Code Generation for Free with Partial Evaluation
CSE 501 Course Project

NATE YAZDANI, University of Washington

1 INTRODUCTION

1.1 Motivation

Synthesis systems typically search and reason about programs in terms of syntax [1]. Many therefore employ domain-specific languages, as the domain specificity reduces the search space and simplifies logical reasoning (*e.g.*, by eliding low-level details or by admitting useful proof rules). These languages are commonly purpose-built for synthesis (*e.g.*, as a solver-aided domain-specific language [6]) and require compilation for efficient execution. The typical solution is to construct a compiler to some general-purpose programming language; this is easier than targeting assembly and permits reuse of existing optimizing compilers. Such a compiler is a *code generator*, and this compilation process is *code generation*.

1.2 Problem

Code generation is a necessary process for a useful synthesis system, but implementation of a code generator is a significant engineering challenge, distracting from innovation in the actual synthesis procedure. In the domain-specific language, a high-level of abstraction facilitates effective search and reasoning, but compiling these high-level abstractions to low-level code introduces complexity and subtlety. Since an effective synthesis system generally requires a language carefully designed for the problem domain, there is little opportunity to reuse code generators.

This report presents an automatic method of code generation for deeply embedded domain-specific languages [1]. To be useful, an automatic method for code generation must exhibit the following qualities:

(1) *Simplicity.* Code generation must require nothing more than the program and interpreter.
(2) *Efficiency.* The generated code must be efficient.
(3) *Integration.* The generated code must support integration with existing libraries.
(4) *Performance.* Code generation itself must be fast.

This report presents a system for automatic code generation that achieves these goals, by translation of the residual program derived from partial evaluation of the interpreter run on the program and unknown input.

1.3 Example

Code generation for a language of integer arithmetic expressions, $L_{\mathbb{Z}}$, serves as the report's running example. Figure 1 shows the syntax of this language, where V representing an infinite supply of variable names (*i.e.*, the set of strings). The usual precedence rules for arithmetic operations apply. Figure 2 shows an example program in $L_{\mathbb{Z}}$, where the variable k is a run-time input to the program. Most expressions have the expected semantics, but two merit explanation. First, the division operation (/) expresses Euclidean division, computing the integer quotient. Second, the "Y"

[1]One usually only considers code generation only for deeply embedded languages, where the program syntax is reified as data.

$$E ::= E + E \mid E - E \mid E * E \mid E \,/\, E \mid \sum_{V:=E}^{E} E \mid \prod_{V:=E}^{E} E \mid \underset{V:=E}{Y} E \mid V \mid \mathbb{Z}$$

Fig. 1. Syntax for a language of integer arithmetic expressions, $L_{\mathbb{Z}}$.

$$\left(\prod_{\mathtt{x}:=1}^{4} \mathtt{k} * \mathtt{x}\right) + \left(\underset{\mathtt{x}:=16}{Y} \mathtt{x} \,/\, 2 + \mathtt{k}\right)$$

Fig. 2. An example program in $L_{\mathbb{Z}}$.

operation expresses fixed-point iteration [2]. Alone, the traditional arithmetic operations on integers (+, −, ⋆, and /) are an easy instance of code generation, as most programming languages include those operators as primitives, but the inclusion of the three iterative operations makes the example more interesting, by adding control flow. In the case of fixed-point iteration, divergence is also possible. A good code generator must successfully manage the interaction of this control flow with unknown inputs (*e.g.*, compile the bodies of loops with unknown bounds).

This report also demonstrates effectiveness of the system on a more complicated example. Section 3 presents the code-generation problem for a language of tree traversal schedules adapted from a real-world synthesis system [5].

1.4 Overview

This report is structured as follows. Section 2 describes the approach and implementation of the project's solution. Section 3 presents an evaluation of the project's system against the desiderata in Section 1.2. Section 4 surveys closely related work. Section 5 discusses lessons learned and future work and ends the report with brief closing remarks.

2 SOLUTION

This section discusses the approach and implementation used in this project's solution for code generation. The source code is available at https://bitbucket.org/nyazdani/cse501-project.

2.1 Approach

The project's solution is based on the well-known technique of *partial evaluation* [4]. The idea of partial evaluation is to split the input of a program into *static* and *dynamic* components and then *specialize* the program to the given static input. The static component is the part of the input known at compilation (*i.e.*, partial-evaluation) time, and the dynamic component is only known at run time. In the case of this project, the program is the language interpreter, the static input is the program for code generation, and the dynamic input is that program's input.

