A Nanopass Framework for Commercial Compiler Development

Andrew W. Keep
University of Utah
akeep@cs.utah.edu

R. Kent Dybvig
Cisco Systems Inc.
dyb@cisco.com

Abstract

Contemporary compilers must typically handle sophisticated high-level source languages, generate efficient code for multiple hardware architectures and operating systems, and support source-level debugging, profiling, and other program development tools. As a result, compilers tend to be among the most complex of software systems. Nanopass frameworks are designed to help manage this complexity. A nanopass compiler is comprised of many single-task passes with formally defined intermediate languages. The perceived downside of a nanopass compiler is that the extra passes will lead to substantially longer compilation times. To determine whether this is the case, we have created a plug replacement for the commercial Chez Scheme compiler, implemented using an updated nanopass framework, and we have compared the speed of the new compiler and the code it generates against the original compiler for a large set of benchmark programs. This paper describes the updated nanopass framework, the new compiler, and the results of our experiments. The compiler produces faster code than the original, averaging 15–27% depending on architecture and optimization level, due to a more sophisticated but slower register allocator and improvements to several optimizations. Compilation times average well within a factor of two of the original compiler, despite the slower register allocator and the replacement of five passes of the original 10 with over 50 nanopasses.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Translator writing systems and compiler generators

General Terms Languages

Keywords Compiler, Nanopass, Scheme

1. Introduction

A compiler is typically structured as a series of passes, each analyzing source or intermediate code for an input program and possibly producing new intermediate code, with the final pass producing assembly or machine code for a target architecture. Historically, compilers employed no more than a few passes, because each pass involved reading source or intermediate code from cards, tape, or disk.

Contemporary compilers, on the other hand, take advantage of large physical memories and virtual address spaces to keep intermediate representations in memory. At the same time, contemporary compilers are substantially more complex, having been called upon to handle larger, higher-level source languages, to generate efficient code for multiple hardware architectures and operating systems, to support automatic storage management, threading, and other run-time activities, and to support debugging, profiling, and other development tools. While it might be possible to support the additional complexity with the same small number of passes, it is no longer important or desirable to do so.

An attractive alternative is to separate a compiler into many, perhaps dozens, of single-task passes. This improves modularity, potentially making the compiler easier to maintain and extend. Constructing a compiler with many single-task passes, however, can require excessive boilerplate code to recur through unchanging forms, increasing compiler size and thus failing to decrease maintenance and extension overhead. If intermediate languages are not well specified, having many passes also increases the risk of errors due to inconsistencies among passes.

A nanopass framework is designed to address these challenges with a domain-specific language (DSL) that supports formally defined intermediate languages, a convenient pattern matching syntax implemented via inexpensive record dispatch, and automatic generation of boilerplate code. When a paper on the first nanopass framework was accepted for ICFP 2004 [20], however, the program committee required the authors to refocus the paper on education because they did not believe the nanopass methodology was suitable for commercial compilers. They were principally concerned with the compile-time overhead of repeated traversals. This concern was impossible to refute at the time because the prototype infrastructure and implementation were not mature enough to put to the test.

This paper describes a nanopass infrastructure that *is* suitable for developing commercial compilers. To establish the infrastructure's suitability for commercial compiler development, we set out to develop a compiler that is 100% compatible with the existing commercial Chez Scheme [5] compiler, produces code that is at least on par with the original compiler, and does so with compile times that are within a factor of two of the original compiler, despite our separate ambition to incorporate a new, slower register allocator. An extensive test suite, used to test the original compiler, is used to demonstrate the new compiler is compatible with the original, while a large set of benchmarks is used to establish its performance. On the benchmarks, the new compiler produces code that is 15–

```
(define-language Lsrc
  (terminals
    (uvar (x))
    (primitive (pr))
    (datum (d)))
  (Expr (e body)
    x
    (quote d)
    (if e0 e1 e2)
    (begin e* ... e)
    (lambda (x* ...) body)
    (let ([x* e*] ...) body)
    (letrec ([x* e*] ...) body)
    (set! x e)
    (pr e* ...)
    (call e e* ...)             => (e e* ...)))
```

Figure 1. The `Lsrc` language

27% faster than the original compiler, depending on architecture and optimization level, with compile times that are well within the factor of two budget.

