Future Directions In Program Transformations[1]

R. Paige
New York University

As the emphasis of Programming Languages has evolved from computing on numbers to computing on programs, the area of program transformations has naturally gained in importance. This paper briefly surveys what transformational programming is about, and how to make progress in the field.

A program transformation is a meaning-preserving mapping defined on a programming language. Transformational programming is a program development methodology in which an implementation I is obtained from a specification S by applying a sequence $T_k...T_1$ of transformations to S. If S and each transformation T_i applied to successive implementations $T_{i-1}...T_1S$ of S are proved correct for $i = 1, ..., k$, then $I = T_k...T_1S$ must also be correct. If we assume that program development requires a proof obligation, then it is easier to develop a program by transformation than by conventional crafting if the labor involved in constructing and verifying S, and of deriving I by transformation is less than the cost of constructing and verifying I from scratch.

In an ideal transformational approach software is maintained at the specification level, and specifications are synthesized and verified by composition, integration, and enhancement; implementations are obtained by re-derivation. The cost of program development and maintenance drops if verified specifications, transformations, and derivations can be reused conveniently. Of central importance to this approach, a transformational programming system is envisioned to reduce labor costs and increase reliability by mechanizing much of the tedious pencil-and-paper calculations involved in constructing derivations.

The long range goal of transformational programming is to improve reliability, productivity, maintenance, and analysis of software without sacrificing performance. The benefits are expected to be greatest for complex algorithmic software, whose demand has been increasing for real time applications, design automation, operations research, genetic engineering, optimizing compilers, complex tools for workstations, etc. The inability of program development by conventional crafting to meet this demand has created an intensive interest in a more formal transformational approach. And the community of researchers contributing to program transformations has been steadily growing.

Testing the viability of transformational programming for practical software development is a formidable undertaking. Much of the evidence, both pro and con, that has been gathered over the last 20 years has been based on experiments deriving implementations of combinatorial algorithms (see [Paige et al. 1985; Paige et al. 1993;

[1]This research was supported by Office of Naval Research Grant No. N00014-93-1-0924 and by National Science Foundation grant CCR-9616993.

Jeuring 1994; Watson 1993]. The purpose has been to uncover common patterns of abstraction in specification and transformation that could form the basis of a useful methodology. Developing widely applicable methods leading to improved algorithm derivations has contributed to the science of programming and is beginning to shed light on the theory of algorithms. Short, elegant transformational proofs 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.

One research direction is to use program transformations to better integrate algorithm design and program development, so that algorithmic results can be more easily utilized in practical applications, and the power of language abstraction could be used to help discover and analyze algorithms. Solutions are likely to depend on useful ways to capture complexity in syntax, and to help guide the selection of transformations based on an accurate prediction of improvement in program performance. One direction would be to make practical use of program transformations (perhaps those that capture aspects of amortized complexity) for the development of efficient incremental software, which plays such a significant role in programming environments(e.g. [Reps and Teitelbaum 1989]), and other interactive systems. Another direction would be to study complex data structures such as fusion trees[Fredman and Willard 1994; Andersson 1995], complex priority queues[Brodal and Okasaki 1996], and persistent data structures[Driscoll et al. 1986], which have been shown to have computational advantages over more naive counterparts. It would be an interesting and worthwhile research project to explicate these data structures in terms of program transformations, and to facilitate their use in developing efficient programs. It would be most impressive to use transformations to actually improve these data structures or widen their range of application.

Transformational methodology rests on semantic foundations especially needed to determine meaning preservedness and equivalence for specifications, implementations, transformations, and derivations. Precise semantics are needed for defining and implementing abstract interpretations[Cousot and Cousot 1977] that are used in 'approximate' program analysis to justify transformations. For syntax-directed editing systems attribute grammar technology has been sufficient to facilitate semantic analysis and re-analysis relative to arbitrary editing changes. However, this technology seems inadequate for specification or efficient implementation of the more global analysis and re-analysis required in transformational systems. Another research direction would be to develop sound and efficient logic-based inference[Borras et al. 1987] and incremental inference integrated with semantics preserving transformations.

The means of production in any form of manufacturing must create trust in the product, whether by human oversight that occurs in any manual craft or by formal scientific guarantees when production is automated. Transformational systems may have the power to perform sophisticated program analysis, and to generate software by transformation at breakneck speed, but, to date, they are not sound. Lacking from them is a convenient mechanical facility to prove that each transformation preserves semantics. In order to create confidence in the products of transformational systems we need to prove correctness of specifications and transformations. Currently, this is too labor intensive to be practical[Pepper and Wirsing 1994; Keller and Paige 1995]. Progress in computational logic, in languages and systems that support the development and reuse of 'proofware' is a major challenge, that, I believe, must be overcome for a viable program development technology to emerge from a transformational approach.