Like many problems in computer science, the primary challenge in partial evaluation is how to handle recursion and termination. If the partial evaluator unrolls recursive calls too eagerly, it may diverge or produce an unacceptably large output program; conversely, if the partial evaluator is not eager enough, significant opportunities for performance improvement (*e.g.*, loop bodies) may go untapped, leaving an unacceptably inefficient output program.

[2]Given a function $f : A \to A$ and an initial point $x_0 : A$, fixed-point iteration computes the fixed point $x : A$ such that $f(x) = x$ and $x = f^n(x_0)$ for some n. If such a point does not exist, fixed-point iteration diverges.

$$\begin{array}{rcl}
e & ::= & v \mid x \mid \texttt{(}e\ e\texttt{)} \mid \texttt{(let (}e\ e\texttt{)}\ e\texttt{)} \mid \texttt{(begin}\ e\ e^{*}\texttt{)} \mid \texttt{(if}\ e\ e\ e\texttt{)} \\
 & \mid & \texttt{(box}\ e\texttt{)} \mid \texttt{(set-box!}\ e\ e\texttt{)} \mid \texttt{(unbox}\ e\texttt{)} \\
 & \mid & \texttt{(cons}\ e\ e\texttt{)} \mid \texttt{(car}\ e\texttt{)} \mid \texttt{(cdr}\ e\texttt{)} \\
 & \mid & \texttt{(+}\ e\ e\texttt{)} \mid \texttt{(-}\ e\ e\texttt{)} \mid \texttt{(*}\ e\ e\texttt{)} \mid \texttt{(/}\ e\ e\texttt{)} \\
 & \mid & \texttt{(<}\ e\ e\texttt{)} \mid \texttt{(<=}\ e\ e\texttt{)} \mid \texttt{(=}\ e\ e\texttt{)} \mid \texttt{(>}\ e\ e\texttt{)} \mid \texttt{(>=}\ e\ e\texttt{)} \\
 & \mid & \texttt{(lambda?}\ e\texttt{)} \mid \texttt{(void?}\ e\texttt{)} \mid \texttt{(null?}\ e\texttt{)} \mid \texttt{(cons?}\ e\texttt{)} \\
 & \mid & \texttt{(boolean?}\ e\texttt{)} \mid \texttt{(integer?}\ e\texttt{)} \mid \texttt{(string?}\ e\texttt{)} \mid \texttt{(equal?}\ e\ e\texttt{)} \\
 & & \\
v & ::= & \texttt{(lambda}\ f\ \texttt{(}x\texttt{)}\ e\texttt{)} \mid \texttt{void} \mid \texttt{null} \mid \texttt{(cons}\ e\ e\texttt{)} \mid \texttt{true} \mid \texttt{false} \mid \mathbb{Z} \mid s
\end{array}$$

Fig. 3. Syntax for the host language, L_H.

This project's solution uses a powerful but conservative strategy based to decide whether to unroll a recursive function application: The partial evaluator unrolls a recursive function application if the argument is a substructure of the previous recursive call or if the application is *non-speculative*. An non-speculative application is on the direct control-flow path of the program, independent of any run-time conditional branching. The first condition permits aggressive expansion of *structurally recursive* functions, which are provably terminating, and the second condition simply accepts any divergence at partial-evaluation time that would definitely occur at run time. For interpreters of high-level domain-specific languages, these conditions are adequate in practice, as such languages frequently omit any unrestricted or particularly complicated looping constructs. The running example exhibits this quality with the simple looping structure of iterated summation, iterated product, and fixed-point iteration, even though fixed-point iteration may diverge.

Another significant challenge for partial evaluation is dealing with side effects of the program. This project's solution takes a conservative approach, by treating all side-effecting expressions as irreducible and immovable, so as to avoid incidentally changing the program's semantics. Particularly when the host language is functional, avoiding excessive usage of side effects in the domain-specific language's interpreter is not a burdensome constraint.

Partial evaluation transforms a given program into a *residual program*, specialized the static input. In this context, that residual program is the interpreter of the domain-specific language specialized to the program for code generation. By itself, this artifact is useful only if one wishes only to run the generated program in the host language of the domain-specific language. However, one may want or even need to run the generated program in another language in order to leverage existing software libraries, optimizing compilers, or other language infrastructure. A final pass of syntax-directed translation converts the residual program into the user's desired target language.

2.2 Implementation

The project's implementation consists of a partial evaluator and syntax translator for a core subset of the Racket programming language, to serve as the host language for interpreters of domain-specific languages. Figure 3 shows the syntax of this language, L_H. Racket is also the implementation language for the code-generation system.

The partial evaluator follows the strategy described in the preceding subsection to handle recursion and standard practice otherwise. That is, the partial evaluator reduces any primitive form whose operands are pure values. To avoid complications introduced by variable shadowing, a

$$\begin{aligned}
[\![\texttt{Void}]\!] &= \texttt{Void} & [\![\texttt{null}]\!] &= \texttt{Null} \\
[\![(e_1\ e_2)]\!] &= \texttt{app}([\![e_1]\!],\ [\![e_2]\!]) & [\![\texttt{(begin } e_1\ e_2\texttt{)}]\!] &= [\![e_1]\!];\ [\![e_2]\!] \\
[\![\texttt{(cons } e_1\ e_2\texttt{)}]\!] &= \texttt{Cons}([\![e_1]\!],\ [\![e_2]\!]) & [\![\texttt{(car } e\texttt{)}]\!] &= \texttt{car}([\![e]\!]) \\
[\![\texttt{(cdr } e\texttt{)}]\!] &= \texttt{cdr}([\![e]\!]) & [\![\texttt{(equal? } e_1\ e_2\texttt{)}]\!] &= ([\![e_1]\!])\ \texttt{=}\ ([\![e_2]\!]) \\
[\![\texttt{(< } e_1\ e_2\texttt{)}]\!] &= \texttt{int\_of\_dyn}([\![e_1]\!])\ \texttt{<}\ \texttt{int\_of\_dyn}([\![e_2]\!]) & [\![\texttt{(lambda? } e\texttt{)}]\!] &= \texttt{is\_lambda}([\![e]\!])
\end{aligned}$$

Fig. 4. A subset of the syntax-translation rules from L_H to OCaml.

```
(+
  (* (* k 1) (* (* k 2) (* (* k 3) (* (* k 4) 1))))
  (let (fix_0 (lambda fix_0 (x_0)
                (if (= (+ (/ x_0 2) k) x_0) x_0 (fix_0 (+ (/ x_0 2) k)))))
    (if (= (+ 8 k) 16) 16 (fix_0 (+ 8 k)))))
```

Fig. 5. Residual program for the example $L_{\mathbb{Z}}$ program (Figure 2).

variable-renaming pass precedes partial evaluation, so that all variables in a context are uniquely named. Figure 5 shows the residual program corresponding to Figure 2.

Following partial evaluation, a syntax-directed translation generates the final output program. This project's implementation uses OCaml as the target language. A minimal runtime library supplies implementations of primitive operations on a dynamic type , shown below. Figure 4 shows a representative subset of the syntax-translation rules from L_H to OCaml.

```
type 'a dyn = Lambda of ('a dyn -> 'a dyn) | Extern of 'a
            | Cons of ('a dyn, 'a dyn) | Null | Void
            | Integer of int | Boolean of bool | String of string
```

3 EVALUATION

This section evaluates the report's proposed system against the desiderata for an automatic method of code generation established in Section 1:

(1) *Simplicity.* Code generation must require nothing more than the program and interpreter.
(2) *Efficiency.* The generated code must be efficient.
(3) *Integration.* The generated code must support integration with existing libraries.
(4) *Performance.* Code generation itself must be fast.

Evaluation of this project's solution applies automatic code generation to two example domain-specific languages implemented in L_H. The first is the running example of integer arithmetic programs in $L_{\mathbb{Z}}$, first introduced in Section 1.3. The second is a language of tree traversal schedules adapted from [5]. Figure 6 shows the syntax of this language, L_S. ?? shows an example tree traversal schedule in this language.

