Can partial evaluation improve the performance of ray tracing?

Kenichi Asai

Department of Information Science, Faculty of Science, Ochanomizu University
2-1-1 Otsuka Bunkyo-ku Tokyo 112-8610 Japan
`asai@is.ocha.ac.jp`

Abstract

This short paper reports our experience on using runtime partial evaluation to improve the performance of ray tracing. By exploiting constant information on objects and light during the ray tracing process, we can produce a specialized ray tracer which runs up to two times faster than the unspecialized version. Thanks to the integration of a partial evaluator into an interpreter, specialization is achieved simply by inserting a `pe` command. Although the implementation is still prototypical, the experimental results show that partial evaluation does have a role in ray tracing.

1 Partial evaluation

Partial evaluation (or program specialization) is a program transformation technique which improves efficiency of a program by performing as much computation as possible *before* all the inputs become available [8]. For example, consider the following power function written in a functional programming language Scheme [9].

```
(define (power m n)
  (if (= n 0)
      1
      (* m (power m (- n 1)))))
```

If the value of `n` is known to be a constant, say 4, then we can make a specialized version of `power` where `n` is always 4 by unfolding the recursive call to `power` *without* knowning the value of `m` as follows:

```
   (power m 4)
-> (* m (power m 3))
-> (* m (* m (power m 2)))
-> (* m (* m (* m (power m 1))))
-> (* m (* m (* m (* m (power m 0)))))
-> (* m (* m (* m (* m 1))))
```

Thus, we obtain the following specialization:

```
(define (power4 m)
  (* m (* m (* m (* m 1)))))
```

which is much faster than the original execution `(power m 4)`.

Partial evaluation is effective when a function is used many times, and some of its arguments are always the same. In the above example, it pays off to make `power4` once, if it is used sufficiently many times afterwards.

The partial evaluation technique is widely used, for example, in generation of compilers from interpreters [6, 8], optimization of object-oriented languages [5], and compilation of reflective languages [10]. Because of its ability to mechanically specialize general-purpose programs into specialized, more efficient programs, it is said that partial evaluation achieves both the productivity and efficiency [8]. Partial evaluation has been mainly studied for functional languages, but there are also partial evaluators for C such as Tempo [4] and C-Mix [1]. (See `http://compose.labri.fr/prototypes/tempo/` for Tempo, and `http://www.diku.dk/research-groups/topps/activities/cmix/` for C-Mix.)

We have implemented a Scheme interpreter where a special partial evaluation construct `pe` is available [3]. For example, we can obtain the specialized power program as follows:

```
> (pe (lambda (m) (power m 4)))
(lambda (m) (* m (* m (* m (* m 1)))))
```

Given a function as an argument, `pe` returns a function which is the specialized version of the original argument. In contrast to the conventional partial evaluators, which have been mainly a source to source program transformation, `pe` specializes its argument *at runtime*. We do not have to prepare a program in a separate file for specialization, but we simply insert `pe` whenever

```
(define (main V-point objects lights)
  (map (pe (lambda (pixel)
             (let ((V-vector (- pixel V-point)))
               (ray-trace V-point V-vector objects lights))))
       ; list all the pixels on the screen))
(define (ray-trace V-point V-vector objects lights)
  (if V-vector intersects with objects
      (add ; ambient light
           ; diffuse reflection (if lighted directly)
           ; specular reflection (via recursion))
      ; return back-ground color))
```

Figure 2: Pseudo code for ray tracing

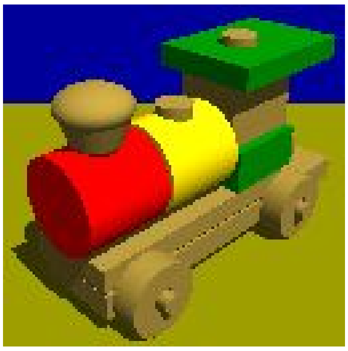

Figure 1: A sample image (128×128 pixels)

we want specialization. This makes it easy to use partial evaluation in a larger program. (In addition, we need to annotate the program to make the recursion variables explicit. We will not discuss it further in this paper, however.)

2 Ray tracing

Ray tracing produces a realistic image of 3D objects by calculating the color of each pixel on the screen. Figure 1 shows a sample output of ray tracing. Given a configuration of 3D objects, we trace light in the reverse direction from the view-point (`V-point`) to each pixel on the screen. When it intersects with an object, we collect various light that reaches that intersecting point, such as ambient light (uniform light whose intensity is the same at all places), diffuse reflection (light coming directly from the light source; effective only when the point is visible from the light source), and specular reflection (reflected light coming from the direction of mirror-reflection; computed using recursion). Figure 2 shows the pseudo code for it.