The program transformation community may be best known for particular tranformational techniques, among which three are mentioned here – partial evaluation[Jones et al. 1993], finite differencing[Paige and Koenig 1982], and data structure selection by real-time simulation[Cai et al. 1991; Goyal and Paige 1997]. Partial evaluation 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 are fixed. It offers an attractive generic 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.

One problem with partial evaluation transformations is that they may fail to terminate. Overcoming this problem by practical heuristics would be generally useful, and may also make it easier to extend partial evaluation to concurrent languages such as CCS that have nonterminating constructs. Current implementation methods are trace-based, faithfully following the prescribed computation dictated by program syntax. It would be interesting to develop ways to preprocess programs to make them more amenable to the technique (as is done in database query processing, where optimization usually follows a complex decomposition phase). The benefits of partial evaluation due to expression simplification and elimination of interpretive overhead exhibit an interesting tradeoff that deserves further investigation. Simplification benefits higher level languages the most, while eliminating interpretive overhead benefits lower level languages the most.

Finite Differencing is a transformation that captures the algorithmic principle of program improvement by replacement of costly repeated calculations by less expensive differential counterparts. This transformation is not an artifact of any particular language, and it can be developed in an almost language-independent way based on the complexity of mathematical operations. It has been successfully adapted to various languages and language paradigms. The idea is seen in ad hoc form as the low level optimization by strength reduction[Cocke and Kennedy 1977; Ferrante and Ottenstein 1983; Cai and Paige 1995]. It has been formalized and used extensively in database applications to concrete view maintenance, integrity control, and trigger implementation[Koenig and Paige 1981]. The use of dynamic indexes to cut down on database search is another example. Finite differencing should be improved by a better understanding of its inherent space/time tradeoff (between maintaining invariants lazily by demand calculations only when they are needed (e.g. [Fong and Ullman 1976]) , and maintaining them eagerly at all times[Paige 1986]), and development of a preprocessing phase to rearrange function compositions to facilitate optimization.

Data structure selection by simulating associative access (e.g. set membership $x \in S$) on a RAM in real time is another algorithmic transformation. Based on a low level theory of data structures[Paige 1989], it has been used to form a compiler that translates a statically typed low level variant of $SETL$ into C[Cai and Paige 1993]. The type and subtype system[Cai et al. 1991] makes this variant of $SETL$ computationally transparent (to facilitate accurate formal analysis of resource utilization in well typed programs). Experiments with this compiler suggest that it can turn typed $SETL$ into C code whose runtime performance is roughly equivalent to good hand-coded C. Within a simple model of productivity we can also infer that typed $SETL$ programs can be manually constructed 5 times faster than equivalent C[Cai and Paige 1993].

Perhaps an ultimate test for viability would be to decide whether transformational methodology can be used to help implement its own tools efficiently, reliably, and with clean semantics. At New York University we are currently working on an experiment that combines partial evaluation with real-time simulation in order to speedup the translation of typed $SETL$ to high performance C.

Let P_L represent a program P implemented in language L. Let S stand for the typed variant of $SETL$ mentioned above, but implemented in the standard runtime environment of dynamically typed $SETL2$[Snyder 1990]. Let R represent RSL, a very high level language for specifying program analysis and transformations, which is implemented by $APTS$, an interpretive transformational programming system implemented in dynamically typed $SETL2$. Then our experiment depends on building the following three software components:

—a rewriting $APTS_S$ of $APTS$ in typed $SETL$,
—a robust reconstruction A_R of the current RSL translator from $SETL$ to C so that it translates all of typed $SETL$,
—a typed $SETL$ partial evaluator PE_S written in typed $SETL$.

Based on crude experimental evidence the runtime performance of any program P_R is expected to be at least 10 times slower than the runtime performance of an equivalent S program P_S. Based on more convincing experimental evidence, we know that P_S runs at least 30 times slower than an equivalent C program P_C.

Using the three preceding software components, we expect to speedup translator A_R in the following way. First, we partially evaluate $Aptss_S$ with fixed input A_R in order to produce translator $A_S = PE_S(Aptss_S, A_R)$ from S to C that is semantically equivalent to A_R but runs 10 times faster. 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 runs 300 times faster than A_R.