This paper is organized as follows. Section 2 describes the new nanopass framework and compares it with the original framework. Section 3 describes the differences between the new and original Chez Scheme compiler. Section 4 evaluates the run-time and compile-time performance of the new compiler against the original compiler. Section 5 presents related work, and Section 6 concludes.

2. The nanopass framework

A nanopass framework is a domain-specific language (DSL) for writing compilers. It provides two main syntactic forms: `define-language` for formally defining the intermediate-language grammars and `define-pass` for specifying passes that operate over these intermediate languages. Formally defining the intermediate languages allows the framework to fill in boilerplate code in passes, permits passes to check that output-language terms are well formed, and allows the framework to represent language terms as records internally while maintaining an S-expression pattern-matching and template-construction syntax.

The new nanopass framework builds on the prototype described by Sarkar et al. [20]. This section briefly describes the new framework and differences between this framework and the prototype. More information can be found in the first author's dissertation [14]. Examples are extracted from a student compiler used in a course on compiler implementation, since the Chez Scheme compiler is not open-source.

2.1 Defining languages

Languages definitions are similar to context-free grammars, in that they are composed of a set of terminals, a set of nonterminal symbols, a set of productions for each nonterminal, and a start symbol. An intermediate language definition for a simple variant of the Scheme programming language, post macro expansion, might look like `Lsrc` in Figure 1.

The `Lsrc` language consists of three terminals (listed in the `terminals` form) and a single nonterminal.

For each terminal, the compiler writer must supply a corresponding predicate. The framework adds a `?` to the terminal name to determine the predicate name. In this case, the nanopass framework expects `uvar?`, `primitive?`, and `datum?` to be lexically visible

```
(define-language L1
  (extends Lsrc)
  (terminals
    (- (datum (d)))
    (+ (constant (c))))
  (Expr (e body)
    (- (quote d))
    (+ (quote c))))
```

Figure 2. The `L1` language

where `Lsrc` is defined. Each terminal clause lists one or more meta-variables, used to refer to the terminal in nonterminal productions. For instance, `x` refers to a `uvar`.

The `Lsrc` language declares the nonterminal `Expr`. Nonterminals specify a name, a set of meta-variables, and a set of grammar productions. The `Expr` nonterminal has two meta-variables, `e` and `body`. These meta-variables, like the terminal meta-variables, are used to represent the nonterminal in a production.

Productions follow one of three forms: a single meta-variable, an S-expression that starts with a keyword, or an S-expression that does not start with a keyword (referred to as an *implicit* production). Productions cannot include keywords past the initial keyword.

The `x` production is the only single-meta-variable production and indicates that a `uvar` is an `Expr`. The only implicit S-expression production is `(pr e* ...)`, which specifies a primitive call with zero or more arguments. (The `...` following `e*` indicates that `e*` contains a list of `Expr`, the `*` suffix is used by convention to indicate plurality.) The `(call e e* ...)` production indicates a procedure call, and the `call` keyword differentiates it from a primitive call production. The `=> (e e* ...)` syntax indicates a *pretty* form for the production. The `define-language` form defines an unparser, and the unparser uses the pretty form when unparsing this production. The remaining productions correspond to the Scheme syntax that they represent.

2.2 Extending languages

The first pass of our student compiler is a simple expander that produces `Lsrc` language forms from S-expressions. The next pass expands complex quoted datum constants into code to construct these constants at load time.

The compiler writer could fully specify the output language, as we did with `Lsrc`. Fully specifying each language, however, results in verbose source code, particularly in a compiler with many intermediate languages. To avoid this, the framework supports a language extension form that succinctly describes only changes from one language to another. Figure 2 shows the output language.

The `L1` language removes the `datum` terminal and replaces it with the `constant` terminal. It also replaces the `(quote d)` production with a `(quote c)` production to indicate that only constants are allowed in the `quote` form.[1] The `extends` clause indicates a language extension form. Terminals are removed using the `-` clause and added using the `+` clause. Productions in a nonterminal are also removed using the `-` clause and added using the `+` clause.

