

An NSF Proposal

Robert Paige

*Department of Computer Science, Courant Institute, New York University
251 Mercer Street, New York, NY 10012, USA*

Abstract. The objectives of this research are to improve software productivity, reliability, and performance of complex systems. The approach combines program transformations, sometimes in reflective ways, to turn very high level perspicuous specifications into efficient implementations. These transformations will be implemented in a meta-transformational system, which itself will be transformed from an executable specification into efficient code. Experiments will be conducted to assess the research objectives in scaled up applications targetted to systems that perform complex program analysis and translation.

The transformations to be used include dominated convergence (for implementing fixed points efficiently), finite differencing (for replacing costly repeated calculations by less expensive incremental counterparts), data structure selection (for simulating associative access on a RAM in real time), and partial evaluation (for eliminating interpretive overhead and simplification). Correctness of these transformations, of user-defined transformations, and of the transformational system itself will be addressed in part. Both the partial evaluator and components of the transformational system that perform inference and conditional rewriting will be derived by transformation from high level specifications. Other transformations will be specified in terms of Datalog-like inference and conditional rewriting rules that should be amenable to various forms of rule induction.

Previously, [Cai and Paige 93] used an ideal model of productivity free from all human factors in order to demonstrate experimentally how a transformation from a low level specification language into C could be used to obtain a five-fold increase in the productivity of efficient algorithm implementation in C in comparison to hand-coded C. However, only small-scale examples were considered. The proposed research includes a plan to expand this model of productivity to involve other specification languages (and their transformation to C), and to conduct experiments to demonstrate how to obtain a similar five-fold improvement in productivity for large-scale examples of C programs that might exceed 100,000 lines.

The proposal lays out extensive evidence to support the approach, which will be evaluated together with its theoretical underpinnings through substantial experiments. If successful, the results are expected to have important scientific and economic impact. They are also expected to make interesting, new pedagogical connections between the areas of programming languages, software engineering, databases, artificial intelligence, and algorithms.

1. Introduction

Program Transformations is about semantics-based analysis and manipulation of programs. Over the past twenty years we have made contributions to the area by developing two distinct tracks: (1) general-



© 2005 Kluwer Academic Publishers. Printed in the Netherlands.

purpose problem specification and its transformation to efficient programs, and (2) special-purpose specification of systems that implement the program analysis and transformations used in Track (1). The long term goal is to combine both tracks.

Previously, using the approach of Track (1) we were able to demonstrate effective translations of high level specifications of algorithms into high performance codes limited to small-scale examples. In [12] Cai and Paige developed an ideal model of productivity free from human factors. Within this model they gave experimental evidence that their transformational approach to program development leads to at least a five-fold improvement in productivity of efficient algorithm implementation in C as compared to hand-coded C. However, those experiments only considered small-scale examples of procedureless programs no more than ten pages long. In the proposed grant period, we plan to extend our specification languages and the transformations that implement them in order to specify and develop efficient large-scale systems with a similar improvement in productivity.

Previously, using the approach of Track (2) we were able to demonstrate effective development of large-scale systems limited to inefficient prototypes. The complex translation of a statically typed variant of SETL into C used in the productivity experiments mentioned above was specified in RSL (Rule Specification Language), a high level language implemented by the APTS program transformation system [37]. The translation suffered from two major sources of inefficiency. First, APTS is implemented in SETL2 [50], which runs 30 times slower than C at best. Second, APTS only provides an interpretive implementation for RSL. Hence, the translation of SETL to C was bogged down to 24 lines per minute on a SPARC 2. In the proposed grant period we plan experiments to test the feasibility of a radical new transformational methodology that combines reflective forms of partial evaluation (to eliminate interpretive overhead) and data structure selection (to replace the naive SETL2 runtime) used in Track (1) in order to gain a 300-fold speedup in RSL execution.

By combining improvements to both tracks, we plan to demonstrate a five- to ten-fold improvement in productivity for implementing large-scale systems with more than 100,000 lines of C. Our automated program development methodology is most effective in development of systems with high algorithmic content, which are among the most difficult to construct and maintain by hand. Included in this class of systems are those that implement complex program analysis and transformation, which is the application domain for the research proposed here. An example is the SETL-to-C translator, an RSL specification which our methods are expected to speedup by a factor of 300.

Section 2 of this proposal describes a cohesive research project that should take three to five years to complete for three Ph.D. students and the PI. Section 3 details a number of experiments, including a novel, reflective combination of partial evaluation and data structure selection, that are expected to be completed within the three year funding period. Although this proposal only seeks funding for one Ph.D. student, two other student participants will be funded by other sources – one by NYU fellowship and another by an ONR grant that partly overlaps with this proposal.

2. Background and Objectives

2.1. TRACK (1) BACKGROUND

2.1.1. *Specification Languages and Transformations*

Within Track (1) we consider an implementation language (e.g., C) that serves as a conventional RAM model of computation. We also consider three successively more abstract specification languages, each implemented in terms of the next successively lower level language by a distinct transformation. By associating syntactic constructs with asymptotic complexity, we are able to predict precisely how each transformation can improve program running time and space. Consequently, the selection of transformations can be guided by complexity considerations, and the three specification languages are made *computationally transparent* (i.e., amenable to formal algorithmic analysis).

Low SETL, our lowest level specification language, is a statically typed, executable variant of SETL2 [50], a pointerless, block structured, imperative language augmented with a repertoire of primitive set operations such as membership testing, element addition and deletion, arbitrary choice, for-loops through a set, map application, indexed map assignment, and so forth. Its perspicuousness rests on copy/value semantics, and its ability to navigate through data directly rather than by indirect location formulas using pointers or cursors. Associative access and nondeterministic selection and search contribute to its readability and succinctness. The data structure selection transformation [38, 36, 7] is based on a low level theory of data structures in which set and map operations (such as associative access) are simulated on a RAM in real time; i.e., a conventional physical structure written in C is selected for implementing each primitive set operation (for search arguments and type-compatible set or map domain elements of any data type) in unit WORST-case time.

An important aspect of this work is that the type system (fragments of which are found in [7, 28, 24, 40]) is a strongly typed variant

of the Curry/Hindley type discipline for the λ -calculus [17, 26]. It is parametric since it contains type variables, but currently is not polymorphic. The type system together with the data structure selection transformations make it possible to analyze Low SETL programs for their worst-case time and space complexity. This allows programmers to be guided by complexity considerations, which is essential to the production of high performance systems. Using Low SETL as a solid foundation, the other two specification languages obtain computational transparency by the way they are mapped into Low SETL.

High SETL is a statically typed, imperative, executable superset of Low SETL augmented with such abstract operations as set comprehension, quantification, and a variety of operations on binary relations. Computational transparency is obtained by implementing High SETL expressions in Low SETL in two ways. Firstly, the cost of fresh evaluations of high level expressions can be determined by implementing them directly into Low SETL. More interestingly, we can use our finite differencing transformation [31, 35] to evaluate repeated costly High SETL expressions by more inexpensive incremental counterparts in Low SETL. We determine the cost of these differential calculations by associating precise amortized complexities with an eager strategy for maintaining equality invariants $E = f(x_1, \dots, x_k)$ within worst-case sequences of modifications to variables x_1, \dots, x_k ; i.e., each time a modification to x_1, \dots, x_k occurs, variable E is updated to reestablish the invariant. High SETL allows us to avoid having to write error-prone bookkeeping operations that maintain invariants differentially. By associating amortized complexity with maintenance of basic invariants, and by closure rules that allow us to determine the cost of maintaining collections of interdependent invariants, we can generate languages of High SETL invariants that can be maintained differentially with precise complexities [8].

The highest level specification language is SQ2+ [9], a nonexecutable, statically typed, functional subset of High SETL augmented with least and greatest fixed points. These fixed point expressions abstract error-prone, iterative looping constructs. Computational transparency is obtained for SQ2+ by associating fixed point operations with precise implementations in High SETL using our dominated convergence transformation [9]. Dominated convergence computes fixed points in terms of High SETL loops generating ‘chaotic’, finitely converging sequences [16] that are less expensive than Tarski sequences [51]. Both fresh and differential calculations of fixed point expressions have been considered. We also developed specification languages that are subsets of SQ2+ with guaranteed worst-case execution complexities in time and space for each polynomial degree [8, 10].

Over the years we have uncovered extensive evidence that our methodology will scale up and be capable of improving the productivity of large-scale system implementation. Three kinds of evidence have been considered. The specification languages and the transformations that implement them have been shown to be effective in terms of (1) explaining complex algorithms, (2) discovering new algorithms, and, most importantly, (3) improving the productivity of efficient implementations of algorithms. In order to enhance the credibility of this proposal, we will present some of this evidence in the following two subsections.