REFERENCES

ANDERSSON, A. 1995. Sublogarithmic searching without multiplications. In *36th Annual Symposium on Foundations of Computer Science* (Milwaukee, Wisconsin, 1995), pp. 655–663. IEEE.

BORRAS, P., CLEMENT, D., DESPEYROUX, T., INCERPI, J., KAHN, G., LANG, B., AND PASCUAL, V. 1987. Centaur: the system. Rapports de Recherche 777, INRIA.

BRODAL, G. AND OKASAKI, C. 1996. Optimal purely functional priority queues. *Journal of Functional Programming 6*, 6 (Dec.).

CAI, J., FACON, P., HENGLEIN, F., PAIGE, R., AND SCHONBERG, E. 1991. Type transformation and data structure choice. In B. MOELLER Ed., *Constructing Programs From Specifications*, pp. 126–164. Amsterdam: North-Holland.

CAI, J. AND PAIGE, R. 1995. Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science 145*, 1-2 (July), 189–228. URL http://cs.nyu.edu/cs/faculty/paige/papers/hash.ps.

CAI, J. AND PAIGE, R. Dec. 1993. Towards increased productivity of algorithm implementation. In *Proc. ACM SIGSOFT* (Dec. 1993), pp. 71–78.

COCKE, J. AND KENNEDY, K. 1977. An algorithm for reduction of operator strength. *CACM 20*, 11 (Nov.), 850–856.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Proc. 4th ACM Symp. on Principles of Programming Languages* (1977), pp. 238–252.

DRISCOLL, J., SARNAK, N., SLEATOR, D., AND TARJAN, R. 1986. Making data structures persistent. In *Proc. 8th ACM STOC* (May 1986), pp. 109–121.

FERRANTE, J. AND OTTENSTEIN, K. J. 1983. A program form based on data dependency in predicate regions (Austin, Tex., Jan. 1983). pp. 217–236.

FONG, A. AND ULLMAN, J. 1976. Induction variables in very high level languages. In *Proc. Third ACM Symp. on Principles of Programming Languages* (Jan. 1976), pp. 104–112.

FREDMAN, M. AND WILLARD, D. 1994. Surpassing the information theoretic barrier with fusion trees. *J. Comput. Sys. Sci. 48*, 533–551.

GOYAL, D. AND PAIGE, R. 1997. The formal reconstruction and improvement of the linear time fragment of Willard's relational calculus subset. In *IFIP TC2 Working Conf. on Algorithmic Languages and Calculi* (1997).

JEURING, J. 1994. The derivation of on-line algorithms, with an application to finding palindromes. *Algorithmica 11*, 2 (Feb.), 146–184.

JONES, N., GOMARD, C., AND SESTOFT, P. 1993. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

KELLER, J. AND PAIGE, R. 1995. Program derivation with verified transformations – a case study. *CPAM 48*, 9-10, 1053–1113.

KOENIG, S. AND PAIGE, R. 1981. A transformational framework for the automatic control of derived data. In *Proc. 7th Intl. Conf. on VLDB* (Sep 1981), pp. 306–318.

PAIGE, R. 1986. Programming with invariants. *J IEEE Software 3*, 1 (Jan), 56–69.

PAIGE, R. 1989. Real-time simulation of a set machine on a ram. In N. JANICKI AND W. KOCZKODAJ Eds., *Computing and Information*, Volume II, pp. 69–73. Toronto: Canadian Scholars' Press.

PAIGE, R. AND KOENIG, S. 1982. Finite differencing of computable expressions. *ACM Trans. on Programming Languages and Systems 4*, 3, 401–454.

PAIGE, R., REIF, J., AND WACHTER, R. Eds. 1993. *Parallel Algorithm Derivation and Program Transformation*. Kluwer.

PAIGE, R., TARJAN, R., AND BONIC, R. 1985. A linear time solution to the single function coarsest partition problem. *Theoretical Computer Science 40*, 1 (Sep.), 67–84.

PEPPER, P. AND WIRSING, M. 1994. Korso: A methodology for the development of correct software. Technical Report 93-36 (Nov.), TU Berlin.

REPS, T. AND TEITELBAUM, T. 1989. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, New York.

SNYDER, K. 1990. The SETL2 programming language. Technical Report 490, Courant Insititute, New York University.

WATSON, B. 1993. A taxonomy of finite automata minimization algorithms. Computing Science Note 93/44 (Dec.), Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands.