Transforming Experiences

James M. Neighbors
Bayfront Technologies, Inc.
1280 Bison B9-231
Newport Beach, CA 92626 USA
+1 714 436 0322
James.Neighbors@BayfrontTechnologies.com

ABSTRACT
This paper describes our experience with program transformation systems from 1974 to the present. It provides a rationale for how our use of the technique has evolved.

Keywords
Transformation, optimization, refinement, domain analysis

1 EXISTING PROGRAM IMPROVEMENT

In early 1974 I worked with Tim Standish on "Interactive Program Manipulation" to improve existing programs using source-to-source program transformations. We did not consider changing the meaning of the program. Our method was to apply correctness-preserving transformations one at a time. This work resulted in a catalog that listed low-level compiler-like optimizations aimed at an Algol-level programming language. A prototype source-to-source transformation system was built and it was populated with some of the transformations from the catalog. Very quickly we found it tedious to apply transformations one at a time.

2 EXISTING PROGRAM SPECIALIZATION

We wondered why bother performing these transformations at all? It was nice to see a clarified program but was really much gained? Well in one case we did see huge gains. That was when we added an external assumption to a program and then transformed it. The prototype source-to-source transformation system was renamed "Specialist" and we documented our experiences [4]. Source-to-source transformations are very powerful at customizing a general program under a specific case.

3 DOMAINS FOR PROGRAM SYNTHESIS

Many times with the Specialist system we had the feeling that we were dealing with the wrong level of abstraction. As an example, we could transform a matrix multiply specified as a series of loops into a matrix copy given the assertion that one matrix was the identity matrix. This specialization took over 100 low-level transformations. However at the "correct" level of abstraction, say that of APL, this would be a single transformation (e.g., A:=B*1 => A:=B). I did not believe that simple low-level transformations with a complex planning technology were the right idea. I believe that using higher levels of abstraction with simple techniques is the right idea.

In 1978 I started working with Peter Freeman to combine the ideas of systems analysis, automatic programming, software reuse, and transformations to specify and synthesize systems. There were four main groups pursuing this basic approach: Harvard [2], MIT [8], Stanford [3], and USC/ISI [1]. These projects were primarily using wide-spectrum descriptions where one notation suffices to describe a system from its problem domain to implementation. I opted to use narrow-spectrum languages I called *domains* and defined the idea of *domain analysis* to determine their contents [5,7]. I was hoping to get the "correct" level of abstraction from the domains. The prototype system I constructed was called *Draco* [6]. It manipulates a hierarchy of domains each with their own notations.

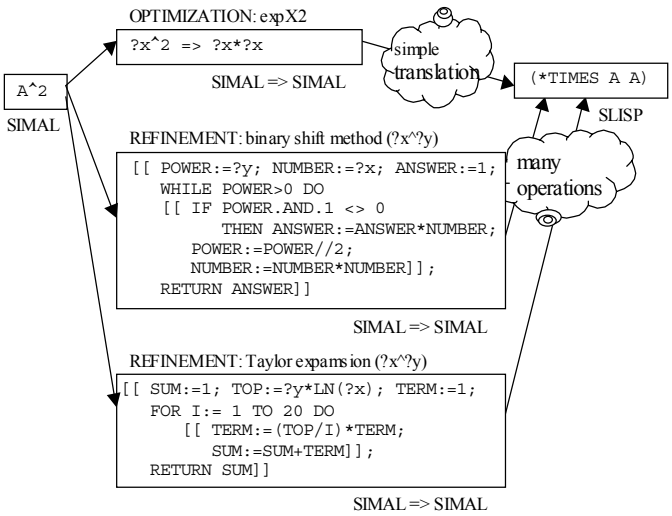

Figure 1. Transformation implementation of exponentiation

Figure 1 demonstrates the basic concepts of Draco. Assume a simple language SIMAL with exponentiation and another language SLISP without exponentiation. Given an instance

of exponentiation in SIMAL figure 1 provides our choices for refinement. We can use the instance context to optimize which is valid for all implementations if the enabling conditions are met. Alternatively, we can use one of two refinements if their individual enabling conditions are met. This is a hierarchy of two domains: SIMAL and SLISP. Our Specialist difficulty was analogous to instantiating the "binary shift expansion" and then using many low-level transformations to reduce it to a single multiply. Notice that the "Taylor expansion" should never reduce to the multiply. It *models* exponentiation and so no general set of equivalence preserving transformations should be able to reduce the expansion to a single multiply.