2.1.2. *Algorithm Explanation and Discovery*

Most mainstream research in Program Transformations emphasizes principles for constructing derivations with illustrations drawn from known algorithms. The purpose has been to uncover common patterns of abstraction in specification and transformation that could form the basis of a useful methodology for making algorithm design and program development easier. Short, elegant transformational proofs (of correctness integrated with analysis) document implementations of complex algorithms, lend greater confidence in the correctness of complex codes, and provide greater assurance in the reliable modification of such codes.

Perhaps the first examples of nontrivial algorithms being derived by finite differencing were presented in [34, 31]. Included among these examples is a SETL specification of Dijkstra's naive Bankers Algorithm and its transformation into Habermann's efficient solution. This derivation was done without knowledge of Habermann's solution, and there were no dead ends. Finite differencing homed right in on a solution matching Habermann's time/space bounds.

It is well known that the construction of optimizing compilers is a costly labor intensive task. Can this labor be reduced by our methods? To answer this question in part, we showed how easy it was to specify dozens of programming language and compiler analysis problems in SQ2+, to simplify these specifications, and to transform them by dominated convergence into High SETL prototypes [9]. In [8] we showed how the constant propagation algorithm of Reif and Lewis [42] could be expressed as set-theoretic equations in a subset of SQ2+ that could be mapped into RAM code guaranteed to run in linear time in the size of the program dataflow relation.

The use of notation has been regarded as a burden to algorithm designers ever since Knuth came out with Mix [30]. But can notation also help satisfy the needs of the algorithm community – precise algorithmic analysis and succinct exposition? An SQ2+ specification of the Single Function Coarsest Partition Problem and its transformation by dominated convergence, finite differencing, and real-time simulation

was used to derive a new linear time solution [39]. That algorithm paper was selected for publication in a special issue of TCS as a best paper from ICALP. In [40] a much improved explanation of the algorithmic tool called Multiset Discrimination in [11] was obtained by specifying the algorithm in Low SETL and using its type system to formally explain and analyze the low level implementation that would be obtained by real-time simulation. The earlier presentation of this algorithmic device involved so much indirection from pointer-based primitives, that most readers were confused. Some of these earlier readers agreed that the Low SETL presentation clarified their understanding.

The viability of a transformational methodology can be demonstrated by using it to ‘explain’ or ‘prove’ well known algorithms. However, if in the course of such formal explication no new deep structure is uncovered that leads to improved solutions, then its impact on algorithmics and programming productivity is likely to be limited. More powerful evidence favoring a transformational methodology is provided if it can be demonstrated to help facilitate the discovery of new algorithms. Our success with algorithm discovery may be attributed to our reliance on complexity in both specification and transformation.

Our first instance of algorithm discovery by transformation was reported in [38], where we used all three transformations to turn an SQ2+ specification of Horn Clause Propositional Satisfiability into a new linear time pointer machine algorithm. Previously, Dowling and Gallier found a linear time algorithm [19] that relied heavily on array access. In [4] we used dominated convergence and finite differencing to derive a Low SETL executable prototype from an SQ2+ specification of ready simulation. We then showed informally how the Low SETL prototype could be turned into an algorithm that runs 5 orders of magnitude faster than the previous solution in [3]. All three transformations, but especially real-time simulation (where types were shown to be useful in modeling complex data structures), were involved in the discovery of a new improved solution to the classical problem of DFA minimization [28]. Finite differencing was used extensively in [15] to derive a new improved solution to the classical problem of turning regular expressions into DFA’s. Perhaps our most convincing paper-and-pencil result was in [24], where Goyal and Paige used Low SETL specifications and real-time simulation to improve Willard’s time bound for query processing from linear expected to linear worst-case time without degrading space. Willard’s original algorithm was extremely difficult, and involved over 80 pages of proofs. Our transformational approach yielded much shorter but also more precise constructive proofs that led to an implementation design. Here is a first successful example of scaling up.

Two summers ago, Ph.D. student Deepak Goyal designed and implemented ‘practical’ algorithms in Java at Microsoft. He believes that his use of Low SETL and the data structure selection transformation as part of a programming methodology increased his productivity and improved the quality of the code that was produced.

2.1.3. *Experimental Foundations For Productivity Improvement*

Perhaps the most compelling evidence that our transformational methodology will scale up and provide a dramatic improvement in the productivity of large high performance complex systems may be found in the experiments by Cai and Paige [12]. In that paper we developed a simple but conservative model of productivity. Within that model we demonstrated a five-fold improvement in productivity of high performance algorithm implementation in C in comparison to hand-coded C.

Those experiments tested an approach to producing C programs by writing programs in a simple variant of Low SETL, and compiling them into high performance C. The high performance of the C code produced by the SETL-to-C translator is based on the translator’s ability to simulate associative access (e.g. finite set membership or finite map application) on a RAM in real time.

Measuring productivity improvement depended on two assumptions. The first is that one line of Low SETL takes no more time to compose than one line of C. This assumption is not controversial. Our experience is that one line of Low SETL can be produced faster than a line of C. The generally lower level of discourse in C creates an intellectual gap between the program and the mathematical function it computes. For example, in order to implement SETL’s element deletion operation ($s \text{ less} := x$) efficiently in C would require at least 10 carefully chosen C operations. The need to access data through pointers and cursors in C creates a greater level of indirection (which complicates understanding) than in SETL, where data is accessed directly through values.

The second assumption is that the C code generated automatically from Low SETL has roughly the same number of lines as equivalent hand-coded C. We found that the generated C was between 10% and 30% larger than hand-coded C, and concluded that the errors in the two assumptions would cancel each other out.

Suppose we measure programming productivity in a given programming language as the number of pretty-printed source lines produced per unit time. Suppose also that productivity decreases as the conceptual difficulty in understanding a program grows. Then in our investigation, which is restricted to highly algorithmic programming (as is found in complex language systems and environments), conceptual difficulty is roughly reflected in the size of the dataflow relation, which

can be expected to grow nonlinearly with the number of source lines. Thus, programming productivity $P(L)$, as a function of the number of program source lines L , increases as L decreases.

Let L_2 be the size of a C Program C_2 compiled from a Low SETL specification S_1 of size L_1 , and let C_3 be a hand-crafted C program equivalent to C_2 . By assumption one, we can use the same productivity function $P(L)$ for programs written in both Low SETL and C. By assumption two, we know that the size of C_3 is roughly the same as the size L_2 of C_2 . Then the improvement in productivity by using our automated program development methodology versus hand-crafted programming is given by the time $L_2/P(L_2)$ to manually produce C_2 divided by the time $L_1/P(L_1)$ to manually produce S_1 , which is

$$(L_2/L_1)(P(L_1)/P(L_2)) > L_2/L_1$$

since $P(L_1)/P(L_2) > 1$ whenever $L_1 < L_2$.

It should be emphasized that our experiments did not measure productivity directly, which would have required difficult and costly analysis of human factors such as programming expertise and intelligence. Instead we measured productivity improvement by exploiting the two assumptions mentioned above to obtain an objective, inexpensive, and credible framework for conducting comparative productivity experiments that could avoid all human factors. The ratio of lines of C code generated automatically from Low SETL divided by the number of lines of Low SETL being compiled gives a lower bound on productivity improvement. Every algorithm that we tested in [12] yielded ratios that exceeded 5, and that grew as the input size grew.

Although we found that the C code generated automatically from Low SETL had runtime performance comparable to good hand code (whose running time, excluding I/O, was at least 30 times faster than SETL2 running time executed by the standard SETL2 interpreter), only small-scale examples were used. The largest C program was about 10 pages. Low SETL lacked procedures, and the type system was highly restricted. The SETL-to-C production rate was too slow – about 24 lines of C per minute on a SPARC 2. Finally, high level SETL input had to be translated into low level C input at compilation time.

The hypothesis that productivity improvement will scale up by our methods is based on the methodology and the application domain. Real-time simulation is applied uniformly to each instruction of a program regardless of program size. Program analysis and transformation algorithms have a complex combinatorial nature, which fits our model of productivity.

2.2. TRACK (1) SCALEUP OBJECTIVES

In order for Track (1) to be useful in developing systems, the three specification languages need to be reconstructed. The foundation for this reconstruction is a full-scale design of Low SETL. To this end we plan to enrich the functionality of Low SETL and extend its type system. Formal semantics for Low SETL need to be worked out along with the type system as it was in the earlier typed SETL variant found in [7]. And in order to support computational transparency, we need to incorporate complexity assertions within a formal rule system along the lines of Goyal and Paige [24].

