A Nanopass Infrastructure for Compiler Education

Dipanwita Sarkar
Indiana University
dsarkar@cs.indiana.edu

Oscar Waddell
Abstrax, Inc.
waddell@abstrax.com

R. Kent Dybvig
Indiana University
dyb@cs.indiana.edu

ABSTRACT

Compilers structured as a small number of monolithic passes are difficult to understand and difficult to maintain. Adding new optimizations often requires major restructuring of existing passes that cannot be understood in isolation. The steep learning curve is daunting, and even experienced developers find it hard to modify existing passes without introducing subtle and tenacious bugs. These problems are especially frustrating when the developer is a student in a compiler class.

An attractive alternative is to structure a compiler as a collection of many small passes, each of which performs a single task. This "micropass" structure aligns the actual implementation of a compiler with its logical organization, simplifying development, testing, and debugging. Unfortunately, writing many small passes duplicates code for traversing and rewriting abstract syntax trees and can obscure the meaningful transformations performed by individual passes.

To address these problems, we have developed a methodology and associated tools that simplify the task of building compilers composed of many fine-grained passes. We describe these compilers as "nanopass" compilers to indicate both the intended granularity of the passes and the amount of source code required to implement each pass. This paper describes the methodology and tools comprising the nanopass framework.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Translator writing systems and compiler generators*

General Terms

Design, languages, reliability

Keywords

Compiler writing tools, nanopass compilers, domain-specific languages, syntactic abstraction

1. INTRODUCTION

Production compilers often exhibit a monolithic structure in which each pass performs several analyses, transformations, and optimizations, both related and unrelated. An attractive alternative, particularly in an educational setting, is to structure a compiler as a collection of many small passes, each of which performs a small part of the compilation process. This separation of concerns aligns the actual implementation of a compiler with its logical organization, yielding a more readable and maintainable compiler. Bugs that arise are more easily isolated to a particular task. Writing individual passes is easier since new code need not be grafted onto existing passes nor wedged between two logical passes that would be combined in a monolithic structure.

A few years ago we switched to this "micropass" structure in our senior- and graduate-level compiler courses. Students are supported in the writing of their compilers by several tools: a pattern matcher with convenient notations for recursion and mapping, a set of macros that can be used to expand the output of each pass into executable code, a reference implementation of the compiler, a suite of (terminating) test programs, and a driver. The driver runs the compiler on each of the programs in the test suite and evaluates the output of each pass to verify that it returns the same result as the reference implementation. Intermediate-language programs are all represented as s-expressions, which simplifies both the compiler passes and the driver.

The switch to the micropass methodology and the tools that support it have enabled our students to write more ambitious compilers. Each student in our one-semester compiler class builds a 50-pass compiler from the s-expression level to Sparc assembly code for the subset of Scheme below.

```
expr  ⟶  constant
       |  (quote datum)
       |  var
       |  (set! var expr)
       |  (if expr expr)
       |  (if expr expr expr)
       |  (begin expr expr*)
       |  (lambda (var*) expr expr*)
       |  (let ((var expr)*) expr expr*)
       |  (letrec ((var expr)*) expr expr*)
       |  (primitive expr*)
       |  (expr expr*)
```

The compiler includes several optimizations as well as a graph-coloring register allocator. Students in the graduate course implement several additional optimizations. The

Week 1: simplification
- verify-scheme[1]
- rename-var
- remove-implicit-begin[1]
- remove-unquoted-constant
- remove-one-armed-if
- verify-a1-output[1]

Week 2: assignment conversion
- remove-not
- mark-assigned
- optimize-letrec[2]
- remove-impure-letrec
- convert-assigned
- verify-a2-output[1]

Week 3: closure conversion
- optimize-direct-call
- remove-anonymous-lambda
- sanitize-binding-forms
- uncover-free
- convert-closure
- optimize-known-call[3]
- uncover-well-known[2]
- optimize-free[2]
- optimize-self-reference[2]
- analyze-closure-size[1]
- lift-letrec
- verify-a3-output[1]

Week 4: canonicalization
- introduce-closure-primitives
- remove-complex-constant
- normalize-context
- verify-a4-output[1]

Week 5: pointer encoding/allocation
- specify-immediate-representation
- specify-nonimmediate-representation

Week 6: start of UIL compiler
- verify-uil

Week 7: introducing labels and temps
- remove-complex-opera*
- lift-letrec-body
- introduce-return-point
- verify-a7-output[1]

Week 8: virtual registerizing
- remove-nonunary-let
- uncover-local
- the-return-of-set!
- flatten-set!
- verify-a8-output[1]

Week 9: brief digression
- generate-C-code[4]

Week 10: register allocation setup
- uncover-call-live[2]
- optimize-save-placement[2]
- eliminate-redundant-saves[2]
- rewrite-saves/restores[2]
- impose-calling-convention
- reveal-allocation-pointer
- verify-a10-output[1]

Week 11: start of register allocation
- uncover-live-1
- uncover-frame-conflict
- strip-live-1
- uncover-frame-move
- verify-a11-output[1]

Week 12: setting up call frames
- uncover-call-live-spills
- assign-frame-1
- assign-new-frame
- optimize-fp-assignments[2]
- verify-a12-output[1]

Week 13: introducing spill code
- finalize-frame-locations
- eliminate-frame-var
- introduce-unspillables
- verify-a13-output[1]

Week 14: register assignment
- uncover-live-2
- uncover-register-conflict
- verify-unspillables[1]
- strip-live-2
- uncover-register-move
- assign-registers
- assign-frame-2
- finalize-register-locations
- analyze-frame-traffic[1]
- verify-a14-output[1]

Week 15: generating assembly
- flatten-program
- generate-Sparc-code

Table 1: Passes assigned during a recent semester, given in running order and grouped roughly by week and primary task. Notes: [1]Passes supplied by the instructor. [2]Challenge-assignment passes required only of graduate students, not necessarily during the week shown. [3]Actually written during Week 4. [4]Pass not included in the final compiler. During Week 6, students also had an opportunity to turn in updated versions of earlier passes. Week 9 was a short week leading up to spring-break week. Most of the passes are run exactly once; the passes that comprise the main part of the register and frame allocator are repeated until all variables have been given register or frame homes.

passes included in the compiler are listed in Table 1. Due to space limitations, we cannot go into the details of each pass, but the pass names are suggestive of their roles in the compilation process.

