SPECIALIZING VISUALIZATION ALGORITHMS

Stephan Diehl
FR Informatik
Universität des Saarlandes
diehl@cs.uni-sb.de

Abstract In this paper we look at the potential of program specialization techniques in the context of visualization. In particular we look at partial evaluation and pass separation and how these have been used to automatically produce more efficient implementations and how they can be used to design new algorithms. We conclude by discussing what we think are the applications were program specialization is most promising in visualization.

Keywords: visualization, program transformation, partial evaluation, pass separation, marching cubes

Introduction

The design and optimization of algorithms often starts with simple algorithms which are then transformed. These transformations are either based on insights from the underlying domain or on the semantics of the notation used to encode the algorithm. Such transformations have to preserve the input-output behavior of the algorithm, while improving its efficiency. In particular in scientific visualization, algorithms have to process huge amounts of data and efficiency is an important issue. In this paper we look at the specialization of visualization algorithms using two program transformation techniques which have been extensively applied in the context of compiler generation.

Program Specialization

The term *staging transformation* has been introduced by Jørring and Scherlis, 1986 for a class of program transformations including partial evaluation and pass separation.

Partial Evaluation: A partial evaluator takes a program and values for some of its inputs and produces a new, specialized program parameterized by the remaining inputs

Pass Separation: Pass separation splits a program into two parts. The second part reads the output of the first part and produces the final result.

Partial evaluation and pass separation differ in the kind of information that they have about the program. Partial evaluation knows the exact values of some part of the input, whereas pass separation only knows which part of the input will be available first.

Let P be a program, x and y the static and dynamic inputs to this program and $\overline{x}$ the statically known value of x, then *partial evaluation* of P with respect to $\overline{x}$ yields a residual program $P_{\overline{x}}$, such that $P_{\overline{x}}(y) = P(\overline{x}, y)$. In contrast, *pass separation* transforms the program P into two programs P_1 and P_2 such that $P_2(P_1(x), y) = P(x, y)$. What is important about this equation is that here P_1 produces some intermediate data, which are input to P_2. As is well known, partial evaluation can be used to generate compilers in various ways according to the Futamura Projections (Jones et al., 1993; Futamura, 1971). When it comes to the generation of compiler/executor pairs, pass separation provides an immediate solution. We pass separate the interpreter *interp* into an executor *exec* and a compiler *comp*, such that: $interp(prog, data) = exec(comp(prog), data)$. Despite its potential for compiler generation there is only little work on pass separation. Actually we are only aware of the somewhat hand-waving article (Jørring and Scherlis, 1986) and the provably correct pass separation transformations of term rewriting systems (Hannan, 1994), evolving algebras (Diehl, 1995), as well as our fully automatic semantics-directed compiler generator which is based on pass separation (Diehl, 1996).

Example: Partial Evaluation of Ray Tracing

Already in 1986 Mogensen applied partial evaluation to a ray tracer implemented in a functional language (Mogensen, 1986). Ten years later Andersen did a similar experiment with a ray tracer implemented in C (Andersen, 1996). The ray tracer spends much of its time testing intersection of each object in the scene with the ray. Andersen specialized the ray tracer with respect to a given scene (set of objects).

```
intersectionTest(sphere,ray)
{ ...
   x=sphere.c.x - ray.p.x;
  ...
}
```