In [40] tagged alternation, user defined types, and recursive subtypes were proposed for Low SETL. Polymorphism and higher order functions are also needed. Thus far, we have successfully modelled list-like data structures in the type system. The next step is to show how the type system can be used to model more realistic hybrid data structures built up from arrays and lists. We believe that our solutions might benefit the implementation of vectors in Java.

We also need to develop a type inference model that augments the model found in [7] and the inference algorithm found in the SETL-to-C translator [12] to handle the new type system. The powerful batch read feature found in [40] allows external input in string form to be validated and converted to complex data structures in linear time in the length of the string for any list of variables with any signature in the type system. This needs to be augmented with interactive input/output. Modules, procedures, and type conversions across procedure and module boundaries need to be added too.

Perhaps the main unresolved problem in implementing Low SETL has to do with its copy/value semantics. This is a hard problem that has been the major source of inefficiency in two generations of SETL compilers. The strategy of SETL1 [48] and SETL2 is to implement lazy copies. That is, assignment of large objects (e.g. sets and tuples with arbitrary depth of nesting) or incorporation of a large object into another large object is implemented by only copying pointers. Since a large object may be shared under this strategy, it is first copied whenever it is updated in order to avoid any side effect. With this approach hidden copies can potentially degrade program performance from $O(f(n))$ expected time to $O(f(n)^2)$ actual time. In [12] we observed 30,000-fold slowdowns in SETL2 performance due to unnecessary hidden copies.

Schwartz [45, 47] developed an interesting but complicated value flow analysis for SETL1 [48] in order to detect when hidden copies could be avoided, and update operations could be performed in place. The difficulty of the analysis seems to stem, in large part, from the fact

that SETL1 did not implement reference counts, so that the analysis had to prove that an r-value was unshared. It was never implemented, and a completely different naive approach that was eventually used proved to be unsatisfactory.

In SETL2 dynamic reference counts of all references (these are used for default boxed implementations, and are not part of the SETL2 language) to each tuple or set value are maintained. Highly restricted circularity of references ensures that when a value has a reference count greater than 1, then that value is *shared*, and cannot be updated unless it is first copied. In order to perform element addition $S \text{ with:}:= a$ (which adds element a to set S) in place, the location that stores S must have a reference count of 1 and be different from the location that stores a . Otherwise, a hidden copy of S is made, and the update is performed on the copy. Unfortunately, the backend of Snyder's SETL2 compiler introduces so many compiler-generated temporary variables (which don't get garbage collected until the end of scope) that practically all data is shared at runtime.

In [36, 12], we solved this problem by assuming that all updates were performed on unshared values, and so, could be updated in place. Any program that violated this assumption was considered 'erroneous' (in the sense of Ada). This approach obtained some credibility in [12, 24], where SQ2+ and High SETL specifications were transformed into Low SETL programs guaranteed not be erroneous a priori. However, this approach may not be satisfactory for manual programming directly in Low SETL in scaled up applications. Recently, Goyal and Paige solved this problem using dynamic reference counts, dead code analysis, and analysis of when large data MUST share the same location [25]. A very local implementation of this approach for SETL2 was shown to speedup APTS runtime by a factor of 10. However, dynamic reference counts do incur a constant factor overhead in running time, so it would be interesting to see if more powerful and complicated analysis of when data MAY share the same location could be used in order to detect a wide range of contexts where dynamic reference counts can be avoided.

Development of a robust Low SETL specification language would serve as the foundation for the other two higher level specification languages. In [7] variants of High SETL and SQ2+ were formulated within a simple type system, which was only crudely associated with complexity in an ad hoc way. We propose to reconstruct both specification languages with an enriched type system based on Low SETL.

Scaled up applications planned during the grant period include a reimplementing of APTS modules for pattern matching and inference in High SETL to be compiled into C. The batch read algorithm will be written in High SETL, translated into C, and benchmarked relative

to the hash-based SETL2 read method. Finally, we plan to implement Willard's RCS queries in High SETL to be compiled into C.

Developing better scientific means of measuring productivity and productivity improvement are important but extremely difficult. The model of productivity improvement developed in [12] was necessarily ideal, and only used for the Low SETL specification language. We would like to explore how to extend this model in order to test productivity improvement in C for High SETL and other specification languages.

2.3. TRACK (2) BACKGROUND

2.3.1. *Specification Language and Implementation*

Track (2) has to do with the methodology needed to implement Track (1). RSL (Rule Specification Language) is a high level language for specifying language translators, analyzers, and program transformations used in Track (1). RSL specifications are compiled and executed using the APTS meta-transformational system, which was built by Cai and Paige to implement this methodology [37]. Appendix A explains the methodologies of Tracks (1) and (2) by illustrating an actual APTS transcript of automatic program development using the SETL-to-C translator.

APTS was designed to implement complex program transformations and program analysis for ANY deterministic context free language. It is a collection of modules, each performing an independent task implemented by an interpreter for a different language paradigm, including logic-based inference, conditional rewriting, finite differencing, commands, and syntax. RSL is a single integrated transformational language with programming-in-the-large features such as an Ada-like library, separate compilation, and fine-grained incremental compilation. APTS is entirely written in only 15,000 lines of SETL2 source code (including comments), and it contains no foreign tools. It can be extended by call-in and call-out capabilities relative to compatible SETL2 modules.

Compilers written in RSL make use of the following APTS components. The Rule Database (RDB) contains inference rules that define relations storing program properties, e.g. type or dataflow. These relations may be defined over a variety of domains including conventional domains (booleans, integers, strings), the domain of abstract program points (i.e., simple contexts), program terms (the syntactic values that occur at program points), and Lisp-like S-expressions (a general domain used to embed arbitrarily deeply nested types amenable to first order unification). Inference rules are specified in a language similar to Datalog [52] augmented with function symbols and primitive pattern

matching predicates. The APTS inference engine analyzes the program (being compiled) for properties specified in the RDB, and it outputs the Program Database (PDB) of finite relations (sets of tuples that represent ground terms) storing the program's properties.

The Transformation Database (TDB) contains conditional meaning-preserving program transformations of two kinds – rewriting or finite differencing. The transformation engine selects a transformation T by matching T with a portion of the program, and by ensuring that the applicability condition for T , when instantiated with the current PDB, is satisfied. It then applies transformation T to the program to obtain a new transformed program. Consequently, the PDB must be updated to be consistent with the new program and the RDB.

The inference and transformation engines make use of the efficient bottom-up pattern matching algorithm of Cai, Paige, and Tarjan [13]. The inference engine used to calculate RDB relations [6] combines this bottom-up pattern matcher with RETE style pattern matching [22] and seminaive evaluation of Datalog [2, 1, 52]. Part of the signature of a relation allows seminaive evaluation to calculate relations as in a simple addition system or with built-in unification.

The use of APTS as a crucial vehicle in the proposed research may be justified simply by convenience – we have access to its source code, and the source language is SETL2, which is not hard to rewrite into Low SETL. However, its functionality also compares favorably with other systems. The APTS logic-based inference method for program analysis implementation was influenced by the earlier Mentor and Centaur systems [18, 5], which were among the first systems to break away from attribute grammars and use a more general logic-based approach to program analysis. Reliance on foreign tools such as Lex, YACC, and Prolog makes Centaur powerful but inefficient. Unlike Centaur's reliance on Prolog's general-purpose inference engine, our implementation has been designed specifically for high performance program analysis.

The goal of using incremental computation as part of a transformational environment for APTS was influenced by the Synthesizer Generator [43]. It uses a first-order functional language called SSL to specify syntax, attribute equations, as well as program transformations. In the SG, program analysis is performed by attribute evaluation and incremental attribute evaluation relative to program editing modifications [44]. Although efficient attribute re-evaluation is an important benefit of attribute grammars, this approach limits navigation to the syntax tree, which makes global analysis inefficient. Thus, it is often the case that an escape from the attribute grammar formalism is warranted, and C procedures are introduced.

Although RSL includes a command language, which, like SSL, can operate directly on syntax trees, the RSL subcomponents for logical inference, conditional rewriting, and built-in finite differencing are more abstract and declarative. The RSL style discourages direct reference to the syntax tree or its structure, and encourages reference to the tree indirectly by pattern matching. The algorithms that implement those subcomponents are highly sophisticated; e.g., the on-line preprocessing algorithm for multi-pattern tree matching [13].

Refine [41] is a robust, commercially available transformational system, that offers a language like SSL for manipulating syntax trees. KIDS [49] (which is built on top of Refine) implements rewriting and inference on top of Refine, and is highly regarded as a well-engineered system in the transformational community. The directed inference mechanism used by KIDS uses the full power of first order theorem proving, which is needed for program synthesis tasks. However, this approach trades efficiency and automation for generality, and is probably not well suited to scaled up applications, where automatic program analysis is essential.

