A Research Retrospective

Bob Paige

September 19, 1999

1 WG 2.1

The group was particularly exciting in the 1970's, when we were groping for direction and divided by different orientations. I guess it was in this atmosphere that combined purpose with uncertainty where I found my own voice. The common goal was a transformational program development methodology that would improve productivity of designing and maintaining correct software. The emphasis was on algorithmic software.

We differed as to how to achieve this goal, and my approach was out on a limb. Based on a few transformations, the most exciting of which was Jay Earley's iterator inversion [11] combined with high-level strength reduction, and also on an overly optimistic faith in the power of science to shed light on this subject, I believed that algorithms and algorithmic software could be designed scientifically from abstract problem specifications by application of a small number of rules, whose selection could be simplified (even automated in some cases) if it could be guided by complexity. Most all others (including the SETL crowd at Courant) disagreed, and accepted the notion that algorithm design was 'inspired', and that the most significant steps in a derivation were unexplainable 'Eureka' steps.

I knew that my goals were ambitious and with little supporting evidence. The case was too flimsy to begin work on the components of a program development methodology. It seemed better to gather facts first, to uncover compelling examples that might lead to a theory, and to put together this theory only after the pieces were sufficiently understood.

In order to test the viability of a new transformational idea, I tried to demonstrate how it could be used to improve some aspect of a 'conventional' area of CS; e.g., databases, algorithms, programming languages, etc. (but not transformational programming itself). A conservative, neutral test would be determined by an evaluation of the improvement (independent of the transformational idea) by members of the conventional area. A more subjective, but still useful clinical test could be made by me checking whether the new idea could improve the quality of education or facilitate my research. (I used the class room not only as a laboratory for clinical tests, but also as a way for me to learn and retool in new subject areas.) After developing the transformational idea further, it could be evaluated directly by the program transformation community, e.g., at WG 2.1 meetings, other conferences, or through the publication process.

2 SETL

Perhaps the most important component of a transformational methodology is a wide-spectrum language for expressing all levels of abstraction from problem specifications down to hardware-level implementations. Since I was a student at NYU with Jack Schwartz as my adviser, naturally I went with SETL [25], and found it convenient to use in my thesis work in finite differencing (from 1977-1979). As it turned out I was teaching full time (4 courses per year) at Rutgers starting Fall 1977, and tried lots of research ideas out in the class room early on.

Despite serious semantic flaws in SETL (some of which were fixed much later in SETL2 [26]), a small subset of the language, augmented with a few standard abstract notations found in the mathematical preliminaries chapters of standard CS texts, provided students with a simple, powerful, and sometimes executable mathematical notation. SETL's small but powerful repertoire of abstract operations proved convenient in being widely reusable in modeling a variety of computer-science concepts without the need for language extension.

In algorithms courses the builtin abstractions of SETL allowed data structures and rudimentary strategies of many algorithms to be specified perspicuously and without extraneous implementation detail. Default implementations of SETL's abstract operations allowed such specifications to be studied as executable prototypes, and tested empirically. Such specifications could also be used as the starting points for derivations of more efficient implementations. This made algorithm understanding easier, even (especially) with weak students who were overwhelmed by the detail of Mix-like or Algol-like implementation-level specifications. It seemed that this approach allowed me to cover much more material and convey much greater understanding.

In database courses SETL notations seemed better than the standard query languages. A Philips Research technical report by Lacroix and Pirotte, which compared over 100 DB queries in 9 different query languages plus English was instructive. SETL variants sketched out by Koenig and me seemed to have greater clarity. A uniform wide-spectrum notation facilitated description and analysis of file and database organization, and made it easier to describe mappings from query to physical level and from one data model to another. It facilitated teaching different data models, and different levels of description including implementations at the level of file systems and indices.

The wide-spectrum nature of SETL made it easy to illustrate program transformations at and between various levels of abstraction. The first transformation that I developed was finite differencing (a generalization of strength reduction plus Earley's Iterator Inversion), whose goal was to speedup programs by replacing costly repeated computations by less expensive differential counterparts.

3 Finite Differencing

Differencing is an old mathematical idea with a long history. The chain rule mechanism and other features of finite differencing presented in my thesis [18] can be described independently of any particular language, but the wide-spectrum nature and simplicity of SETL seemed ideal for illustrating the power and broad applicability of the transformation in a separate implementation design. Using SETL finite differencing, one can easily explain how to put together and analyze (using a natural notational form of worst case and amortized complexity) so very many efficient algorithms in a systematic way. This made teaching algorithms much easier.