3.1 Criteria

Applied to the two examples, the following benchmarks will measure success or failure at achieving the above qualities:

$$
\begin{array}{lcl}
\mathit{Sched} & ::= & (\texttt{seq}\ \mathit{Sched}\ \mathit{Sched}) \mid (\texttt{par}\ \mathit{Sched}\ \mathit{Sched}) \mid \mathit{Trav} \\
\mathit{Trav} & ::= & (\texttt{pre}\ \mathit{Stmt}^{*}) \mid (\texttt{post}\ \mathit{Stmt}^{*}) \\
\mathit{Stmt} & ::= & (\texttt{set}\ \mathit{Child}\ \mathit{Label}\ \mathit{Expr}) \\
\mathit{Expr} & ::= & (\texttt{+}\ \mathit{Expr}\ \mathit{Expr}) \mid (\texttt{-}\ \mathit{Expr}\ \mathit{Expr}) \mid (\texttt{*}\ \mathit{Expr}\ \mathit{Expr}) \mid (\texttt{/}\ \mathit{Expr}\ \mathit{Expr}) \\
 & \mid & (\texttt{var}\ \mathit{Child}\ \mathit{Label}) \mid \mathbb{Z} \\
\mathit{Child} & ::= & \texttt{self} \mid \texttt{left} \mid \texttt{right} \\
\mathit{Label} & ::= & \texttt{x} \mid \texttt{y} \mid \texttt{z}
\end{array}
$$

Fig. 6. Syntax for a simplified language of tree traversal schedules, L_S.

```
(seq (pre  (set left x (+ 16 (var self x)))
           (set right x (+ 16 (var self x))))
     (post (set self y (+ (var left y) (var right y)))))
```

Fig. 7. An example tree traversal schedule in L_S.

(1) *Simplicity.* Whether code generation requires nothing more than the program and interpreter.
(2) *Efficiency.* Whether generated code embeds any syntax of the source program or any fragments of the interpreter.
(3) *Integration.* Whether generated code successfully integrates with a library independent of the code-generation system.
(4) *Performance.* How fast the entire code-generation pipeline is.

3.2 Results

Table 1 summarizes the results of the evaluation. The following subsection offers an analysis of them.

Benchmark	**Schedules**	**Arithmetic**
(1) Simplicity	✓	✓*
(2) Efficiency	✓	✓*
(3) Integration	✓	✓
(4) Performance	7ms	11ms

Table 1. Results of benchmarks.
* Caveat discussed below.

3.3 Analysis

The first benchmark passed because the code-generation system operates on the application of the interpreter to the source program. While this precise property is independent of the domain-specific language, ease of use does vary between languages. The arithmetic language required that program inputs be bound in the language's environment, rather than being left free in the overall expression in the host language; otherwise, code generation will preserve the entire interpreter in the generated program. Such a metavariable could hold any arbitrary syntax in the arithmetic language at run time, whereas the language assumes that variables in its own environment are integers. Conversely, the schedule language required no such care, because the input, a tree, is an external, mutable data structure, rather than a value potentially computable within the language.

The second benchmark passes for both languages, because the example program for each language compiles to a target program free of the original program syntax and any literal fragments of the interpreter. As mentioned in the previous paragraph, some extra care may be needed depending on how the language treats input variables.

The third benchmark passes due to the code-generation system's generic support for library integration: Free variables are treated as unknown dynamic input and preserved in the generated output code. When applied as functions, they are conservatively considered potentially side-effecting, forbidding the system from duplicating or removing their occurrences. The user writes stub methods to interface and marshal data between a library of interest and the dynamic-typing library used by the generated code. This process is entirely syntactic and amenable to automation. To test this benchmark, the author integrated the code generated from the example schedule in Figure 7 with tree subroutines implemented in pure OCaml; similarly, the author integrated the code generated from the example arithmetic program in Figure 2 with an integer square-root routine implemented in OCaml.

The results of the fourth benchmark were well within the range for reasonable code-generation times, in the opinion of the author. The measurements were taken on a 2013 MacBook Pro with a quad-core 2 GHz Intel Core i7 processor. While the example programs tested here are small, that is generally true of programs in the high-level domain-specific languages typical of the synthesis systems targeted by this method for code generation.

4 RELATED WORK

This section briefly surveys closely related prior work in the literature.

Partial evaluation. Partial evaluation itself has been the subject of extensive prior work [4]. The original paper outlined a high-level algorithm for partial evaluation and discusses the fundamental trade-offs required in a practical implementation. The paper also suggests the famous application of specialization for compiler-compilers.

Partial evaluation for implementation of domain-specific languages. Past work has investigated applying partial evaluation to the efficient implementation of domain-specific languages [2]. The motivation of this work is performance of executing programs in a domain-specific languages with the same host language, where this report emphasizes the need to translate the residual program to a different target language. This work addressed issues of recursion and termination by using a dependently typed language, in which all programs are total.

GPU compilation of interpreted languages with partial evaluation. Past work has also investigated the use of partial evaluation for cross-language compilation [3]. This work applied partial evaluation to specialize run-time hot paths of a program in an interpreted, dynamically typed language into efficient code for execution on graphics-processing hardware (*e.g.*, with CUDA or OpenCL). Unlike the work presented in this report, this work...