2.4. TRACK (2) SCALEUP OBJECTIVES

The most pressing and perhaps most difficult open problem in APTS system research is in devising an automatic scheme to maintain the consistency of PDB relations incrementally after a program is changed by transformation. The idea is to automatically generate an INCREMENTAL RULE DATABASE (IRDB) from the BATCH RDB and a program transformation. This would allow the inference engine to recalculate the PDB efficiently by executing the IRDB each time a program transformation is applied. Some combination of differential techniques and partial evaluation is needed to solve this crucial but difficult research problem.

The current version of APTS handles this problem automatically for extensional relations such as *monotone* and *type* (which are invariant with respect to equivalence-preserving expression replacement). However for intensional relations such as *free* and *bound* variables, *control flow*, and most subtype relations, APTS requires the user to annotate each conditional rewriting rule R with instructions on how to update the PDB each time R is applied.

Solving this problem fully automatically would eliminate one of the most difficult and error-prone aspects of RSL programming. Of course, a good algorithmic solution must depend on a clean formal semantics, leading to the practical integration of analysis and transformation. More generally, it would also enhance a top-down stepwise refinement

framework in which global analysis is reserved for the highest level, perspicuous specifications, and local analysis is sufficient to select and justify transformations whose application propagates semantic facts to lower level implementations.

The only related work we are aware of is that of Emma van der Meulen [53], who gave initial solutions to incremental conditional rewriting for the ASF+SDF system [29], a descendant of Centaur. Her methods were based on reducing primitive recursive schemes to strongly noncircular attribute grammars, and either applying the Reps, Teitelbaum, Demers algorithm [44], or using a batch-oriented approach with lazy incremental updates. We would be seeking a sharper, more practical solution within our RETE-based inference strategy that exploits the fact that a transformation preserves semantics.

Another major problem has to do with ensuring reliability of the program development process. Currently, all transformations used within APTS preserve program semantics, but this is only proved on paper outside the system. Since APTS is capable of unbridled production of dangerously large quantities of code without any human intervention or oversight, the absence in APTS of machine-assisted meta-level support to prove transformations correct is a major deficiency that needs to be overcome. We expect that unfunded participants in the area of computational logic from the U. of Catania and the U. of L'aquila will investigate how to verify our transformations by mechanical proof checking and theorem proving. Recently, formal verification of our implementation design of Willard's query processor [24] has been achieved with paper and pencil in Cantali's Thesis [14] at the U. of Catania. Cantali's immediate future plans are to pass his proofs through the ETNA set theoretic mechanical proof checker.

Within the proposed research we plan to consider two other aspects of formal correctness. First, we will derive APTS system components that implement pattern matching and inference. Second, we plan to investigate various forms of rule induction to prove properties of RSL inference and conditional rewriting rules that implement particular kinds of program analysis and transformation.

The main objective of Track (2) research is to improve the speed of RSL execution in scaled up applications. Unless otherwise stated, our analysis will not include running time for input and output methods. Slow speed results from levels of interpretive overhead, the fact that RSL is fully specified in its own formalism, interfacing between modules, and the fact that APTS is implemented in SETL2. Slow speed of SETL2 results from dynamic memory allocation, dynamic typing, the high cost of redundant expressions, a model of computation based on associative access, hidden copies, etc.

In order to speedup RSL execution, we will reconstruct the existing SETL-to-C translator (written in RSL) into a robust compiler, called the Low SETL-to-C Accelerator, for the full Low SETL language. Next, we will rewrite APTS in Low SETL, and compile it into C using the SETL Accelerator. Based on previous experiments [12], this should speedup APTS by a factor of 30.

We will also build a self-applicable partial evaluator for Low SETL, written in Low SETL itself. RSL will be used to generate the Low SETL abstract syntax tree for the partial evaluator. This would allow us to partially evaluate the Low SETL version of APTS together with any RSL specification to produce a seamless Low SETL program equivalent to the original RSL specification, but running about 10 times faster. We could then compile this Low SETL specification into C using the Low SETL Accelerator to gain another factor of 30 in speed. In this way we could speedup the RSL specification of the SETL Accelerator by a factor of 300.

Since RSL syntax is fully specified in the RSL syntax formalism, the power of RSL can sometimes be used to analyze and implement itself more succinctly than in SETL. In this regard, we plan to replace the current APTS SETL modules for finite differencing and for RSL compilation by more perspicuous RSL modules. We also plan to make RSL statically typed, and to equip the RSL compiler with an RSL specification for type analysis. Partial evaluation would then be used to undo the extra level of interpretive overhead that would otherwise make this kind of bootstrapping unreasonably inefficient.

Partial evaluation [27] may be the most widely applicable and potentially practical transformation around. It is based on a general software engineering principle in which highly parameterized programs are concretized by simplification when a subset of the parameters is fixed. It offers an attractive generic implementation strategy that traces a portion of the computation for which expressions can be evaluated (for any datatype and operation in the language). Consequently, partial evaluation must be implemented in full accordance with language semantics. A partial evaluator can turn an interpreter with fixed program input into a compiled program, or it can turn a partial evaluator with fixed interpreter input into a compiler. Amazingly, it can turn a partial evaluator with fixed partial evaluator input into a compiler generator. Partial evaluation has been worked out for various programming languages within the major language paradigms.

An off-line partial evaluator for a low level subset of dynamically typed SETL2 has been built with a finite, uniform, congruent division and polyvariant specialization based on the method for flowchart languages found in [27]. It includes interprocedural analysis for control

flow, dataflow, and live variables, which is used for various optimizations including procedure unfolding during specialization. So far the partial evaluator has been used successfully to implement the First Futamura Projection on several interesting examples, including an interpreter for a simple imperative language. Minor modification is underway to implement the Second Futamura Projection, which is critical for automatic transformation of the APTS interpreter into a compiler.

However, the partial evaluator has been implemented using a mixture of SML and SETL2 programs. We need to rewrite it more perspicuously as a mixture of RSL and Low SETL specifications. The partial evaluator needs to be extended to handle modules (including module variables of static extent). It also needs to be rewritten to partially evaluate Low SETL. Finally, it should be said that designing a good self-applicating partial evaluator for Low SETL may be extremely difficult, and alternative approaches known to work and achieve the same ends will be taken if need be.

One possible alternative to get a good generating extension is to directly implement a compiler generator (cogen, also known as generating extension generator) [32], whose functionality is the same as the result of double self-application of a partial evaluator as formulated by the Third Futamura Projection. This new approach avoids several technical problems of self-applicating approach, which arise in statically typed languages such as Low SETL. The relationship of a compiler generator and a partial evaluator is like that of a compiler and an interpreter, so it is not too difficult to write a compiler generator by hand.

In addition, it is also difficult to design a good binding-time analysis, especially for separately compiled modules (which is an open problem in itself). For the specific application here, however, manually annotating and refining the binding-time of variables and operators would be a practical alternative, giving the developers a finer control over the generated code.

3. Research Plan for Automated Software Manufacturing

Within the grant period we intend to turn the rudimentary ideas found in [12] and outlined above into a practical technology offering a 5- to 10-fold improvement in productivity for large-scale examples of over 100,000 lines of C. In order to accomplish the preceding goals, we will combine *data structure selection* by real time simulation of a set machine on a RAM [36, 7] and *partial evaluation* [27]. We plan to use the software components described in the preceding section to automatically build other components that would be the essential tools for a

new practical program development technology. These new components are described below.

Let P_L be a program P implemented in language L . Let R denote RSL, and let S stand for Low SETL implemented in the standard runtime environment of dynamically typed SETL2. Using the transformational products in the project, including the SETL Accelerator written in RSL (denoted by A_R), the APTS system written in Low SETL (denoted by $APTS_S$) and the partial evaluator for Low SETL written in Low SETL (denoted by PE_S), we propose a series of transformational experiments that, in scaled up applications, test the feasibility of combining partial evaluation with our own transformations (sometimes in complex reflective ways), test speedups predicted for transformed code, and test productivity improvement for C implementations.

From experience we have found that the runtime performance of any program P_R is roughly 10 times slower than the runtime performance of an equivalent S program P_S . We would like to test whether partial evaluation of RSL yields similar speedups. Based on the experiments reported in [12] and unpublished independent experiments by Snyder (the designer and implementer of SETL2), SETL2 programs P_S should run 30 times slower than equivalent C programs P_C . (We assume here, that P_S would have no hidden copies, or else it might run many more times slower.) Our experiments [12] showed that the SETL-to-C translator produced C codes that matched the 30-fold speedup observed in hand-coded C. Since partial evaluation and data structure selection are completely independent, we plan to test the hypothesis that P_C will run 300 times faster than P_R regardless of whether P_C is produced mechanically or by hand.

We will first apply A_R to PE_S to give us a partial evaluator of Low SETL in C, *i.e.* $PE_C = A_R(PE_S)$, which should be 30 times faster than PE_S . Next, we will apply PE_C to $APTS_S$ and specialize it w.r.t. A_R to give us a Low SETL Accelerator in Low SETL, *i.e.* $A_S = PE_C(APTS_S, A_R)$, which should be 10 times faster than A_R . Self-application of A_S produces an equivalent translator $A_C = A_S(A_S)$ from S to C that is written in C and should run 300 times faster than A_R .

The next set of experiments relate to generating a fast generic C read method. First, we apply A_C to the read method written in Low SETL, denoted by $read_S$, to give us a generic C read method, *i.e.* $read_C = A_C(read_S)$. To get a specialized version of the read method for a fixed type signature sig (which includes a type assignment for the input variables and subtype constraints), we first apply PE_C to $read_S$ and sig to obtain a Low SETL version of the specialized read routine

for signature sig , i.e. $read_S^{sig} = PE_C(read_S, sig)$. We further apply A_C to $read_S^{sig}$ to get its C version equivalent, i.e. $read_C^{sig} = A_C(read_S^{sig})$.

In a similar way, we can improve the speed for APTS. Applying A_C to $APTS_S$ will yield a C version of APTS, i.e. $APTS_C = A_C(APTS_S)$. We get a compiler version of APTS by specializing PE_S w.r.t. $APTS_S$ using PE_C , i.e. $CAPTS_S = PE_C(PE_S, APTS_S)$. This can further be converted to C , i.e. $CAPTS_C = A_C(CAPTS_S)$. Now, our transformations can be done quickly using $CAPTS_C$ and A_C , i.e. $P_S = CAPTS_C(P_R), P_C = A_C(P_S)$ for any program P_R .

Among the tools generated for free from the three implemented tools PE_S , A_R , and $APTS_S$, we believe that $APTS_C$, $CAPTS_C$, and even A_C are likely to exceed 100,000 lines of C . Furthermore, programs P_C that result from compiling substantial RSL programs P_R into Low SETL programs P_S by $CAPTS_C$, which are further translated into C by A_C , can easily form codes of 100,000 lines or more.

A successful production of $APTS_C$ would, for the first time, yield C modules implementing efficient forms of extremely difficult algorithms and subsystems that would be useful to the programming language community. These include, (1) the fastest known preprocessing algorithm for bottom-up multi-pattern tree matching [13], (2) an inference engine combining our fast pattern matching algorithm with RETE-style forward chaining [22] to implement logic-based program analysis and computation, and (3) a bottom-up conditional rewriting engine that makes use of our fast pattern matching algorithm to implement source program transformation.

As a final application Willard's Predicate Retrieval theory [54, 55, 56] deals with a large database query optimization, and serves as an attractive scaled up application for implementing and verifying a difficult query compiler. Conceptually and technically difficult, an RCS query compiler has resisted all previous attempts at an implementation. Nevertheless, we believe that our implementation plans will succeed. RCS queries expressed in High SETL will be transformed into semantically equivalent programs in Low SETL using an RSL specification W_R . We expect the speed for this transformer to be considerably improved by the transformation $W_C = A_C(CAPTS_C(W_R))$. The Accelerator A_C will turn Low SETL versions of these queries into C for execution.

4. Results From Prior NSF Support

The present grant proposal stems from an ongoing feasibility study sponsored by the National Science Foundation under the SGER program within the Software Engineering and Languages area of CISE/CCR.

The ongoing NSF grant has award number CCR-9616993, funding amount \$100,000., and support period Sep. 1, 1996 to Aug. 31, 1999. The title is “Improving Productivity of Algorithm and System Implementation in Scaled Up Applications”.

Results from the currently funded grant have been reported earlier in this proposal. They include considerable progress in the design of Low SETL, in the implementation of a SETL partial evaluator, and in the investigation of the hidden copy problem. Publications include a POPL 97 paper [40] on the formal semantics and algorithmic development of a batch reading method for Low SETL. This paper was coauthored by the Principal Investigator and Ph.D. student Zhe Yang, and presented at POPL by Yang. As a result of this work Yang won a prestigious BRICS fellowship to study last year with Olivier Danvy at the U. of Aarhus in Denmark. This year he has returned to NYU to work on his Thesis.

While at BRICS, Yang worked on the type encoding problem in languages with Hindley-Milner type system. This work is partly motivated by the effort to type the generic read routine for Low SETL [40]. This routine has its input arguments dependently typed; i.e., one of the arguments provides the type signature for the remaining inputs. Results of this work were reported in an ICFP 98 paper [57] that was presented by Yang. The ICFP 98 paper formulated this kind of problem in terms of type-indexed values, and developed several general approaches to program with them within a Hindley-Milner type system, the basis for many popular functional languages such as ML. These approaches are based on encoding types as higher-order polymorphic functions, whose types reflect the encoded types themselves. For the general functional programming community, this paper provides programming techniques for writing dependently-typed programs using commonly available languages. We are also further convinced that a Hindley-Milner type system with some variations can provide the type basis for Low SETL.

As an application of the above type encoding paper, we solved the problem of implementing type-directed partial evaluation *natively* in ML [58]. Although native implementations of type-directed partial evaluators have been reported to be several magnitudes faster than meta-language implementations, previously, they were only implemented using untyped languages such as Scheme, which cannot guarantee run-time type safety.

Another publication [24], coauthored with Ph.D. student Deepak Goyal, demonstrated how the Low SETL type system could actually be used to improve the runtime complexity of Willard’s database query processing method. This work is unusual in the sense that a difficult

algorithmic improvement was not obtained by the usual combinatorial arguments but by algebraic and logical reasoning using theoretical programming language concepts. Goyal has given invited talks on this work at SUNY Albany, the U. of Copenhagen, and the U. of Aarhus. As noted earlier, these also inspired a Bachelor's Thesis in the area of Mechanical Verification at the U. of Catania in Italy. This year Goyal and I coauthored another paper [25] on how to avoid hidden copy operations in an imperative language with large datatypes and copy/value semantics with a lazy copy strategy, such as Low SETL. Goyal presented the work at SAS 98 (which took place in Pisa, Italy), and gave invited talks at several universities in Italy.

Goyal is the unique holder of the prestigious Dean's Dissertation Fellowship in the Computer Science Department at NYU, so he will not need academic support for next year. Zhe will be funded by an ONR grant that partly overlaps with the current proposal. This proposal seeks funding for one Ph.D. student in addition to Goyal and Yang for a total of three students working full-time on the project. The principal investigator will also be on sabbatical next year, and will remain at NYU in order to make good progress with the ambitious work proposed here.

5. Conclusion

There is compelling experimental evidence that SETL Accelerator A_C will compile programs P_S into C codes P_C whose performance is comparable to hand-coded C for small examples. We also believe that the performance of C codes P_C produced by our methods will actually grow in competitiveness with hand-coded C as the number of source C lines grow. If this is confirmed, then our automated software manufacturing technology may not only work for scaled up applications, but may allow us to build high performance systems that cannot even be built under current technology.

An important feature of this technology is the natural way in which it can evolve as each of its separate basic components (PE_S , A_R , or $APTS_S$) is improved. Improvements are of two kinds, both of which lead to a merger of Tracks (1) and (2) as part of a bootstrapping process. One kind of improvement has to do with progressive elevation of the partial evaluation language, the Accelerator source language, and the implementation languages of PE_S and $APTS_S$ from Low SETL to High SETL to either SQ2+ or RSL. Another kind has to do with combining the logic-based relational style of RSL with the func-

tional set-based style of SQ2+ into a single very high level specification language.

Appendix

A. Transcript of an APTS Derivation

This section illustrates our specification languages and transformational methodology by tracing through an actual transcript of mechanical program development in APTS. The transcript demonstrates how a formal specification of live code analysis (for an imperative well structured programming language), written in only a few lines of SQ2+, can be turned automatically into a C program of several hundred lines with worst-case running time and space linear in the input space. We also show how live code analysis can be specified in RSL, and used as part of the SQ2+-to-C compiler.