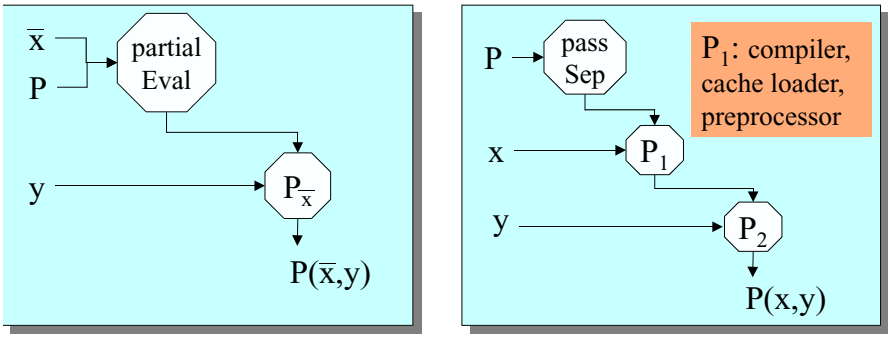

Figure 1. Partial evaluation vs. pass separation

Thus for the 32nd object in the scene he got for example the following specialized version of the intersection test.

```
intersectionTest_32(ray)
{ ...
   x=1.0 - ray.p.x;
  ...
}
```

His partial evaluator performs inter-procedural constant propagation and unfolds the loop which iterates over all objects, thus yielding specialized functions for each object.

Andersen reports speedups of 1.3 to 3.0 and increase in code size of 15 to 90 times, but the truth is that because he unfolds the loop, the increase in code size is in the order of the number of objects.

Example: Pass Separation of Shaders

Although they do not use the term, Guenther et. al. pass separated shaders into cache loaders and cache readers (Guenther et al., 1995). In the excerpts of the program code of a Phong shader below we assume that all parameter values be known before the value of `specular` becomes known. All values which depend on it are underlined and have to be computed in the final pass.

```
Phong(i,j,surface_normal, ..., specular, ...)
{ ...
  n_dot_h=VecDot(surface_normal, h_vector)
  if (n_dot_h>0)
     specular_component=pow(n_dot_h,specular);
```

```
    else
      specular_component=0;
  ...
  rgb=VecScalarMult(rgb, (kd*diffuse_component+ambient)
                       + ks*specular_component);
  SetPixel(I,j,rgb,dib);
}
```

The specializer separates the algorithm into a cache loader, cache reader and a fixed driver. The cache loader forms the first pass, while the cache reader forms the second pass. Now we look at the generated source code of the loader and reader.

The loader shown below computes all values, which have not been underlined and stores them in a data structure referenced by `pcache`:

```
n_dot_h=VecDot(surface_normal, h_vector)
if (pcache->c0 = n_dot_h>0)
   pcache->c1=n_dot_h
...
pcache->c2= kd*diffuse_component+ambient;
```

The reader is obtained from the original program by replacing all non underlined expressions by references to the precomputed values in `pcache`:

```
if (pcache->c0>0)
   specular_component=pow(pcache->c1,specular);
 else
   specular_component=0;
...
rgb=VecScalarMult(rgb, pcache->c2 + ks*specular_component);
SetPixel(I,j,rgb,dib);
```

After generating the loader and the reader. The user can now render the scene with different values of `specular`. Note that the loader is only executed once, while the reader is invoked each time the user enters a new value for `specular`:

For all pixels `(i,j)`: `Loader(i,j, ...)`
User/GUI supplies values for `specular`
For all pixels `(i,j)`: `Reader(i,j,specular)`

The authors report speedups of up to 95 times and an increase in code size of factor 2. Using caching instead of unfolding prevents them from a linear increase in code size as in the previous approach.

As all currently known pass separation transformations, their approach is restricted to a very small class of programs. It seems that their method can only be applied to functions (like `Phong` here) which do not contain recursion or iteration.

Example: How to invent marching cubes

The Warren Abstract Machine WAM is the de-facto standard for implementing logic programming languages (Warren, 1977). Most subsequent work extends or optimizes Warren's ingenious virtual machine. In the article "How to invent a Prolog Machine" (Kursawe, 1986) Kursawe shows how some of the WAM instructions for unification can be systematically derived using partial evaluation. Later Nilsson (Nilsson, 1993) derived most of the remaining instructions using partial evaluation and pass separation. In a sense these papers deprived the WAM of some of its mystique. Based on this work the current author later demonstrated how partial evaluation and pass separation can be used to design new abstract machines (Diehl, 1997).