4 USING APPLICATION DOMAINS

In the late 1980s I studied very large commercial systems. They drove me to respect architecture and functional diversity. It goes without saying that the goal of system specification is to ultimately obtain a working system. However, during development many other needs must be met. As an example the input specification of a protocol can be used to generate working protocol code, simulation code, and analysis tool input data all from the same description. Each workproduct probably has a different architecture. This reinforces the ideas of *variable target systems* and *variable implementations*. In the 1990s Bayfront Technologies, Inc. tried these concepts out in the area of commercial communications protocols and we believe they were received well.

5 TRANSFORMATIONAL BELIEFS

Every transformation has enabling conditions. Even the most innocuous of transformations (e.g., ?A*(?B+?C) => (?A*?B+?A*?C)) have enabling conditions. Enabling conditions can help guide transformation because they can prune choices.

Metalevel planning is necessary. We saw the application of transformations form into patterns when we applied them one at a time. These patterns infer plans that drive these patterns. Applying individual transformations one at a time is tedious and error prone. Low-level tactics and high-level strategies are necessary.

Source-to-source transformations work well for specialization and optimization. It is easier to find and remove unused generalization than to generalize something that is specific. The latter requires the addition of knowledge not in the code being manipulated.

Transform at the correct level of abstraction. The "correct" level is a level where the concepts you are manipulating are directly represented. Once again do not try to infer any knowledge that's already been removed from the code.

Optimization and refinement are not the same thing. Optimization works for all implementations of the objects being manipulated. Optimization does not change the level of abstraction. Refinements make an implementation choice, change the level of abstraction, and are irreversible.

Use narrow-spectrum languages rather than wide-spectrum languages. Wide-spectrum languages include all of the formal theory languages and general specification languages. In order to achieve the "correct" level of abstraction as we have characterized it above, wide-spectrum languages will have to use abstraction notations. Once they do, they revert to narrow-spectrum languages with cumbersome notations.

Reduce transformation mechanism power to achieve useful strategic planning. If you use incredibly complex transformations, then it is hard understand what they do. Worse, it's hard to characterize what they do for strategic planning purposes.

Architecture evolves as a consequence of implementation techniques in addition to system function. There are many ways to implement the same construct. Inline code, threaded code, and threaded code interpreters are all valid schemes for implementing a single component.

Synthesis is easier than understanding. It's tempting to search for the right-hand side of a transformation and claim program understanding. However you must have enough knowledge to synthesize before you can recognize.

Domains provide education. This is the largest impact of this work. I never would have guessed it at the time.

REFERENCES

1. Balzer, R.M., Goldman, N.M., and Wile, D., On the Transformational Implementation Approach to Programming, *Proc. 2nd ICSE*, pages 337-344, 1976.

2. Cheatham, T.E., et.al, A System for Program Refinement, *Proc 4th ICSE*, pages 53-62, 1979.

3. Green, C., The Design of the PSI Program Synthesis System, *Proc. 2nd ICSE*, pages 4-18, 1976.

4. Kibler, D., Neighbors, J.M., and Standish, T.A., Program Manipulation via an Efficient Production System, *SIGPLAN Notices*, 12(8):163-173, 1977.

5. Neighbors, J.M., *Software Construction using Components*, PhD diss., UC Irvine, 1980 <http://www.BayfrontTechnologies.com/thesis.htm>.

6. Neighbors, J.M., *Draco 1.2 Users Manual*, UCI, 1983 <http://www.BayfrontTechnologies.com/manual.htm>.

7. Neighbors, J.M., The Draco Approach to Constructing Software from Components, *IEEE Trans. on Software Engineering*, **SE-10**, 5 (1984) pp. 564-574.

8. Rich, C., Schrobe, H.E., and Waters, R.C., Overview of the Programmer's Apprentice, *Proc. 6th IJCAI*, pages 827-828, 1979.