2.3 Defining passes

The pass in Figure 3 converts an input program from the `Lsrc` intermediate language to the `L1` intermediate language. This pass

[1] If we failed to replace the `(quote d)` form, it would result in an error, since the `d` meta-variable is not bound in the new language.

```
(define-pass convert-complex-datum : Lsrc (x) -> L1 ()
  (definitions
    (define const-x* '())
    (define const-e* '())
    (define datum->expr
      (with-output-language (L1 Expr)
        (lambda (d)
          (cond
            [(pair? d) `(cons ,(datum->expr (car d))
                              ,(datum->expr (cdr d)))]
            [(vector? d)
             (let ([n (vector-length d)])
               (if (fxzero? n)
                   `(make-vector (quote 0))
                   (let ([t (unique-name 't)])
                     `(let ([,t (make-vector
                                  (quote ,n))])
                        (begin
                          ,(map (lambda (i v)
                                  `(vector-set! ,t
                                     (quote ,i)
                                     ,(datum->expr v)))
                                (iota n)
                                (vector->list d)) ...
                          ,t)))))]
            [else `(quote ,d)])))))
  (Expr : Expr (ir) -> Expr ()
    [(quote ,d) (guard (not (constant? d)))
     (let ([t (unique-name 't)])
       (set! const-x* (cons t const-x*))
       (set! const-e* (cons (datum->expr d) const-e*))
       t)])
  (let ([x (Expr x)])
    (if (null? const-x*)
        x
        `(let ([,const-x* ,const-e*] ...) ,x))))
```

Figure 3. The `convert-complex-datum` pass

removes the structured quoted datum by making the construction of the data explicit. To avoid constructing these constants more than once at run time, the pass also lifts datum creation to the start of the program.

A pass definition starts with a name and a signature. The signature specifies the input language (in this case, `Lsrc`) and list of formals (`x`) followed by the output language (`L1`) and a list of extra return expressions (`()`).

Following the name and signature, this pass specifies definitions for `const-x*`, `const-e*`, and `datum->expr` in the `definitions` clause. These definitions are scoped at the same level as the transformers in the pass. The `const-x*` and `const-e*` variables are initialized to null. The `const-x*` is extended with a new uvar and `const-e*` is extended with a new `Expr` each time a structured quoted datum is encountered. The `datum->expr` procedure recursively processes a structured quoted datum and produces the `L1 Expr` needed to construct it. Recursion terminates when a quoted constant is found.

Next, a transformer from the input nonterminal `Expr` to the output nonterminal `Expr` is defined. The transformer is named `Expr` and has a signature similar to that of the pass, with an input-language nonterminal and list of formals followed by the output-language nonterminal and list of extra-return-value expressions.

The transformer has a single clause that matches a quoted datum and uses a *guard* to ensure it is a pair or a vector. The `define-pass` macro autogenerates clauses matching the other input-language forms and producing equivalent output-language forms.

Each user-supplied clause consists of an input pattern, an optional `guard` clause, and one or more expressions that specify zero or more return values. The input pattern is derived from the productions specified in the input language. Pattern variables are unquoted, i.e., preceded by `,`. For instance, the clause for the `quote` production matches the pattern `(quote ,d)` and binds `d` to the `datum` specified by the `quote` form.

The output-language expression is constructed using a quasiquoted template. In the example, the quoted output-language expression is in the `datum->expr` procedure. Here, quasiquote, (`), is rebound to a form that constructs language forms based on the template, and unquote (`,`), is used to escape back into Scheme. The `,(datum->expr (car d))` thus puts the result of the recursive call to `datum->expr` into the output-language `(cons ,e0, e1)` form.

Following the `Expr` transformer is the body of the pass, which calls `Expr` to transform the `Lsrc Expr` term into an `L1 Expr` term and wraps the result in a `let` expression if any structured quoted datums are found in the program that is being compiled.

2.4 Comparison with the prototype nanopass framework

The prototype nanopass framework demonstrated that a nanopass framework is a viable approach to writing compilers, but only the first half of the student compiler was ever implemented using the framework. As such, the prototype framework has some rough edges that needed to be smoothed out in order to implement a replacement for the Chez Scheme compiler. The new nanopass framework focused on improvements in two areas, usability and performance.