The marching cubes algorithm (Lorensen and Cline, 1987) is probably among the most influential developments in scientific visualization so far. It reconstructs isosurfaces, i.e. locations of a constant scalar value within a volume. In the next sections we try to reconstruct it using a similar approach as Kursawe and Nilsson.

Surface Construction

To explore three-dimensional data sets, e.g. of medical data, isosurfaces are widely used. The problem of surface construction can be briefly characterized as follows. The physical reality to be visualized can be modelled as a continuous space of real data $d(p, q, r)$ where p, q and r are real numbers. But we only measure a finite number of sample data $D(i, j, k)$ where i, j and k are integers. D is called a grid. An isosurface is the set of all data points (d, p, r) with $d(p, q, r) = c$ for a constant value c. The problem of surface construction is to find a set of triangles which approximates the isosurface based on the sample data in the grid.

First shot

We start by a simple algorithm which processes the cubical cells of the grid one after another. The algorithm uses a function *triangulate*() which given a set of unstructured points returns a set of triangles, where the vertices of the triangles are the given points.

> For each edge of the 12 edges of a cell perform the intersection test:
> e.g. edge (i, j, k) to $(i, j + 1, k)$:
> if $D(i, j, k) \geq c$ and $D(i, j + 1, k) < c$ then
> interpolate intersection point as $P_1 = (i, j + \frac{D(i,j,k)-c}{c-D(i,j+1,k)}, k)$
> else $P_1 = null$
> Let $P_1, \ldots, P_{12}$ be the intersection points, then $T = triangulate(\{P_i | P_i \neq null\})$ are the triangles for this cell.

The problem with this algorithm is efficiency, because for each cell a triangulation has to be computed.

Second shot

In the next step we use the geometric insight, that instead of using the interpolated intersection points we can use arbitrary intersection points, here the mid points of each edge, and do the interpolation later.

> For each edge of the 12 edges of a cell perform the intersection test:
> e.g. edge (i, j, k) to $(i, j + 1, k)$:
> if $D(i, j, k) \geq c$ and $D(i, j + 1, k) < c$ then
> define intersection point in unit cube as $P_1 = (0, \frac{1}{2}, 0)$
> else $P_1 = null$
> Let $P_1, \ldots, P_{12}$ be the intersection points, then $T = triangulate(\{P_i | P_i \neq null\})$ is a set of triangles in the unit cube.
> Replace P_i in the triangles by the interpolated intersection point P_i',
> e.g. $P_1' = (i, j + \frac{D(i,j,k)-c}{c-D(i,j+1,k)}, k)$

Final shot

Triangulating before interpolating does not yet improve the performance of the algorithm, but it has changed the information on which the triangulation depends. We can now pass separate the above algorithm, all mid points of the edges of the unit cube are known, before we have to run the algorithm for a given grid. Here the cache loader precomputes all triangulations for all possible combinations of mid points of edges of the unit cube.

> Cache Loader:
> Let P_i be the points in the middle of each edge of the unit cube. For all subsets $S \subseteq \{P_1, \ldots, P_{12}\}$ compute $cache[S] = triangulate(S)$
>
> Cache Reader:
> For each edge of the 12 edges of a cell perform the intersection test:
> e.g. edge (i, j, k) to $(i, j + 1, k)$:
> if $D(i, j, k) \geq c$ and $D(i, j + 1, k) < c$ then
> define intersection point in unit cube as $P_1 = (0, \frac{1}{2}, 0)$
> else $P_1 = null$
> Let $P_1, \ldots, P_{12}$ be the intersection points then $T = cache[P_i | P_i \neq null]$ is a set of triangles in the unit cube.
> Replace P_i in the triangles by the interpolated intersection point P_i',
> e.g. $P_1' = (i, j + \frac{D(i,j,k)-c}{c-D(i,j+1,k)}, k)$

As in the original marching cubes algorithm, in our presentation we ignored the problem of ambiguities (Nielson and Hamann, 1991; Matveyev, 1994; Montani et al., 1994), which are caused by the cache. In the marching cubes algorithm the cache is called lookup table and it is further reduced from 256 to 14 entries by exploiting symmetries.