5 CONCLUSION

This section reviews key lessons learned by the author in this project and summarizes the main points of this work with some closing remarks.

5.1 Lessons Learned

Mismatch between the source language, Racket, and target language, OCaml, introduced a lot of implementation complexity. In particular, bridging differences in the language's static semantics was a significant challenge. For OCaml, this meant effecting implementing a shallow embedding of Racket. Initially, the author had unsuccessfully tried two other target languages, first C and then Rust, beforehand. The C language was a poor fit, because of the simple type system and lack of first-class functions. At first, Rust was a promising target, but the affine type system (*i.e.*, the value ownership mechanism) prevented any simple syntax translation that the author could think of. Seemingly innocuous design decisions in the host language, like mutable let-bound variables, could significantly complicate the syntax translation phase. Always reifying the creation of mutable references with Racket boxes permitted a common strategy to handle them and potentially side-effecting calls to library subroutines.

Recursion and termination were incredibly difficult issues to handle and constituted the majority of the author's debugging effort. The author found it difficult to conjecture and reason about possible strategies for unrolling recursive calls. At many points, it seemed as though adding one condition would cause some tests to pass and cause others to diverge or generate unacceptably poor output. Once the author settled on an appropriate recursion strategy, implementing it correctly was a further challenge. While structural recursion is a perfectly clear high-level concept, an implementation must attend to numerous low-level subtleties: tracking arguments to recursive functions, handling nested recursive functions, making repeated evaluation behave consistently, what free variables are permitted in the argument, and more.

Testing this system also presented an unexpected challenge. Despite a seemingly large test suite, most bugs were left undetected by them. The correctness of individual features and units of functionality was rarely problematic; rather, problems lay hidden in subtle edge cases of the interaction of these features and units of functionality. Covering such cases with tests would have increased the size of the test suite from linear in the number of features to quadratic. For a manually maintained test suite, this blow-up is a high cost.

5.2 Future Work

Extensibility. There is a high degree of regularity in implementation effort for many features of the host language. A mechanism for regular extension could avoid duplicating the effort for each new language primitive and even simplify the code required for support of the current ones. Moreover, users could then extend the code-generation system to better support their key libraries or simply to automate any marshalling to and from the dynamically typd code.

Infrastructure. Integration of this system for code generation with the real Racket language would make it much more useful to programmers. In particular, this would enable seamless use with solver-aided domain-specific languages built in the Rosette language, an extension of Racket. While the standard implementation of the Racket language does support the necessary deep integration, the functionality was too complex for the author to manage in the span of a course project.

Static semantics. Translating from a dynamically typed language like Racket to a statically typed language like OCaml introduces significant linguistic overhead. In terms of efficiency, this is likely not a significant concern, as modern compilers for functional languages are effective at optimizing away the associated runtime overhead when possible. However, the dynamic typing does impede integration with existing libraries in the target language, by requiring the programmer to write stubs that marshal data between the dynamic type and the underlying static types. Translation to a language like Rust requires even more careful consideration and management of differences in static semantics.

5.3 Closing Remarks

This report has presented an automatic method for code generation from an interpreter of the source language. This method is easy to use, requiring only the source program and the interpreter of the domain-specific language. This method generates efficient code, free of literal fragments of the original program's syntax tree and the interpreter of the domain-specific language. This method also permits integration with libraries in the target language. Finally, code generation by this method is fast.

REFERENCES

[1] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-Guided Synthesis. In *Dependable Software Systems Engineering*.

[2] Edwin C. Brady and Kevin Hammond. 2010. Scrapping Your Inefficient Engine: Using Partial Evaluation to Improve Domain-specific Language Implementation. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 297–308. DOI: http://dx.doi.org/10.1145/1863543.1863587

[3] Juan José Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. 2017. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *VEE*.

[4] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process - An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12 (1999), 381–391.

[5] Leo A. Meyerovich, Matthew E. Torok, Eric Atkinson, and Rastislav Bodik. 2013. Parallel Schedule Synthesis for Attribute Grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '13)*. ACM, New York, NY, USA, 187–196. DOI: http://dx.doi.org/10.1145/2442516.2442535

[6] Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013)*. ACM, New York, NY, USA, 135–152. DOI: http://dx.doi.org/10.1145/2509578.2509586