The SQ2+ specification of live code analysis appears pretty-printed by APTS just below:

```

program useless ;
1  assume oneone ( instof ) ;
2  assume onemany ( iuses ) ;
3  assume anyone ( compound ) ;
4  assume disjoint ( range instof , range compound ) ;
5  read ( instof , usetodef , iuses , compound , crit ) ;
6  print ( clfp ( crit ,
               live +
               instof [ usetodef [ iuses [ live ] ] ] +
               compound [ live ] , live ) ) ;
end ;

```

where *crit* is a set of initial live statements (read and print), *iuses* is a one-to-many map from statements to uses of variables, *usetodef* is a many-to-many map from uses to definitions (left-hand-side occurrences of variables) of variables that can reach these uses along definition-clear program paths, *instof* is a one-to-one map from definitions to their enclosing statements, and *compound* is a many to one map from statements to immediately enclosing compound statements (i.e., if-statements and while-loops). Assumptions are given to improve the quality of the compiled code. Image set expression $f[s]$ yields the image of set s under binary relation f ; i.e., $\{y: \exists x \in s \mid [x,y] \in f\}$. The conditional least fixed point expression *clfp*... computes the smallest set *live* (with respect to set containment) that includes set *crit* and satisfies equation $live = live \cup instof[usetodef[iuses[live]]] \cup compound[live]$.

For example, in the following program,

```
program test;
  read(b);
  if c2 then c := y;
  elseif c3 then a := y;
  end if;
  print (a);
end program;
```

let statements `print(a)` and `read(b)` belong to set `crit`. Then all but assignment `c := y` would be live.

Based on the SQ2+ type inference system described in [7], the APTS inference engine computes types for each program expression. It also computes other program properties, such as monotone expressions, and bound and free variables of expressions. These properties are all stored as relations in the the APTS PDB (Program Database).

The binary ‘type’ relation is defined over the domain of program terms (the first component) and s-expressions representing types (the second component). The binary relation ‘mono’ is defined over program terms. RSL code for two of the inference rules used to compute types are shown below,

```
match(%expr, .x%) | null(z, type(%expr, .x%, z)) ->
  bind(.t, newatom(t)) and type(%expr, .x%, .t);
match(%expr, .x + .y%) | type(%expr, .x%, .t) ->
  type(%expr, .x + .y%, .t) and type(%expr, .y%, .t);
```

The first rule states that for every program point `p` in the SQ2+ specification with syntactic category ‘`expr`’ (i.e., for every occurrence of an expression) such that ‘`type`’ is not defined for the term `r` stored at `p`, create a new type variable `t`, and make `t` the type for `r` (i.e., store pair `[r,t]` in the ‘`type`’ relation). The second rule is more complicated. It states that for every program context `p1 + p2`, where `p1` and `p2` are program points containing terms `t1` and `t2` respectively, if the ‘`type`’ relation contains pair `[t1,t]`, then perform the following actions. Obtain a most general unifier `t'` of `t` and the types of `t2` and `t1+t2` if these types already exist, and let `t' = t` otherwise. Then store pairs `[t1,t']`, `[t2,t']`, and `[t1+t2,t']` in relation ‘`type`’ after deleting each of the types for `t1`, `t2`, and `t1+t2` that may exist.

Inference rules in APTS may be performed in any order. There is a notion of ‘safe’ rules (under a closed world assumption) to allow for limited forms of negation and built-in predicates, and a semantics similar to the logic databases found in [52]. More details about the inference rules and how they are implemented in APTS can be found in [6].

The conditional rewriting transformation ‘nminfp’ can turn the functional live code analysis specification above into a simple imperative program that computes the conditional least fixed point by dominated convergence. The transformation is displayed in APTS using the ‘help’ feature.

```

>:help nminfp;
lhs code to be matched
  print ( clfp ( .w , .s + .k , .s ) ) ;
rhs replacement code
  .x := .w ;
  while exists .z in ( .k - .x ) loop
    .x with := .z ;
  end loop ;
  print ( .x ) ;
pattern expansion code is
  readvar ( .x ) and
  genvar ( .z ) and
  subst ( .k , .s , .x )
database action code is
  type(.x , [set, .t ]) and
  type(.z, .t) and
  type(%expr, .k-.x%, [set, .t])
enabling predicate
  mono ( % expr , .s + .k %, .s ) and
  type ( .s , [ set , .t ] )

```

This rule applies to any program point that is matched by the left-hand-side pattern and that satisfies the enabling predicate. Nonlinear pattern matching is carried out between the left-hand-side pattern (essentially a sentential form in the SQ2+ grammar) and the program. Variables preceded by a period are pattern variables that match program points. For matching to succeed, all occurrences of the same pattern variable in a pattern must match the same terms.

An environment *env* that maps pattern variables to program points is created as a side effect of matching. Matching the left-hand-side of ‘nminfp’ with the SQ2+ specification results in the following environment:

```

env(.w) = crit
env(.s) = live
env(.k) = instof[usetodef[iuses[live]]] + compound[live]

```

Next, the enabling predicate pattern is instantiated relative to environment *env*. Predicate `mono(live+instof[usetodef[iuses[live]]] + compound[live], live)` and `type(live, [set, .t])` which results from sub-

stitution, must then be matched against the mono and type relations stored in the PDB. That is, the pair $[\text{live} + \text{instof}[\text{usetodef}[\text{iuses}[\text{live}]]] + \text{compound}[\text{live}], \text{live}]$ must be stored in relation *mono*, and relation *type* must store pair $[\text{live}, [\text{set}, \tau]]$ for some s-expression τ that represents a type. Both conditions are satisfied. Since variable *live* has type $[\text{set } t5]$, the environment is extended so that $\text{env}(t) = t5$.

At this point the environment is extended further by executing the pattern expansion commands. Command *readvar* (*.x*) requires the user to supply an identifier from the terminal. In response we will supply the string *livest*, after which $\text{env}(x) = \text{livest}$. Command *genvar* (*.z*) creates a new identifier automatically. In this case the system supplies *x1*, after which $\text{env}(z) = x1$. Command *subst* (*.k*, *.s*, *.x*) is a general substitution mechanism that replaces all occurrences of $\text{env}(s)$ in $\text{env}(k)$ by $\text{env}(x)$. In this case, substitution will result in $\text{env}(k) = \text{instof}[\text{usetodef}[\text{iuses}[\text{livest}]]] + \text{compound}[\text{livest}]$.

The right-hand-side replacement code in the ‘*nminfp*’ transformation can now be instantiated relative to environment *env*, and used to replace the SQ2+ subtree matched by the left-hand-side. Finally, the database action code indicates how to update the type relation in the PDB for the new terms that are introduced by tree replacement. The High SETL program just below results from dominated convergence,

```

program useless ; --Assumptions elided
5 read ( instof , usetodef , iuses , compound , crit ) ;
6 livest := crit ;
7 while exists x1
    in instof [ usetodef [ iuses [ livest ] ] ] +
    compound [ livest ] - livest loop
8   livest with := x1 ;
   end loop ;
9 print ( livest ) ;
   end ;

```

The SQ2+-to-C translator proceeds to the next phase of compilation, which applies finite differencing (see [20, 21, 49, 33] for related work). In order to expose opportunities for finite differencing and also to regularize the program into a simplified form for which a limited number of finite differencing rules can have wide utility, the translator turns the program into a normal form. This is done by applying a group of conditional rewriting rules exhaustively bottom-up until there is no further change. Consequently, line 7 is replaced by the following equivalent code:

```

7 while exists x1
    in { x2 in instof [ usetodef [ iuses [ livest ] ] ]
        + compound [ livest ] | x2 notin livest } loop

```

The algorithm used to implement group transformations is based on the incremental linear pattern matching preprocessor found in [13]. Regardless of the number of rewriting transformations belonging to a group, matching can proceed bottom-up so that the exact subset of individual rules whose left-hand-sides match a given program subtree can be computed in unit time, and presented in linear time in the subset size. Efficient incremental preprocessing of groups with respect to rule addition and deletion is also supported.