Future Work

In our future work we intend to look at what other optimization and specialization tricks have been used in visualization algorithms, e.g. how it is possible to lift the visibility test from the renderer to the isosurface computation (Livnat and Hansen, 1998).

Specialize once, run multiple

Currently there is no pass separation tool which (semi-) automatically transforms programs in a conventional programming language Such a tool would be very helpful to assist the manual specialization and design of other visualization algorithms. Automatic partial evaluators of very different quality exist for many programming languages.

Whether we use manual or fully automatic specialization, the costs for specializing a program can only be amortized if we intensively use the specialized program. We briefly discuss two visualization scenarios where the costs of automatic specialization will certainly be amortized.

Computational Steering. In computational steering visualization is used to control scientific applications including the generation and filtering of data (Johnson et al., 1999). If the user is restricted to change only some parameters of the application, the visualization algorithms can be specialized with respect to the fixed parameters.

Comparative Visualization. Different data sets can be compared by visualizing them using the same parameters for a visualization algorithm. As a result the algorithm can be specialized with respect to these parameters.

Conclusions

The goal of this paper was to draw the attention of the visualization community to the potential of program specialization techniques for visualization. It remains a vastly unexplored field and this paper may serve as a first map to those who want to break new ground.

Acknowledgements. The author wants to thank Hans Hagen, Thomas Ertl and Reinhard Wilhelm for instructive discussions and encouragement.

References

Andersen, P. H. (1996). Partial evaluation applied to ray tracing. In Mackens, W. and Rump, S., editors, *Proc. of Software Engineering in Scientific Computing*. Vieweg.

Diehl, S. (1995). Transformations of Evolving Algebras. Technical Report FB14 No. A-02/95, University Saarbrücken.

Diehl, S. (1996). *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, University Saarbrücken, Germany. http://www.cs.uni-sb.de/~diehl/phd.html.

Diehl, S. (1997). An Experiment in Abstract Machine Design. *Software – Practice and Experience*, 27(1).

Futamura, Y. (1971). Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5).

Guenther, B., Knoblock, T. B., and Ruf, E. (1995). Specializing shaders. In *Proc. of SIGGRAPH95*. ACM SIGGRAPH, Computer Graphics.

Hannan, J. (1994). Operational Semantics-Directed Compilers and Machine Architectures. *ACM Transactions on Programming Languages and Systems*, 16(4):1215–1247.

Johnson, C., Parker, S., Hansen, C., Kindlmann, G., and Livnat, Y. (1999). Interactive simulation and visualization. *Computer*, 12.

Jones, N., Gomard, C., and Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

Jørring, U. and Scherlis, W. (1986). Compilers and staging transformations. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 86–96.

Kursawe, P. (1986). How to invent a Prolog machine. In *Proc. Third International Conference on Logic Programming*, volume LNCS 225, pages 134–148. Springer Verlag.

Livnat, Y. and Hansen, C. (1998). View dependent isosurface extraction. In *Proc. of Visualization'98*.

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. *ACM Computer Graphics*, 21(4).

Matveyev, S. V. (1994). Aproximation of isosurface in the marching cubes: Ambiguity problem. In *Proc. of Visualization'94*.

Mogensen, T. (1986). The Application of Partial Evaluation to Ray-Tracing. Master's thesis, DIKU, University of Copenhagen, Denmark.

Montani, C., Scateni, R., and Scopigno, R. (1994). Discretized marching cubes. In *Proc. of Visualization'94*.

Nielson, G. and Hamann, B. (1991). The asymptotic decider: resolving the ambiguity in marching cubes. In Nielson, G. and Rosenblum, L., editors, *Proc. of IEEE Visualization '91*. IEEE Computer Society Press.

Nilsson, U. (1993). Towards a Methodology for the Design of Abstract Machines for Logic Programming. *Journal of Logic Programming*, 16:163–188.

Warren, D. H. (1977). Implementing prolog – compiling predicate logic programs. D.A.I Research Report, No. 40.