On the usability side, the new nanopass framework introduces new features and improves error reporting. In the new framework, language definitions are no longer restricted to the top-level and can appear anywhere a Scheme definition can appear. Passes can be defined without an input language or output language, allowing a pass to take a non-language term as input or generate a non-language term as output. This allows passes to create general parsers, predicates, and code generators. Passes can also take additional arguments and return additional values, allowing information to flow through a pass without being encoded in the language term. Language terms can be constructed outside of a pass using the `with-output-language` form, as shown in Figure 3 and matched outside a pass using the `nanopass-case` form. The catamorphisms [17] syntax, used to recur on a sub-form of a language term in a pattern, supports passing extra arguments to transformers expecting more than one argument.

Error reporting has been improved, both to make messages easier to understand and to include source information for where the error occurred. Language definitions perform better checking, e.g., ensuring meta-variables are not repeated, removed productions existed in the base language, and the fields of a production are uniquely named.

On the performance side, the new nanopass framework uses an integer tag to perform pattern matching, which is slightly faster than the record dispatch previously used. In transformers, the order of clauses is respected, so that programmers can place clauses that are more likely to match first. The new framework also generates less code, so compiling a compiler generated with the nanopass framework is faster.

3. The new Chez Scheme compiler

Chez Scheme is a commercial Scheme compiler for R6RS Scheme with extensions, first released in 1985 [5] and under continuous development and improvement since. The compiler is written in Scheme and is comprised of 10 large, multipurpose passes. The compiler is almost absurdly fast, able to compile its own source code in roughly three seconds on contemporary hardware. The compiler is also designed to generate efficient code.

3.1 Workings of the existing Chez Scheme compiler

The compiler begins with a `syntax-case` expander, extended with a module system and R6RS libraries [6, 8, 12, 23]. The expander produces a simplified core language with `letrec`, `letrec*`, and `case-lambda` as binding forms, quoted constants, primitive references, procedure calls, variable references, and a handful of other forms. The first pass records source information used for debugging and profiling. The next pass places validity checks for variable references bound by `letrec` and `letrec*` [13, 24]. The next pass is the source optimizer [22] and can be run one or more times or not at all, depending on the options set in the compiler. A pass for handling `letrec` and `letrec*` [13, 24] is run at least once and is run after each run of the source optimizer. After this, either the interpreter is invoked to interpret the program or the back-end compiler is invoked to finish compilation and either execute the resulting code or write the machine code to the file system.

When the back-end compiler is invoked, the code is still in roughly the same form as the input language, although `letrec*` has been eliminated and `letrec` bindings bind only unassigned variables to `case-lambda` expressions. The compiler performs assignment conversion, closure conversion, various optimizations, and further code simplifications and replaces primitives in the source language with an internal set of simpler, although still higher level than assembly language, primitives. It also performs register allocation using a linear-scan style register allocator with a lazy register save and restore strategy [3], and generates code using destination-driven code generation [7].

The back end consists of five passes. The first pass, `cp1`, begins the process of closure conversion, recognizes loops, recognizes direct application of λ-expressions, begins handling multiple return value calls, sets up the foreign and foreign callable expressions, and converts primitive calls into a set of slightly lower-level inline calls. The next pass, `cpr0`, begins the register allocation process, determines the actual free variables of closures after performing closure optimization, performs assignment conversion, makes explicit all arguments to inlined primitives, and flags tail calls and loops. The register allocator reserves the architectural stack register, a Scheme frame pointer register, a thread context register pointer, and at least two temporary registers, depending on the target machine architecture, for use by the assembler. After this, the pass `cpr1` assigns variables to their initial register homes. The `cpr2` pass finishes register allocation, generating register saves and restores across non-tail calls and removing redundant register bindings. Finally, the `cp2` pass finishes compilation, converting the higher-level inlined primitives into a set of high-level instruction inlines that the assembler can convert into machine instructions for the target platform. Each of these back-end passes performs several optimizations in addition to those mentioned above, and some optimizations span multiple passes.

3.2 Workings of the new compiler

The new compiler starts with the same set of front-end passes, updated to use the nanopass framework.