Next, the finite differencing transformation automatically detects invariants of the form $x = e$ (where e is a program expression and x is a new variable uniquely associated with e) that should be maintained and exploited in order to avoid the costly repeated calculation of the truth set at the top of the while-loop at line 7. The following invariants are detected automatically by bottom-up analysis of the truth set expression; their left-hand-side variables are all supplied by the user:

```
uses = iuses [ livest ]
defs = usetodef [ uses ]
livedf = instof [ defs ]
livecf = compound [ livest ]
liveall = livedf + livecf
work = { x in liveall | x notin livest }
```

After the six invariants are detected, a stream processing transformation based on Goldberg and Paige [23] inserts code that aims to establish the invariants on entry to the while loop using a minimal number of loops. In this case stream processing generates the code appearing on lines 7 through just before line 29 below. A chain rule is used to perform the necessary bookkeeping operations needed to reestablish these invariants just before they are falsified by the modification to `livest` at line 45. Consequently, the costly truth set expression at the top of the while loop is made redundant, and is replaced by variable `work`. The Low SETL program that results from finite differencing appears just below.

```
program useless ; --Assumptions elided
5   read ( instof , usetodef , iuses , compound , crit ) ;
6   livest := crit ;
7   livecf := { } ;
8   livedf := { } ;
9   defs := { } ;
10  uses := { } ;
11  for x3 in livest loop
12    if compound ( x3 ) notin livecf then
13      livecf with := compound ( x3 ) ;
      end if ;
```

```

14   for x5 in iuses { x3 } loop
15     for x9 in usetodef { x5 } | x9 notin defs loop
16       livedf with := instof ( x9 ) ;
17       defs with := x9 ;
18       end loop ;
19     uses with := x5 ;
20   end loop ;
21   work := { } ;
22   liveall := { } ;
23   for x11 in livedf loop
24     if x11 notin livest then
25       work with := x11 ;
26     end if ;
27   liveall with := x11 ;
28   end loop ;
29   for x12 in livecf loop
30     if x12 notin livest then
31       work with := x12 ;
32     end if ;
33   liveall with := x12 ;
34   end loop ;
35   while exists x1 in work loop
36     if compound ( x1 ) notin livecf then
37       if compound ( x1 ) notin livest then
38         work with := compound ( x1 ) ;
39       end if ;
40     liveall with := compound ( x1 ) ;
41     livecf with := compound ( x1 ) ;
42   end if ;
43   for x6 in iuses { x1 } loop
44     for x9 in usetodef { x6 } | x9 notin defs loop
45       if instof ( x9 ) notin livest then
46         work with := instof ( x9 ) ;
47       end if ;
48     liveall with := instof ( x9 ) ;
49     livedf with := instof ( x9 ) ;
50     defs with := x9 ;
51   end loop ;
52   uses with := x6 ;
53   end loop ;
54   if x1 in liveall then
55     work less := x1 ;

```

```

        end if ;
45     livest with := x1 ;
        end loop ;
46     print ( livest ) ;
    end ;

```

Finite differencing often causes intermediate invariants and code in the untransformed program to become useless. The next step of the SQ2+-to-C translator performs a dead code analysis and elimination transformation. First inference rules are used to compute control flow, data flow, types, and an abstraction of the program that allows live code to be calculated. These consist of around 3 pages of RSL specifications, including the following two axioms to determine the initial live statements:

```

match(%statement, print(.x);%)
-> live(%statement, print(.x);%);

```

```

match(%statement, read(.x);%)
-> live(%statement, read(.x);%);

```

Next, the following two inference rules

```

--a use of variable z is live if it occurs free
--in expression y that is reached from a definition of z
--along a path clear of other definitions to z
reach(.z, .y) and freevar(.y, .z) -> live(.z);
--a program component that encloses a live subcomponent
--is also live
live(.x) and neq(pred(.x), nil) -> live(pred(.x));

```

are used to determine those program statements that contribute either directly or indirectly (via dataflow and control flow) to the print statement at line 46. This step is carried out in logic programming fashion by calculations involving only relations in the PDB and without reference to the program syntax tree. Analysis determines that statements at lines 10, 19, 42, and 44 are dead. These statements are removed from the program, and control structures are subsequently simplified using RSL conditional rewriting rules.

The final transformation, which makes use of concepts found in [46], but relies mainly on the type/subtype inference mechanism of [7], implements all set and map datatypes using array and list data structures. This transformation rests on the discovery of finite universal sets, called *bases*, to be used for data sharing and for creating aggregate data structures that serve to simulate associative access (e.g. x in s) in real-time; i.e., each such access is implemented by a unit time array or pointer

access. The final C code with 342 labeled statements runs in time linear in the size of the usetodef relation. Similar to the benchmarks reported in [12], it runs 30 times faster than the SETL2 program running under Snyder's 'stlx' interpreter, which indicates performance comparable to good hand-coded C.

References

1. F. Bancilhon. Naive evaluation of recursively defined relations. In M. Brodie and J. Mylopoulos, editors, *On Knowledge-Base Management Systems*, pages 165–178. McGraw-Hill, 1986.
2. R. Bayer. Query evaluation and recursion in deductive database system, 1985. unpublished manuscript.
3. B. Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages*. PhD thesis, Massachusetts Institute of Technology, Sept. 1989.
4. B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Science of Computer Programming*, 24(3):189–220, 1995. <http://cs.nyu.edu/cs/faculty/paige/papers/readysc.ps>.
5. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. *Rapports de Recherche 777*, INRIA, 1987.
6. J. Cai. A language for semantic analysis. Technical Report 635, Courant Institute, New York University, 1993.
7. J. Cai, P. Facon, F. Henglein, R. Paige, and E. Schonberg. Type transformation and data structure choice. In B. Moeller, editor, *Constructing Programs From Specifications*, pages 126–164. North-Holland, Amsterdam, 1991. <http://cs.nyu.edu/cs/faculty/paige/papers/subtype.ps>.
8. J. Cai and R. Paige. Binding performance at language design time. In *Proc. Fourteenth ACM Symp. on Principles of Programming Languages*, pages 85 – 97, Jan. 1987.
9. J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1988/1989. <http://cs.nyu.edu/cs/faculty/paige/papers/fixpoint.ps>.
10. J. Cai and R. Paige. Languages polynomial in the input plus output. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology*, Workshops in Computing, pages 287–302. Springer-Verlag, 1992. Conference Record of the Second AMAST.
11. J. Cai and R. Paige. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1-2):189–228, July 1995. <http://cs.nyu.edu/cs/faculty/paige/papers/hash.ps>.
12. J. Cai and R. Paige. Towards increased productivity of algorithm implementation. In *Proc. ACM SIGSOFT*, pages 71–78, Dec. 1993. <http://cs.nyu.edu/cs/faculty/paige/papers/prod.ps>.
13. J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106(1):21–60, Nov. 1992. <file://cs.nyu.edu/pub/tech-reports/tr604.ps.Z>.
14. A. Cantali. "Using ETNA to prove correctness and complexity of a linear time implementation of a subset of Willard's RCS. Bachelor's thesis, University of Catania, Catania, Italy, 1997.

15. C.-H. Chang and R. Paige. From regular expressions to DFA's using compressed NFA's. *Theoretical Computer Science*, 178(1-2):1–36, 1997. <http://cs.nyu.edu/cs/faculty/paige/papers/cnnfa.ps>.
16. P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific J. Math.*, 82(1):43–57, 1979.
17. H. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.
18. V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang. Programming environments based on structured editors: the mentor experience. In *Interactive Programming Environments*. McGraw-Hill, 1984.
19. W. Dowling and J. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *J. Logic Programming*, 1(3):267 – 284, 1984.
20. J. Earley. High level iterators and a method for automatically designing data structure representation. *J. of Computer Languages*, 1(4):321–342, 1976.
21. A. Fong and J. Ullman. Induction variables in very high level languages. In *Proc. Third ACM Symp. on Principles of Programming Languages*, pages 104–112, Jan. 1976.
22. C. Forgy. Rete, a fast algorithm for the many patterns many objects match problem. *Artificial Intelligence*, 19(3):17–37, 1982.
23. A. Goldberg and R. Paige. Stream processing. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 53–62. ACM, 1984.
24. D. Goyal and R. Paige. The formal reconstruction and improvement of the linear time fragment of Willard's relational calculus subset. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 382 – 414. Chapman & Hall, 1997. http://cs.nyu.edu/phd_students/deepak/lrcs.ps.
25. D. Goyal and R. Paige. A new solution to the hidden copy problem. In G. Levi, editor, *Proc. 5th International Static Analysis Symposium*, number 1503 in LNCS, pages 327–348. Springer, September 1998. http://cs.nyu.edu/phd_students/deepak/copy.ps.
26. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, 146:29–60, Dec. 1969.
27. N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
28. J. Keller and R. Paige. Program derivation with verified transformations – a case study. *CPAM*, 48(9-10):1053–1113, 1996. <http://cs.nyu.edu/cs/faculty/paige/papers/ltmjform.ps>.
29. P. Klint. The asf+sdf meta-environment user's guide, version 26. Technical report, Centrum voor Wiskunde en Informatica, 1993.
30. D. Knuth. *The Art of Computer Programming, 3 Volumes*. Addison-Wesley, 1968-1972.
31. S. Koenig and R. Paige. A transformational framework for the automatic control of derived data. In *Proc. 7th Intl. Conf. on VLDB*, pages 306–318, Sep 1981.
32. J. Launchbury and C. K. Holst. Handwriting cogen to avoid problems with static typing. In C. K. H. R. Heldal and P. Wadler, editors, *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 210–218, Skye, Scotland, 1991. S-V.
33. Y. Liu. Principled strength reduction. In R. Bird and L. Meertens, editors, *Algorithmic Languages and Calculi*, pages 357 – 381. Chapman & Hall, 1997.
34. R. Paige. *Formal Differentiation*. UMI Research Press, 1981.