3 Where to use partial evaluation

The interesting observation on the above pseudo code is that (1) `ray-trace` is called as many times as the number of pixels on the screen, and that (2) for each execution, the object configuration `objects` and the light configuration `lights` are constants. Since the number of times `ray-trace` is called is quite large, we can expect that the execution will become faster if we make a specialized version of `ray-trace` (with respect to the fixed `objects` and `lights`) once, and use this optimized version to compute the color of all the pixels. This is exactly where partial evaluation can contribute.

We do so by inserting `pe` in `main` as is already done in Fig. 2. The `pe` construct makes the specialized version of `ray-trace` where `objects` and `lights` are fixed, but `pixel` is unknown (since it is λ-abstracted). Intuitively, it inlines all the traversals over the objects and lights configuration into the resulting specialized program.

4 Experimental results

We have implemented a ray tracer that conforms to the requirements in the ICFP 2000 programming contest, where the task was to implement a ray tracer. (See `http://www.cs.cornell.edu/icfp/` for the details of this contest.) It supports five object types (spheres, cubes, cylinders, cones, and planes), three operations on objects (union, intersection, and difference), three light types (directional light, point light, and spotlight), various transformations (translation, scaling, and rotation), and procedural surface ren-

scene	size	without PE	with PE	ratio	size	without PE	with PE	ratio
spheres	32×24	23.7	12.1	1.96	320×240	7.0	4.3	1.65
spheres2	32×24	28.1	13.7	2.05	320×240	8.4	4.8	1.74
reflect	32×24	51.8	23.9	2.17	320×240	13.9	7.7	1.80
fib	32×24	48.5	42.0	1.16	320×240	17.1	11.6	1.47
cube	32×20	4.0	2.6	1.53	320×200	1.7	1.36	1.25
dice	64×40	241.1	138.8	1.74	640×400	75.1	47.9	1.57
spot	32×24	16.1	10.1	1.48	320×240	5.0	3.9	1.30
golf	32×24	80.3	38.8	2.07	320×240	21.4	11.9	1.80

Table 1: Experimental results in seconds. The columns ‘size’ show the number of pixels. The left half uses `pe` construct while the right half uses manual annotation to achieve partial evaluation.

dering. We have also implemented bounding volumes to reduce the number of intersection calculation. The experiment was done on Sun Blade 1000 (UltraSPARC-III, 750MHz, 512 MB memory) using Chez Scheme compiler with optimize-level 3.

Table 1 shows the experimental results. The column ‘scene’ shows the name of the scenes that we used. The left half of the table shows how the use of `pe` makes the execution faster. We can observe 16 percent to 2 times speedup by the use of partial evaluation. In all cases, the time for partial evaluation was ignorable.

However, the absolute figures for these experiments are not very satisfactory. Even after partial evaluation, it takes more than ten seconds for a scene of size only 32×24 pixels. One of the reason for it is that the ray tracing program is executed (interpreted) on top of the partial evaluator we made. Although the partial evaluator itself is compiled via Chez Scheme compiler, the ray tracing program is interpreted by this (compiled) partial evaluator. If we could implement the partial evaluator by extending the Chez Scheme compiler directly, we could obtain better figures.

To see it, we have made another experiment. Since it is not very easy to directly extend the Scheme interpreter to cope with `pe`, we implemented the effect of partial evaluation manually by rewriting the ray tracing program by hand instead. It is well-known that partial evaluation can be simulated by separating the program into two stages and inserting *quote* and *unquote*. (Note that we can no longer achieve partial evaluation by simply inserting `pe`. Instead, we have to rewrite major parts of the program. Proper knowledge on (offline) partial evaluation is also required to do it.) Then, the resulting program is compiled using Chez Scheme compiler. The experimental results for this hand-tweaked ray tracer are given in the right half of Table 1.

First, notice that the sizes of scenes are 100 times larger than the previous experiment. This means that this implementation is about 300 times faster than the previous one. (For example, the scene ‘spheres’ finished $23.7 \times 100/7.0 \approx 338$ times faster.) From this, we can see that the overhead of interpretation was about the magnitude of 300 times. Partial evaluation then adds about 25 to 80 percent speedup. Although the speedup is rather modest than the previous experiment, partial evaluation does improve the performance.