The back end of the new compiler diverges significantly from that of the original compiler. Where the original compiler is structured as a set of multipurpose passes that each performs several tasks, the new compiler is organized as approximately 50 passes, with each pass completing primarily one task, using approximately 35 nanopass languages.

These passes implement most of the optimizations from the original compiler and improve on some, including support for implicit cross-library optimization, improvements to closure optimization [15], and improved handling of procedures that return multiple values.

The other big difference between the original compiler and the new compiler is the use of a graph-coloring register allocator. This change necessitated several other changes in the compiler, including the expansion of code into a near-assembly language form much earlier in the compiler, so that all of the temporaries that might be needed for the final code to conform to the operand requirements of the machine can be met. It also means that a full live analysis must be performed to compute the conflict graph needed by the register allocator. In the original compiler, the cost of the live analysis is largely avoided by tracking the liveness of registers, rather than the liveness of variables. Additionally, in the original compiler, primitive expansion is delayed until code generation at the cost of reserving two or more registers, depending on the target architecture.

The benefit of using a graph-coloring register allocator is that it packs spilled variables tighter on the frame, makes better use of registers, and generally produces more compact code. The new register allocator also uses move biasing to avoid frame-to-frame moves. This contributes to the generated code's faster run time, at the cost of substantially more compile-time overhead. Both the original and new compilers try to make good use of variable saves and restores around non-tail calls, allowing call-live variables to be accessed from a register, rather than from the frame. The original compiler follows a lazy-save strategy [3], while the new compiler attempts to get similar results by using a heuristic that estimates the cost of saving and restoring versus the cost of spilling to a frame location permanently. This is one place where the new compiler sometimes underperforms the original compiler.

Not all of the optimizations provided by the original compiler are provided by the new compiler. The most significant missing optimization is block allocation of closures. When several closures are created at the same time, a single allocation is performed to allocate the space for the entire group of closures. A more general block allocation optimization is planned for the new compiler but has not yet been implemented.

3.3 Ensuring compatibility

Over the course of Chez Scheme's development, an extensive suite of unit, functional, and regression tests has been developed to ensure that the compiler conforms to the relevant Scheme standards and documentation for Chez Scheme extensions. Chez Scheme is also bootstrapped, and the first test of the compiler is to compile itself and verify that code generated for the compiler is consistent with each run of the compiler. The new compiler passes all of these tests.

4. Evaluation of the new Chez Scheme compiler

4.1 Comparing the speed of generated code

We compare the performance of the original and new compilers on a set of benchmarks that includes the R6RS benchmarks [4]; a

set of benchmarks, including some larger benchmarks, used to test the source optimizer; and a set of additional benchmarks used for performance regression testing.

The benchmark data was generated on an Intel Core i7-3960X with two CPUs, six cores per CPU, and 64 GB of RAM. The tests were conducted at both optimize level 2 (run-time type-checking enabled) and optimize level 3 (run-time type-checking disabled), using both the 32-bit and 64-bit instruction sets. Table 1 shows the average improvement in run time for benchmarks compiled with the new compiler over the original compiler. In the worst case, optimize level 3 with the 64-bit instruction set, the benchmarks still run 15.0% faster on average.

	x86 machine	x86_64 machine
Optimize Level 2	26.6%	22.0%
Optimize Level 3	22.3%	15.0%

Table 1. Average improvement in benchmark run times

Two of the benchmarks, `similix`, a self-applicable partial evaluator, and `softscheme`, a benchmark that performs soft typing, make use of the compiler during the run of the benchmark. This negatively affects the run time of these benchmarks and the overall average. Table 2 shows the normalized run time of these benchmarks on both the 32-bit and 64-bit versions of the compiler and at both optimization levels.

Opt. level	Machine type	Similix	Soft Scheme
2	x86	1.54	1.27
2	x86_64	1.41	1.32
3	x86	1.29	1.22
3	x86_64	1.39	1.24

Table 2. Normalized run times of `similix` and `softscheme`

Outside of these two outliers, the performance ranges on the 32-bit version between 0.417 and 1.05 at optimize level 2 and between 0.457 and 1.05 at optimize level 3. It ranges on the 64-bit version between 0.519 and 1.03 at optimize level 2 and between 0.489 and 1.14 at optimize level 3.