One of the best examples of how surprising connections can be made, due to finite differencing, took place at the Chamrouse WG2.1 Workshop (1989) in France. We were all trying to derive one of those efficient max sequence algorithms, and the use of finite differencing at a major step suggested a similar step (to compute nested collective minimum values) in the derivation of Dijkstra's *Single Source Shortest Path* algorithm.

During these early years there was also one interesting objective test of finite differencing in the area of databases. In his Turing award address [9] Codd said that integrity control was an important open problem in relational databases. Koenig and I showed how finite differencing could be used to solve view maintenance and integrity control [17, 19]).

In compiler classes at Rutgers I also developed a crude form of dominated convergence in order to derive Hecht's workset algorithms [13] for solving global program analysis problems specified by fixed points. Work on this transformation progressed as it was seen to be progressively more useful in deriving an increasingly wider range of algorithms. And I continued to use it in combination with finite differencing in my compiler lectures in order to derive the more difficult algorithms. However, it was not until my collaboration with Cai that a comprehensive investigation and development of this transformation was first completed [5].

It was apparent from the beginning that I was looking for a transformational program development methodology whose final step would be data structure selection beyond which lay conventional compiler optimization. I also favored an approach limited to composable simple data structures as are found in SETL's method of basings and Wiederhold's file system design methodology [27] instead of an expert system style as proposed by Barstow [1] and Kant [14]. Despite some ingenious ideas, SETL data structure selection was largely ad hoc, not amenable to formal complexity analysis, over-relying on hashing, missing a linked list, and never proven (either analytically or empirically) to be an improvement over naive unoptimized data structures.

What was needed were data structures guaranteed to support associative access (e.g., set membership testing) in unit worst case time for search arguments and set elements of arbitrary type. Also needed were principles for selecting such data structures. It has taken many years to develop this transformation, and a first comprehensive treatment will appear in Deepak Goyal's thesis (expected in Fall 1999). Only recently was the transformation equipped with a suitable read method to create the desired data structures ([23]). And I still do not understand how to eliminate enough overhead explaining even very simple forms of data structure selection (described in several papers cited earlier) to use this transformation conveniently in a standard algorithms class.

Nevertheless, I believe that it is an essential final stage of and underpinning to a transformational methodology that (1) begins with dominated convergence to compute fixed point specifications, (2) segueds into finite differencing to implement repeated expensive high-level expressions more efficiently by exploiting the differential maintenance of program invariants, and, finally, (3) uses data structure selection by real-time simulation to implement associative access operations in unit worst case time on a RAM.

4 Assessment

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. Our SETL-based 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.

4.1 Explaining Complex Algorithms

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 [18, 21]. 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 SQ^+, a functional language composed from the expressions of SETL augmented with least and greatest fixed point expressions. It was also shown how to simplify these specifications, and to transform them by dominated convergence into high-level SETL prototypes [5]. Cai and I showed how the constant propagation algorithm of Reif and Lewis [24] could be expressed as set-theoretic equations in a subset of SQ^+ that could be mapped into RAM code guaranteed to run in linear time in the size of the program dataflow relation [4].

The use of notation has been regarded as a burden to algorithm designers ever since Knuth came out with Mix [16]. But can notation also help satisfy the needs of the algorithm community – precise algorithmic analysis and succinct exposition? I was thrilled that coauthor Bob Tarjan agreed to explain our new linear time solution to the Single Function Coarsest Partition Problem as being derived from an SQ^+ specification by dominated convergence, finite differencing, and real-time simulation; this explanation appears in a special issue of TCS containing a selection of the best papers from ICALP84 [22]. In [23] a much improved explanation of the algorithmic tool called Multiset Discrimination [6] was obtained. This was done by specifying the algorithm in low level 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 level SETL presentation clarified their understanding.

