a program derivation step or a program specification, it is usually necessary for the tool to shield the user from the full extent of detail contained in the object being considered—and instead to present some view (i.e., projection or abstraction) of the complete object. Each such view presents a single aspect of the object; a collection of these views can provide a more natural and useful portrayal for a given interaction than a single monolithic representation. The use of views for manipulating objects allows internal object representations to include a wealth of supplementary information.

2. User Interaction with an Inferential Programming System.

If we are to develop useful tools, we must take a realistic attitude towards the way in which programmers will interact with semantically based environments. We must accept, for example, the fact that the vast majority of programming steps will likely involve the revision of previously made decisions, either because requirements have changed or in order to explore further a space of possible designs.

An example of inferential programming activity. Consider the sorts of abstract types that might be used in the implementation of a simple compiler. For certain types, such as syntax trees or run-time stack frames, choice of representation has a significant effect on the ultimate compiler design. In both of these cases the objects are used in so many different ways that, once the necessary specialized operations are included in the type interface, there would arguably be little representational structure left to hide. The author of a code generator, for example, will need to know the full details of stack frame layout in order to generate tolerably efficient code sequences for block entry and so on. It could then be argued that there is no point to

making the abstract type explicit in the code since the intended abstraction boundary must be so consistently violated.

In the program derivation framework, uses of abstractions and the definitions of their representations need not coexist within individual programs (i.e., at individual derivation steps), but are instead spread over the derivation. This idea has been illustrated in a number of derivations in the literature. In these examples, instances of abstract operations appearing in early derivation steps are replaced in later steps by the corresponding operations on the representation, which, in still later steps, may be optimized based on the context in which they appear.

How, then, would we use an inferential programming system to help maintain a program containing abstract types in this regime? Ideally, every user interaction would result in addition of structure to the derivation under construction, either through commitment, simplification, or whatever. But, since semantically based tools are to be used by human programmers, we must allow for backtracking and revision of earlier decisions. For example, it may be necessary to revise various commitment steps along the way from specification to implementation, or it may be necessary to revise the specification itself (if it exists yet—derivations need not be constructed in any particular order).

Inferential programming steps. This brings us to our point. In interacting with an inferential programming tool, a user can make two kinds of steps. One would hope that most steps are *elaborative* steps, in which commitments or simplifications are made, augmenting an existing partial derivation. But there will always be some number of *lateral* steps, in which functional changes or design changes are made.

Attempts to work out this kind of scenario in greater detail inevitably lead to the question of what is the "right" notion of program derivation. Our view, and this is somewhat of a philosophical commitment, is that this is primarily a foundational question—as distinct from an analytical one. Analysing our programming experience will provide direction, but our purpose is not to build tools that imitate this experience.

Rather, our intent is to formulate notions of program derivation (together with criteria for their adequacy) through investigations into semantics and transformations.

3. Conclusions.

It is one of our fundamental theses that major improvements in software engineering practice will come about only through the development and use of software development tools. This thesis is based on our belief that formal methods will ultimately have a far more profound effect on software engineering productivity than management based methods, programming language design, or fast hardware. But we also believe that formal methods, by their nature, are suitable for practical use only in mechanized systems. This is not to say that *everything* is to be automated; the point is that the actual formal steps must be automated even if most of the guidance for their use is to come from the user. This is the same kind of observation that prompted the developers of LCF to introduce the ML type structure to protect the notion of "theorem," together with formal deductive structure that defines it, from the surrounding heuristic apparatus. It must be expected that heuristics will be under continual development, but a deductive system is fragile and will change only infrequently.

Acknowledgements. We thank Penny Anderson, Scott Dietzen, Paola Giannini, and Carl Gunter for careful readings of this paper on short notice.