As discussed in Section 3.2, the new compiler contains several improvements over the original compiler, and each of these contributes to the better performance of the benchmarks. The biggest contributing factor, and the one that is consistent in all of the benchmarks, is the graph-coloring register allocator. The graph-coloring register allocator makes more efficient use of available registers, which is particularly helpful on the 32-bit Intel target, where only eight registers are available.

4.2 Comparing compilation speed

We set out with the goal of implementing a new compiler that ran within a factor of two of the original compiler. The extra compile-time budget was intended to allow the more expensive graph-coloring register allocator to be incorporated into the new compiler. Ideally, we would have created a nanopass compiler as near in function to old compiler as possible for apples-to-apples compile-time comparisons, but we did not have the resources to create two new compilers.

The two compilers were tested by compiling each benchmark five times and averaging compile times. The compile times are normalized, using the original compiler as a base. Table 3 presents the normalized numbers by machine type and optimization level. In the worst case (the 64-bit version at optimize level 2) the average compile time is a factor of 1.75 longer on the new compiler than on the original compiler. This time is well within our factor of two goal. Nevertheless, further tuning is still possible and will make the compile times for the new compiler even closer to those of the original compiler.

	x86 machine	x86_64 machine
Optimize Level 2	1.71	1.75
Optimize Level 3	1.64	1.71

Table 3. Normalized compile times of benchmarks

The compile time varies from one benchmark to another. On the 32-bit version, the normalized time ranges from a factor of 1.00 to 4.44 at optimize level 2 and 0.968 to 3.13 at optimize level 3. On the 64-bit version, normalized time ranges from a factor of 1.00 to 4.73 at optimize level 2 and 1.00 to 3.80 at optimize level 3.

It is natural to assume that the variance in compile times is related to the increased computational complexity of the graph-coloring register allocator; however, this does not seem to be the case. Figure 4 shows the normalized times versus the lines of expanded source code. The number of lines of expanded source code is determined by pretty printing the results of the expander run on each benchmark and then counting the number of line breaks. Overall, there does not seem to be a direct relationship between the number of lines of expanded code and the normalized compile time. An understanding of why this occurs might lead to overall improvements in compile time.

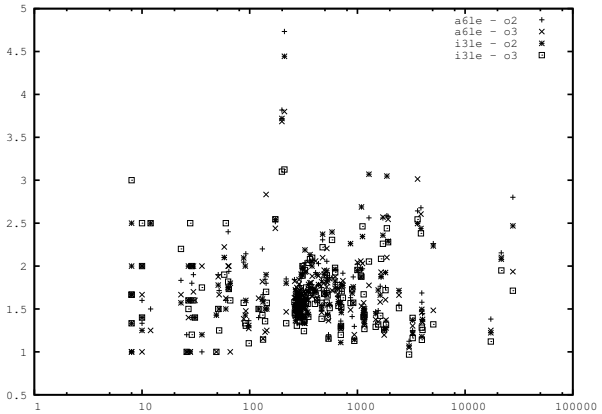

Figure 4. Normalized compile time vs. expanded source code size

5. Related work

Stratego/XT [2] is a DSL for writing source-to-source transformations. It provides a set of combinators for matching and rebuilding abstract syntax tree (AST) forms as well as strategies for performing different traversals of the AST. Pattern matching and AST construction in Stratego is different from that in the nanopass framework, because if construction of an AST fails, the next pattern will be tried, whereas the nanopass framework makes a committed (deterministic) choice with explicit guards for extra-grammatical checks, which simplifies debugging and improves compiler speed.

The JastAdd [9] system allows for the construction of modular and extensible compilers using Java's object-oriented class hierarchy, along with an external DSL to specify the AST and analysis and transformations on the AST. Each type of node in the AST has an

associated class that encapsulates the transformations on the node type. Method dispatch and aspects are used with the visitor pattern to implement passes instead of pattern matching.

SableCC [11] is a system for building compilers and interpreters in Java. It provides a set of tools for writing the lexer and parser for the language and a method for defining multiple passes. Similar to JastAdd, it works on an AST represented by Java classes. Also similar to JastAdd, it uses the visitor pattern to implement the language transformations.