4.2 Discovering New Algorithms

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 [20], where we used all three transformations to turn an SQ^+ specification of Horn Clause Propositional Satisfiability into a new linear time pointer machine algorithm. Previously, Dowling and Gallier found a linear time algorithm ([10]) that relied heavily on array access. In [3] we used dominated convergence and finite differencing to derive a low level SETL executable prototype from an SQ^+ specification of ready simulation. We then showed informally how the low level SETL prototype could be turned into an algorithm that runs 5 orders of magnitude faster than the previous solution in [2]. 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 [15]. Finite differencing was used extensively in [8] 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 [12], where Goyal and I used low level 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 [28] 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. That is our 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 level 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.

4.3 Improving Productivity

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 [7]. 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 statically typed simple variant of low level 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 level 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 level 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 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 level 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 level 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 level 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 level SETL divided by the number of lines of low level SETL being compiled gives a lower bound on productivity improvement. Every algorithm that we tested in [7] yielded ratios that exceeded 5, and that grew as the input size grew.

Although we found that the C code generated automatically from low level 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 level 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. Algorithms for doing program analysis and transformation have a complex combinatorial nature, which fits our model of productivity.

5 Conclusion

Over the last few years, I've felt that, finally, there was enough evidence and know-how to put together much of our work into a formal program development methodology that could achieve my goals of 20 years ago. Although I was too sick to carry this work out, I'm happy that my student Deepak Goyal has done it in his Ph.D. thesis.

References

[1] D. Barstow. *Knowledge-Based Program Construction.* Elsevier North-Holland, 1979.

[2] B. Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-like Languages.* PhD thesis, Massachusets Institute of Technology, 1989.

[3] 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.

[4] 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.

[5] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1988/89.

[6] 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. URL http://cs.nyu.edu/cs/faculty/paige/papers/hash.ps.

[7] J. Cai and R. Paige. Towards increased productivity of algorithm implementation. In *Proc. ACM SIGSOFT*, pages 71–78, Dec. 1993.

[8] C.-H. Chang and R. Paige. From regular expressions to DFA's using compressed NFA's. *Theoretical Computer Science*, 178(1–2):1–36, 30 May 1997.

[9] E. F. Codd. Relational database: A practical foundation for productivity. In *CACM*, volume 25(2), pages 109–117, 1982.

[10] W. Dowling and J. Gallier. Linear time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[11] J. Earley. High level iterators and a method for automatically designing data structure representation. *J. of Computer Languages*, 1(4):321–342, 1976.

[12] D. Goyal and R. Paige. The formal reconstruction and speedup of the linear time fragment of Willard's relational calculus subset. In Bird and Meertens, editors, *Algorithmic Languages and Calculi*, pages 382–414. Chapman and Hall, 1997.

[13] M. S. Hecht. *Flow Analysis of Computer Programs,.* Elsevier, Amsterdam, 1977.

[14] E. Kant. *Efficiency in Program Synthesis.* UMI Research Press, 1981.

[15] J. Keller and R. Paige. Program derivation with verified transformations – a case study. *CPAM*, 48(9-10), 1995.

[16] D. E. Knuth. *The Art of Computer Programming, Vol 1: Fundamental Algorithms.* Addison-Wesley, 1973.

[17] S. Koenig and R. Paige. A transformational framework for the automatic control of derived data. In C. Zaniolo and C. Delobel, editors, *Conference on Very Large Databases*, pages 306–318, Cannes, France, Sept. 1981.

[18] R. Paige. *Formal Differentiation: A Program Synthesis Technique*. UMI Research Press, 1981. Revision of Ph.D. thesis, NYU, Jun 1979.

[19] R. Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In H. Gallaire, J. Minker, and J. M. Nicolas, editors, *Advances in Database Theory*, volume 2, pages 171–209. Plenum Press., New York, 1984.

[20] 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.

[21] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. on Programming Languages and Systems*, 4(3):401–454, 1982.

[22] 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.

[23] R. Paige and Z. Yang. High level reading and data structure compilation. In *Proc. 24th ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Lang.*, pages 456–469, Paris, France, 15–17 Jan. 1997.

[24] J. H. Reif and H. R. Lewis. *Symbolic evaluation and the global value graph.* Harvard University, Aiken Computation Laboratory, 1982.

[25] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.

[26] K. Snyder. The SETL2 programming language. Technical Report 490, Courant Insititute, New York University, 1990.

[27] G. Weiderhold. *Database Design*. McGraw-Hill, 1983.

[28] D. E. Willard. Predicate retrieval theory. Technical Report 83-3, SUNY Albany, 1983. PhD. Thesis.