Then, the natural question to ask is whether this is sufficient or not. Unfortunately, it is still not very fast. The winning program of the ICFP 2000 programming contest is so fast that the execution time for all the scenes used in this experiment was zero! (See `http://www.cis.upenn.edu/~sumii/icfp/` for the details of the winning program.)

To compare the winning program with ours, we have experimented on a larger (and more complicated) scene, and found that the winning program is about 70 times faster than our ray tracer without partial evaluation. (We could not compare the version that uses partial evaluation because the produced specialized code was so big that the Chez Scheme compiler could not compile it.) Assuming optimistically that we obtain 2 times speedup via partial evaluation, we still lose by about 35 times from the winning program. Because the winning program is written in a typed language (OCaml), it is benefited from static type-checking to remove runtime checks. Estimating optimistically that static type checking makes programs three times faster, we still lose by more than ten times. Currently, we have no clue how to bridge this gap.

5 Related work

Specialization of ray tracing was first done by Mogensen[11]. Andersen[2] specialized an already efficient ray tracer written in C. We did the same experiment using Scheme extended with `pe`. Thanks to the presence of `pe`, we could achieve specialization simply by inserting `pe` at an appropriate place. The realistic application of specialization in graphics is found in [7].

6 Conclusion

This short paper reported our experience on using partial evaluation to improve the performance of ray tracing. By exploiting the static information (the objects and light configuration), we can produce a specialized ray tracer which runs up to two times faster than the unspecialized version. Thanks to the integration of a partial evaluator into an interpreter, specialization was achieved simply by inserting `pe` at an appropriate place.

Because we did not directly change the underlying language implementation to support partial evaluation but we implemented it as a stand-alone program, it suffers from severe interpretive overhead. Although this approach is good for prototypical implementation, it does not suit for practical uses of partial evaluation. Since we obtained significant speedup for the ray tracer with manual partial evaluation, however, there is a good hope that implementing partial evaluation directly in the underlying Scheme implementation will be beneficial.

References

[1] Andersen, L. O. *Program Analysis and Specialization for the C Programming Language*, Ph.D. thesis, DIKU, University of Copenhagen (May 1994).

[2] Andersen, P. H. "Partial Evaluation Applied to Ray Tracing," *Software Engineering in Scientific Computing*, pp. 78–85, DIKU report D-289 (1996).

[3] Asai, K. "Integrating Partial Evaluators into Interpreters," In W. Taha, editor, *Semantics, Applications, and Implementation of Program Generation (LNCS 2196)*, pp. 126–145 (September 2001).

[4] Consel, C., L. Hornof, R. Marlet, G. Muller, S. Thibault, E. -N. Volanschi, J. Lawall, and J. Noyé "Tempo: Specializing Systems Applications and Beyond," *ACM Computing Surveys*, Vol. 30, Issue 3es (September 1998).

[5] Dean, J., C. Chambers, and D. Grove "Identifying Profitable Specialization in Object-Oriented Languages," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '94)*, pp. 85–96 (June 1994).

[6] Futamura, Y. "Partial evaluation of computation process – an approach to a compiler-compiler," *Systems, Computers, Controls*, Vol. 2, No. 5, pp. 45–50, (1971), reprinted as *Higher-Order and Symbolic Computation*, Vol. 12, No. 4, pp. 381–391, Kluwer Academic Publishers (December 1999).

[7] Guenter, B., T. B. Knoblock, E. Ruf "Specializing Shaders," *Proceedings of SIGGRAPH 95 (Computer Graphics Proceedings)* pp. 343–350 (1995).

[8] Jones, N. D., C. K. Gomard, and P. Sestoft *Partial Evaluation and Automatic Program Generation*, New York: Prentice-Hall (1993).

[9] Kelsey, R., W. Clinger, and J. Rees (editors) "Revised^5 Report on the Algorithmic Language Scheme," *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, pp. 7–105, Kluwer Academic Publishers (August 1998).

[10] Masuhara, H., S. Matsuoka, K. Asai, and A. Yonezawa "Compiling Away the Meta-Level in Object-Oriented Concurrent Reflective Languages Using Partial Evaluation," *Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pp. 300–315, (October 1995).

[11] Mogensen, T. Æ. "The Application of Partial Evaluation to Ray-Tracing," Master's thesis, DIKU, University of Copenhagen (1986).