POET combines a transformation language and an empirical testing system to allow transformation to be tuned [26]. Although POET allows for some generic manipulation of an AST, it is largely focused on targeting specific regions of source code to be tuned. It can parse fragments of source code and operate on the AST fragment, preserving unparsed code across the transformation.

The ROSE [18] compiler infrastructure provides a C++ library for source-to-source transformation, along with front-ends and back-ends for both C/C++ and Fortran. Internally, ROSE represents source code using the ROSE Object-Oriented IR, with transformations written in C++. Although there is nothing specific that ties the ROSE transformation framework to C/C++ or Fortran, there are no tools to easily add new front-ends and back-ends, limiting the usefulness of ROSE for other languages.

The Rhodium framework takes a different approach from the other tools described here. Instead of using term rewriting, Rhodium bases transformations on data-flow equations and provides a framework for proving the soundness of transformations [16, 21]. The framework has also been extended to support inferring optimizations from the data-flow semantics defined by the compiler writer. The data-flow facts are defined over a C-like intermediate representation. This might be a good complement to the nanopass framework.

CodeBoost [1] is a more targeted tool. Although it was originally developed to support the Sophus numerical library, CodeBoost provides a simple way to write compiler transformations within C++ code. It is implemented using Stratego but provides an array of tools to make writing C++ code transformations easier.

The template-based metacompiler (Tm) [19] provides a macro language, similar to M4, for generating data structures and transformations that can be expanded in a language agnostic way. Tm provides tree-walker and analyzer templates and an existing C back-end for easier use.

Pavilion [25] is a DSL for writing analysis and optimization passes to improve the efficiency of generic programming in C++. The declarative language extends regular expressions with intersection and complement operators, variable quantification, path quantification, function definition, and native language access to Scheme to provide powerful matching during analysis and transformation.

Yoko [10] is a Haskell module for writing functions that transform an input type to a similar output type that requires only the interesting cases be specified, similar to how the nanopass framework operates over languages. The module builds on generic programming techniques to provide the `hcompos` function. The `hcompos` function takes the user-specified cases and autogenerates the necessary clauses to handle constructors from the input type not specified by the programmer, matching them with constructors of the output type with the same name.

6. Conclusion

The new Chez Scheme compiler demonstrates that a nanopass compiler can perform on par with a more traditionally structured compiler. Despite a more expensive register allocator and the replacement of the five back-end passes of the original 10 passes in the compiler with around 50 nanopasses, the new compiler still averages compile times that are within a factor of two of the original compiler. The new compiler also generates more efficient code, showing between a 15% and 26.6% improvement on a set of benchmarks, depending on the optimize level and the target architecture. The new nanopass framework made it easier to implement the new compiler and will make it easier to maintain and extend in the future.

Acknowledgments

Different parts of Keep's effort on this work were partially supported by the DARPA programs APAC and CRASH. Comments from Dan Friedman, Ryan Newton, and the anonymous reviewers led to several improvements in this papers presentation.

References

[1] O. S. Bagge, K. T. Kalleberg, M. Haveraaen, and E. Visser. Design of the CodeBoost transformation system for domain-specific optimisation of C++ programs. In D. Binkley and P. Tonella, editors, *Proc. 3rd International Workshop on Source Code Analysis and Manipulation*, SCAM '03, pages 65–75, Amsterdam, Sept. 2003. IEEE Computer Society Press. URL `http://www.codeboost.org/papers/codeboost-scam03.pdf`.

[2] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52–70, June 2008. URL `http://dx.doi.org/10.1016/j.scico.2007.11.003`.

[3] R. G. Burger, O. Waddell, and R. K. Dybvig. Register allocation using lazy saves, eager restores, and greedy shuffling. In *Proc. ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 130–138, New York, 1995. ACM. URL `http://doi.acm.org/10.1145/207110.207125`.

[4] W. D. Clinger. Description of benchmarks, 2008. URL `http://www.larcenists.org/benchmarksAboutR6.html`.

[5] R. K. Dybvig. *Chez Scheme Version 8 User's Guide*. Cadence Research Systems, 2009.

