Visualizing Partial Evaluation

Oscar Waddell and R. Kent Dybvig
Indiana University

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*; D.2.2 [**Software Engineering**]: Tools and Techniques—*User interfaces*

General Terms: Languages, Processors

Additional Key Words and Phrases: Visualization, Resource-bounded partial evaluation

1. MOTIVATION

Visualization can help both implementors and users of partial evaluators to understand (1) where reductions are applied in a given source program, (2) what residual code is produced by these reductions, (3) how these transformation decisions are made, and (4) how these transformations affect the code generated by subsequent passes of a compiler. We have identified the following as important characteristics of visualization tools designed to help answer these questions.

—The source and residual programs are displayed side-by-side and correlated so that a user can see the residual code produced for a given source expression or the source code that produced a given residual-program expression.

—If the partial evaluator is part of a larger compilation system, the output of each subsequent compiler pass can be displayed as well, and correlated with the source and residual code so that the effects of partial evaluation on subsequent compiler passes can be seen.

—Residual code and code from subsequent compiler passes, if any, can be displayed in the concrete syntax of the full source language, regardless of the intermediate representations used by the partial evaluator or compiler. This may require decompilation of intermediate code or unexpansion of macro-expanded code.

—The display of information can be constrained to particular frames of reference. In particular, if the binding-time analysis is polyvariant, it is possible to view call sites for which a given expression is assigned different binding times or to display binding times relative to particular call contexts.

This material is based on work supported in part by the National Science Foundation under grants CDA-9312614 and CCR-9711269.
Address: Indiana University, Bloomington, IN 47405 {owaddell,dyb}@cs.indiana.edu

—The effects of different binding-time annotations and partial evaluation strategies can be compared by highlighting portions of the original source code for which different residual code is produced.

—It is possible to inspect the specific information on which a particular transformation decision is based. For example, if the partial evaluator is resource-bounded [Danvy et al. 1996; Debray 1997], resource constraints that prevent a particular expression from unfolding can be displayed.

—Information about the compile- and run-time effects of partial evaluation is available for display, including code size, success metrics for subsequent optimization passes, compile time, execution time, and profile data. The user can select a level of detail interactively for each portion of the source or residual code.

We have developed a prototype visualization tool that has some of these characteristics. It is used to visualize the results obtained by a resource-bounded online partial evaluator we have implemented and installed as an optimization pass in the *Chez Scheme* compiler [Waddell and Dybvig 1997].

2. IMPLEMENTATION

Abstract-syntax records manipulated by the partial evaluator contain source file and position information collected by the lexical analyzer and maintained during parsing and expansion of syntactic extensions [Dybvig et al. 1993]. Transformations within the partial evaluator and all subsequent compiler passes preserve this source file information.

Because source information is contained within abstract-syntax records, that information is automatically preserved when code is replicated during specialization. Other transformations may collapse an abstract-syntax tree into a single node in the residual program. In such cases we select an appropriate source annotation for the residual code from among the source annotations in the input expression. For example, the residual form produced via copy propagation is annotated with the source information from the variable reference it replaces. Similarly, the residual form produced by constant folding is annotated with the source information from the primitive application it replaces.

The viewer displays the source program and a pretty-printed form of the residual program in separate windows. To facilitate pretty-printing, the residual program is first converted to list structure. To enhance readability of the residual program, the conversion process reverses the expansion of common syntactic extensions, e.g., `let`, `and`, and `or`. Directives embedded within the resulting list structure cause the pretty-printer to correlate the current output position with the source information contained in the abstract-syntax record.

When pretty-printing is complete, we have constructed tables that correlate the position of each source program expression with the positions of zero or more residual program expressions and the position of each residual program expression with the position of at most one source program expression. These tables are consulted when the user selects an expression in the source or residual program window and the corresponding expressions, if any, are marked in both windows as shown in Figure 1.

The partial evaluator can be instantiated with different functions for annotating

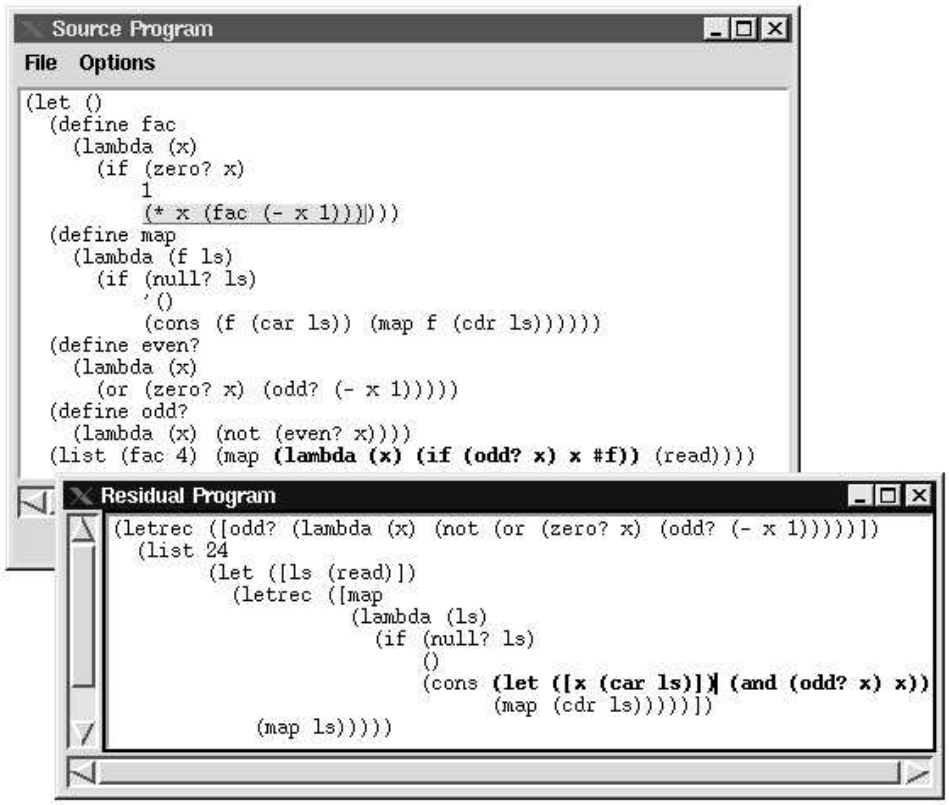

Fig. 1. The user has instructed the viewer to display calls where constant folding took place with a light gray background. The user has also selected a `let` expression in the residual program and the viewer has displayed the corresponding source and residual program expressions in boldface.

residual code with source information. For production use, residual code is annotated as described earlier. An alternative instantiation records additional details such as the transformation that produced each residual form. The viewer supports both forms of source annotation and uses the additional information, if available, to enrich the display. For example, the viewer may use the additional information to mark points in the source program where procedures were inlined. Another instantiation of the partial evaluator is configured to trigger a data breakpoint when processing selected input forms. These breakpoints can be set or cleared by selecting expressions in the source file viewer. An implementor can use this feature to examine the processing of a particular source expression in detail.

3. CONCLUSION

The role of visualization in partial evaluators has primarily been to display source code annotated with binding times. Example systems include Schism [Consel 1996], Similix [Bondorf 1993], and Tempo [Consel et al. 1998]. Visualization plays a more significant role in a partial evaluator that was developed to assist in the task of understanding legacy programs [Blazy and Facon 1994]. That system provides a

correlated display of source and residual programs, much like our own, though for a language that lacks powerful syntactic abstraction facilities.

As partial evaluation becomes a mainstream compiler technology, it is applied to ever larger programs and applied by less sophisticated users. Experience with our system and others suggests that visualization can help implementors and users alike to understand both the process and the results of partial evaluation.

ACKNOWLEDGMENTS

Comments from Olivier Danvy and anonymous referees substantially improved the focus of this paper.

REFERENCES

Blazy, S. and Facon, P. 1994. Partial evaluation for the understanding of FORTRAN programs. *International Journal of Software Engineering and Knowledge Engineering 4*, 4, 535–559.

Bondorf, A. 1993. *Similix Manual, System Version 5.0*. University of Copenhagen, Denmark: DIKU.

Consel, C. 1996. Report on Schism '96. Research report, Irisa, Rennes, France.

Consel, C., Hornof, L., Lawall, J., Marlet, R., Muller, G., Noël, F., Noyé, J., Thibault, S., and Volanschi, N. 1998. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*. To appear.

Danvy, O., Heintze, N., and Malmkjær, K. 1996. Resource-bounded partial evaluation. *ACM Computing Surveys 28*, 2 (June), 329–332.

Debray, S. 1997. Resource-bounded partial evaluation. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (June 1997), pp. 179–192.

Dybvig, R. K., Hieb, R., and Bruggeman, C. 1993. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5*, 4, 295–326.

Waddell, O. and Dybvig, R. K. 1997. Fast and effective procedure inlining. In P. V. Hentenryck Ed., *Fourth International Symposium on Static Analysis*, Volume 1302 of *Lecture Notes in Computer Science* (1997), pp. 35–52. Springer-Verlag.