35. R. Paige. Programming with invariants. *J IEEE Software*, 3(1):56–69, Jan 1986.
36. R. Paige. Real-time simulation of a set machine on a ram. In N. Janicki and W. Koczkodaj, editors, *Computing and Information*, volume II of *ICCI 89*, pages 69–73. Canadian Scholars' Press, Toronto, May 1989. <http://cs.nyu.edu/cs/faculty/paige/papers/realtime.ps>.
37. R. Paige. Viewing a program transformation system at work. In M. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic*, volume 844 of *LNCS*, pages 5–24. Springer-Verlag, Berlin, Sep. 1994. <http://cs.nyu.edu/cs/faculty/paige/papers/viewing.ps>.
38. R. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient ram code - a case study. *Journal of Symbolic Computation*, 4(2):207–232, Aug. 1987.
39. R. Paige, R. Tarjan, and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theoretical Computer Science*, 40(1):67–84, Sep. 1985.
40. R. Paige and Z. Yang. High level reading and data structure compilation. In *Proc. 24th ACM Symp. on Principles of Programming Languages*, pages 456 – 469, 1997. http://cs.nyu.edu/phd_students/zheyang/papers/read.ps.
41. Refine user's guide version 3.0, 1990.
42. J. Reif and H. Lewis. Symbolic evaluation and the global value graph. In *Proc. 4th Annual ACM Symp. on Principles of Programming Languages*, pages 104–118, 1997.
43. T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York, 1989.
44. T. Reps, T. Teitelbaum, and A. Demers. Incremental context-dependent analysis for language-based editors. *ACM TOPLAS*, 5(3):449–477, 1983.
45. J. Schwartz. Automatic data structure choice in a language of very high level. *CACM*, 18(12):722–728, Dec. 1975.
46. J. Schwartz. Automatic data structure choice in a language of very high level. *CACM*, 18(12):722–728, Dec 1975.
47. J. Schwartz. Optimization of very high level languages, parts I, II. *J. of Computer Languages*, 1(2-3):161–218, 1975.
48. J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.
49. D. Smith. Kids - a semi-automatic program development system. *IEEE Transactions on Software Engineering*, pages 129–136, Sept 1990.
50. K. Snyder. The SETL2 programming language. Technical Report 490, Courant Insititute, New York University, 1990.
51. A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific J. of Mathematics*, 5:285–309, 1955.
52. J. Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1988.
53. E. van der Meulen. *Incremental Rewriting*. PhD thesis, CWI, 1994.
54. D. E. Willard. Predicate retrieval theory. Technical Report 83-3, SUNY Albany, 1983.
55. D. E. Willard. Quasi-linear algorithms for processing relational data base expressions. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 243–257, 1990.
56. D. E. Willard. Applications of range query theory to relational data base join and selection operations. *J. Computer and System Sci.*, 52:157–169, 1996.

57. Z. Yang. Encoding types in ML-like languages. In P. Hudak and C. Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300, Baltimore, Maryland, USA, Sept. 1998. ACM Press.
58. Z. Yang. A native ML implementation of type-directed partial evaluation. In O. Danvy and P. Dybjer, editors, *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98*, (Göteborg, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Notes Series, BRICS, Department of Computer Science, University of Aarhus, May 1998.

Appendix

A. Biographical Sketch

Robert A. Paige

Department of Computer Science

Courant Institute

251 Mercer St.

New York, NY 10012

email address: paige@cs.nyu.edu

WWW home page: <http://cs.nyu.edu/cs/faculty/paige>

office phone : 212-998-3156

Employment

1993-present Professor of Computer Science Courant Institute/NYU

1995 (Summer) Summer Faculty, BRICS Center, Univeristy of Aarhus, Denmark

1985,1986,1988(Summers) Summer Faculty IBM Yorktown Heights;

Education

1979 Ph. D., Computer Science, Courant Institute, New York University

Dissertation: "Expression Continuity and the Formal Differentiation of Algorithms;" Adviser: Prof. J. T. Schwartz.

Awards and Honors

- 1995** BRICS VIP, 4 Invited Lectures, "Analysis and Transformation of Set-Theoretic Languages," BRICS, U. of Aarhus, Denmark, Aug. 1995.
- 1995** Invited speaker at Symp. Celebrating the Scientific Trajectory of Jack Schwartz, New York City, May 1995.
- 1994** Invited speaker for Joint 6th Intl. Conf. on Programming Language Implementation and Logic Programming (PLILP) and 4th Intl. Conf. on Algebraic and Logic Programming (ALP), Sept., 1994, Madrid, Spain.
- 1990** Honorary Fellow, WARF Foundation Award at University of Wisconsin.
- 1990** Invited speaker for ESOP '90, May, 1990, Copenhagen, Denmark.
- 1981** Dissertation selected for publication as a book in UMI Research Press outstanding Computer Science dissertation series.

5 Publications Most Closely Related To Project

1. Paige, R. and Yang, Z., "High Level Reading and Data Structure Compilation," Proc. 24th POPL, Jan. 1997, pp. 456 - 469.
2. Goyal, D. and Paige, R., "The Formal Reconstruction and Improvement Of The Linear Time Fragment Of Willard's Relational Calculus Subset," in Algorithmic Languages and Calculi, Richard Bird and L.G.L.T. Meertens (Eds.), pp. 382 - 414, Chapman & Hall, London, 1997.
3. Cai, J. and Paige, R., "Towards Increased Productivity of Algorithm Implementation," Proc. ACM SIGSOFT 1993, pp. 71 - 78, in Soft. Eng. Notes, Vol. 18, Num. 5, Dec. 1993.
4. Cai, J. and Paige, R., "Binding Performance At Language Design Time," Proc. 14th ACM POPL, Jan. 1987, pp. 85 - 97.
5. Paige, R. and Koenig, S., "Finite Differencing of Computable Expressions," ACM TOPLAS, IV(3), July, 1982, pp. 401-454; earlier version appeared as LCSR-TR-8, Rutgers U., New Brunswick, NJ, Aug. 80 [Revised Dec. 81].

5 Other Significant Publications

1. Goyal, D. and Paige, R., "A New Solution To The Hidden Copy Problem," In Proceedings of the 5th International Static Analysis Symposium, LNCS 1503, ed. G. Levi, pp. 327-348, Pisa, Italy, Sept. 1998.
2. Chang, C. and Paige, R., "From Regular Expressions to DFA's Using Compressed NFA's," Theoretical Computer Science, vol. 178(1-2), May, 1997, pp. 1-36.
3. Cai, J. and Paige, R., "Using Multiset Discrimination To Solve Language Processing Problems Without Hashing," Theoretical Computer Science, vol 145, Num 1-2, July, 1995, pp. 189-228;
4. Cai, J., Paige, R., and Tarjan, R., "More Efficient Bottom-Up Multi-Pattern Matching In Trees," Theoretical Computer Science, Vol 106, Num 1, Nov. 1992, pp. 21 - 60; [Special issue of best papers selected from CAAP '90];
5. Cai, J. and Paige, R., "Program Derivation by Fixed Point Computation," Science of Computer Programming, Vol 11, No. 3, 1988/1989, pp. 197 - 261;

List of Collaborators Within the last 48 Months

C.-H. Chang (Institute for Information Industry, Taipei, Taiwan), Jiazhen Cai (ATT, NJ), Robert Tarjan (Princeton U., NJ), Christoff Hoffmann (Purdue U., IN), Bard Bloom (IBM, NY), Nils Klarlund (ATT, NJ), J.-P. Keller (Kepler, Paris, France), Deepak Goyal (NYU, NY), Zhe Yang (NYU, NY);

List of Thesis Advisees and Postdocs Sponsored Within the Last 5 Years

Postdocs(2 total): Mooly Sagiv (U. of Tel Aviv, Israel), Jiazhen Cai (ATT, NJ);

Sponsored or Advised Students(7 total): C.-H. Chang (Institute for Information Industry, Taipei, Taiwan), Jacob Rehof (Microsoft, Redmond WA), Mihnea Marinescu (NYU, NY), Archisman Rudra (NYU, NY), Chung Yung (NYU, NY), Deepak Goyal (NYU, NY), Zhe Yang (NYU, NY);

Graduate Advisors

Ph.D. Advisor: Jacob Schwartz, NYU, NY