[6] R. K. Dybvig. *The Scheme Programming Language*. MIT Press, fourth edition, 2009.

[7] R. K. Dybvig, R. Hieb, and T. Butler. Destination-driven code generation. Technical Report TR302, Indiana University, February 1990.

[8] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in Scheme. *LISP and Symbolic Computation*, 5(4):295–326, Dec. 1992. URL `http://dx.doi.org/10.1007/BF01806308`.

[9] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Science of Computer Programming*, 69(1-3): 14–26, Dec. 2007. .

[10] N. Frisby, A. Gill, and P. Alexander. A pattern for almost homomorphic functions. In *ACM SIGPLAN Workshop on Generic Programming*, 09/2012 2012.

[11] E. Gagnon and L. Hendren. SableCC, an object-oriented compiler framework. In *Proc. 26th Conference on Technology of Object-Oriented Languages*, TOOLS '98, pages 140–154, 1998. .

[12] A. Ghuloum and R. K. Dybvig. Implicit phasing for R6RS libraries. In *Proc. 12th ACM SIGPLAN International Conference on Functional Programming*, pages 303–314, New York, 2007. ACM.

[13] A. Ghuloum and R. K. Dybvig. Fixing letrec (reloaded). In *Proc. 2009 Workshop on Scheme and Functional Programming*, Scheme '09, pages 57–65, 2009.

[14] A. W. Keep. *A Nanopass Framework for Commercial Compiler Development*. PhD thesis, Indiana University, Feb. 2013.

[15] A. W. Keep, A. Hearn, and R. K. Dybvig. Optimizing closures in O(0) time. In *Proc. 2012 Workshop on Scheme and Functional Programming*, Scheme '12, 2012.

[16] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '05, pages 364–377, New York, 2005. ACM. URL `http://doi.acm.org/10.1145/1040305.1040335`.

[17] E. Meijer, M. M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proc. 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 124–144, London, UK, 1991. Springer-Verlag. ISBN 3-540-54396-1. URL `http://dl.acm.org/citation.cfm?id=645420.652535`.

[18] D. Quinlan, M. Schordan, Q. Yi, and B. R. de Supinski. Semantic-driven parallelization of loops operating on user-defined containers. In *Lecture Notes in Computer Science*, volume 2958/2004 of *Lecture Notes in Computer Science*, pages 524–538. Springer Berlin / Heidelberg, 2004.

[19] C. V. Reeuwijk. Tm: A code generator for recursive data structures. *Software: Practice and Experience*, 22(10):899–908, Oct. 1992. URL `http://dx.doi.org/10.1002/spe.4380221008`.

[20] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. In *Proc. 9th ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 201–212, New York, 2004. ACM. URL `http://doi.acm.org/10.1145/1016850.1016878`.

[21] E. R. Scherpelz, S. Lerner, and C. Chambers. Automatic inference of optimizer flow functions from semantic meanings. In *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 135–145, New York, 2007. ACM. URL `http://doi.acm.org/10.1145/1250734.1250750`.

[22] O. Waddell and R. K. Dybig. Fast and effective procedure inlining. In *Proc. 4th International Symposium on Static Analysis*, SAS '97, pages 35–52, London, 1997. Springer-Verlag.

[23] O. Waddell and R. K. Dybvig. Extending the scope of syntactic abstraction. In *Proc. 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 203–215, New York, 1999. ACM. URL `http://doi.acm.org/10.1145/292540.292559`.

[24] O. Waddell, D. Sarkar, and R. K. Dybvig. Fixing letrec: A faithful yet efficient implementation of Scheme's recursive binding construct. *Higher-order and Symbolic Computation*, 18(3/4):299–326, December 2005.

[25] J. J. Willcock. *A Language for Specifying Compiler Optimizations for Generic Software*. Doctoral dissertation, Indiana University, Bloomington, Indiana, USA, Dec. 2008.

[26] Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. POET: Parameterized optimizations for empirical tuning. Technical Report CS-TR-2006-006, University of Texas - San Antonio, 2006. URL `http://www.cs.utsa.edu/{~}qingyi/papers/poet-lang.pdf`.