Research Statement

Scott Draves
spot@cs.cmu.edu

February 27, 1997

My recent research concerns program transformation for signal processing. My thesis is that run-time code generation (RTCG) allows one to write pixel-level graphics in a very flexible high-level language without losing the performance of hand-specialized C. Furthermore, the notions of *binding times* (temporal types) and *compiler generation* from current research in partial evaluation (PE) provide portable and accessible means to implement efficient procedures that create procedures. My dissertation *Lightweight Languages for Interactive Media* describes how I tested my thesis by building prototype systems and benchmarking common kernels.

High performance graphics systems in C implement many special cases—for example to support media with different numbers of bits per pixel, different alignments in memory, or optional channels such as transparency and depth. Unless a particular combination is directly implemented in one routine, its operation will require either conditionals inside an inner loop, or buffers to hold partial results. Conditionals reduce bandwidth; buffers increase latency and memory traffic. RTCG bypasses these problems by generating special cases and fused, one-pass loops *as needed*.

We face three questions: can we produce good enough code? Is the time spent generating this code worth its speed-up? Is the programmer-time spent learning RTCG worth its speed-up? My dissertation examines the first question by building two prototype systems and measuring a collection of examples.

Until 1991 my main research was *fnord*, a system for visualizing differential geometry in teaching and research. As in many such systems, you can describe a surface or curve by typing-in equations and by defining and using symbolic operators. But beyond the ordinary, you can attach a GUI slider to a constant, drag the slider, and see the shape change in real time. *fnord* achieved this dramatically improved performance with an interpreted data-parallel language—essentially by using batching to reduce interpreter overhead. Despite their success, all batched systems such as *fnord* suffer from increased memory traffic, increased latency, and limited data-dependent control flow. The only proven solution to this problem is run-time code generation.

If RTCG is the solution, then what is the problem, and how does it relate to partial evaluation and languages? RTCG has traditionally suffered from a lack of portable, easy-to-use interfaces. CommonLISP's backquote/comma/eval is vastly easier to use than existing C solutions, but the memoization required to produce programs with loops and the lack of variable hygiene are sources of bugs and duplicate code. Scheme's `syntax-rules` and upcoming `syntax-case` address hygiene but not control flow. Partial evaluation uses binding-time analysis, essentially a static type system, to divide a one-stage interpreter into a two-stage compiler. Thus the programmer can write and debug ordinary code, then by tweaking and annotation, convert it to use RTCG. The programmer concentrates on high-level issues such as staging and generalization rather than bookkeeping.

While *nitrous* has demonstrated the potential of PE for graphics, the prototype is quite primitive and contains a number of design errors. Even using it in my own projects is not feasible. My next step in this direction is to build a tool that satisfies myself and attracts a community of users. Of course I will immediately put this tool to use in various graphics applications such as a C-callable 2D graphics library, a mouse/paint/draw program, net-quality-adaptive telepresence, or my artwork. Beyond that, these questions bear examination:

1. How do the benefits of RTCG and specialization depend on the hardware architecture? Are the benefits greater on DSP, FPGA, and VLIW chips relative to general purpose CPUs?

2. How does the programmer effort of manual, CommonLisp-style RTCG compare to automatic, PE-style? Compared to macro systems such as R5RS Scheme or Tim Sheard's meta-programming system? Can we integrate or hybridize these techniques?

3. Beyond sample-level media-processing, what application areas can benefit? First targets include: rendering, modeling, and shading in 3D-graphics; music/audio processing; adaptive compression; operating systems and network-mobile code; and various parts of AI such as artificial life and genetic algorithms.

More generally, I am interested in building programming tools, where programming is widely construed. I include everything from an Scheme integrated developement environment (IDE) to Director's Lingo to After Effects' GUI to hypervideo to RenderMan. Because languages are appearing at an ever-increasing rate (witness the web's HTML, JavaScript, Java, and VRML2), in the longer term I am especially interested in language creation and tighter integration between languages.

My research and experience with computers spans graphics, languages, and partial evaluation. The underlying theme is a question: how can we build tools fostering the creative exploration of visual spaces? My frustration with existing graphics systems and my fascination with language-centric engineering has led me from making pictures to making programmable graphics systems to metaprogramming in general.