Program Transformation – What We Didn't Know

Martin S. Feather
Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive
Pasadena, CA 91109, USA
+1 818 354 1194
Martin.S.Feather@Jpl.Nasa.Gov

Back in the 1970s, the early advocates of program transformation emerged. I joined their ranks. I was inspired by Burstall and Darlington's so-called "folding" techniques for transformation of applicative languages. During the 1980s a small but lively community of us worked on program transformation. Workshops and conferences took place; working groups met; papers and books were published; program transformation systems were constructed. However, there were things about program transformation we didn't know. There were things about program transformation that we *thought* we knew, but didn't.

One of the most strikingly incorrect of our expectations concerned the *application* of program transformation. It was envisioned as being used to go from extremely high-level specification languages to extremely efficient code. It was in competition with conventional compilation techniques. Those techniques carefully limited the expressiveness of their input languages so as to assure the feasibility of (automatic) code generation. Program transformation did not. We were confident that program transformation could, and should, bridge a huge language gap. One way or another, we would guide the transformation process, and so achieve far more than could a purely automatic compiler. Bolstered by these high expectations, some of us became enamoured of ever-more expressive and high-level specification languages. In our wake we left the challenges of building transformations to work with those languages, and controlling the lengthy transformation process that would be needed to go all the way from specification to code.

The more successful applications of program transformation turned out to be those that, in some significant and carefully chosen way, limited their aspirations. The successes I knew best were those by Jim Boyle, and Doug Smith.

Boyle worked with specifications in known and straightforward languages (e.g., pure Lisp), and essentially translated the overall behavior of those specifications into some other language (e.g., Fortran). He would decompose the overall translation into a sequence of small steps, so that in the end it could be accomplished entirely automatically. He avoided the temptation of enriching his input language with new constructs, but, interestingly, would willingly introduce new constructs as intermediaries between input and target. He avoided the temptation of seeking to make use of every kind of optimization, but, interestingly, would plug in the right optimization steps as needed (e.g., to get target-machine-specific performance). Often he would focus on a specific application, for example, a cellular automaton used to solve partial differential equations.

Smith worked with a very constrained set of *solutions!* On each occasion, a single well-known algorithmic style would lie at the core of his solution, e.g., divide-and-conquer. His technique melds a problem specification with an algorithmic specification, and the outcome is a solution to that specification in that algorithmic style. Very elegant. Also, very useful, as it turned out, for a whole host of real-world problems. Scheduling-in-the-large emerged as a fertile area for his results. What looked like such a narrowly focused approach became much more than a scientific curiosity. In truth, the back-end of his approach is now supported by additional transformational activities - data structure selection, constraint propagation, etc. I guess I'm still surprised at how hard it really is - just how much knowledge one really has to bring to bear - to complete the pathway from specification to efficient program.

In many ways, the Y2K problem is the most extreme example of my point. It seems such a trivial problem - adjust some old programs that use two-digit dates to instead use four-digit dates. Hardly the grand challenge that formed the vision for us early program transformation advocates.

Another area, domain specific languages, is a success story. Perhaps this is one that some early visionaries did identify. I'll leave it to others, more knowledgeable than I, to provide insights into this.

What I have been observing more recently are numerous opportunities for using modest, smallish-scale transformation. Again, these quite don't fit the old grand vision. Just recently, I procedurally coded something that I should have done in a transformational style - a translator from constraints (input to a planner) into database queries (input to a test tool that independently checks that generated plans indeed meet their constraints). It was such a modest translation problem that I didn't think I needed to use a transformational approach, but soon came to wish I had! Right now I'm working with some other JPLers who are constructing a transformation-like system. Back in the '80s (oops, the 1980s) I would not have guessed the wide range of applications that we intend for this. Analysis and testing are major activities, so we are working on transformation from UML state diagrams to the input to a model checker. More traditionally, we also want to generate executable code from those diagrams. Both these transformation tasks should share common translation sub-components, not only to save our development time, but, more importantly, to ensure the analysis results correspond to the generated code. Yet another application is to translate to both code, and code to test that code (for people who don't yet fully trust our arguments as to why our transformations are perfect).

There are other aspects of transformation unknown in the early 1980's but that emerged over the years to come. Theory behind transformations (again, Smith's work springs to my mind in this regard). New kinds of transformations (e.g., "finite differencing", "memoization", "staging", "filter promotion"). Techniques to support transformation (e.g., efficient representations for manipulation of huge programs). Obviously we didn't know these, but we knew that things like these would emerge (and they should be remembered and reused, not rediscovered). However, I stick to my point that the one big thing I, and perhaps others, didn't know was today's many and varied applications of transformation.

ACKNOWLEDGEMENTS

Some of the research described in this paper is being carried out by the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.