The micropass methodology and tools are not without problems, however. The repetitive code for traversing and rewriting abstract syntax trees can obscure the meaningful transformations performed by individual passes. In essence, the sheer volume of code for each pass can cause the students to lose the forest for the trees. Also, although we have learned the importance of writing out grammars describing the output of each pass, as documentation, the grammars are not enforced, and it is easy for an unhandled specific case to fall through to a more general case, resulting in either confusing errors or malformed output to trip up later passes. Finally, the resulting compiler is slow, which leaves students with a mistaken impression about the speed of a compiler and the importance thereof.

To address these problems, we have developed a "nanopass" methodology and a domain-specific language for writing nanopass compilers. A nanopass compiler differs from a micropass compiler in three ways: the intermediate-language grammars are formally specified and enforced, each pass needs to contain traversal code only for forms that undergo meaningful transformation, and the intermediate code is represented more efficiently as records, although all interaction with the programmer is still via the s-expression syntax. We use the word "nanopass" to indicate both the intended granularity of passes and the amount of source code required to implement each pass.

The remainder of this paper describes the nanopass methodology and supporting tools. Section 2 introduces our methodology for building nanopass compilers. Section 3 describes tools for building nanopass compilers. Section 4 highlights the different features of the framework and their underlying implementation, with the help of some example language and pass definitions. Section 5 surveys related work. Section 6 concludes with a discussion of future work.

2. NANOPASS METHODOLOGY

In the nanopass framework, a compiler is composed of many fine-grained passes, each operating on programs in a well-specified input language and producing programs in a well-specified output language. A discipline we find helpful in maintaining this fine granularity is to require that each pass perform a single specific task to simplify, verify, convert, analyze, or improve the code.

A simplification pass reduces the complexity of subsequent passes by translating its input into a simpler intermediate language, e.g., removing the primitive not from the language. A verification pass checks compiler invariants that are not easily expressed within the grammar, e.g., that all bound variables are unique. A conversion pass makes explicit an abstraction that is not directly supported by the low-level target language, e.g., converting basic blocks to a linear instruction stream by inserting branches. An analysis pass collects information from the input program and records that information by annotating the output program, e.g., annotating each `lambda` expression with its set of free variables. An improvement pass attempts to optimize the run time or resource utilization of the program.

We require a verification or improvement pass (or group of related analysis and improvement passes) to produce a program in the same intermediate language as its input program so that we may selectively enable or disable individual checks or optimizations. Enabling verification passes can help to identify bugs during the development of upstream passes. Verification passes may be disabled to improve compiler speed. The ability to disable or enable individual optimizations at will simplifies development of tools that automate regression testing with various permutations of compiler switches. Such testing may uncover cases where a routine optimization masks bugs in seldom-executed portions of another pass. Bugs of this kind are otherwise especially difficult to locate since they typically surface only in programs sufficiently complex to defeat the compensating optimizations. Selectively disabling optimizations is also an easy way to support a range of compiler switches that trade compile-time speed for code quality.

We find it helpful to codify the transformation performed by each pass using formal grammars to describe the input and output languages of the pass. Care is taken to make these grammars precise. For example, a pass may transform the input program so that particular forms are eliminated or appear in more limited contexts. Although the output of the pass is still a valid program in the input language, we define a new output-language grammar that explicitly deletes or restricts the affected form.

Intermediate-language grammars are specified formally via language definitions (Section 3.1). Language definitions play an important role when defining passes, and precise grammars are also useful documentation. The transformation performed by a pass can often be understood by comparing the grammars of the input and output languages of the pass.

It may appear that much of the compiler complexity has been pushed into the intermediate-language definitions, particularly given our emphasis on precise grammars. Since each pass performs one, well-specified transformation, however, changes in the intermediate language from one pass to the next are usually slight. Often we can define new languages via inheritance, described in Section 3.2, expressing the differences between the languages more concisely. Moreover, many passes, including improvement and verification passes, operate on the same source and output languages. So, although we write many passes and emphasize precise grammars, we write fewer language definitions than passes, and many of the ones we do write are concisely specified in terms of an earlier language.

In addition, the compilation system compares the results obtained by evaluating the output of each pass against the results produced by a reference implementation. This practice helps to isolate correctness-preservation failures to a particular pass. Having isolated an offending pass, we can view intermediate-language programs in a readable s-expression form when tracing through the output of the pass searching for the cause of the failure.

3. NANOPASS TOOLS

This section describes tools for defining new intermediate languages and compiler passes. These tools comprise a domain-specific language for writing nanopass compilers and are implemented as extensions to the host language, Scheme, via the `syntax-case` macro system [6]. This language-embedding approach provides access to the full host language for defining auxiliary procedures and data structures, which are particularly useful when writing involved passes, such as a register allocation pass. The host language also provides the evaluator that we use to evaluate the output of each compiler pass during development.

3.1 Defining intermediate languages

Intermediate language definitions take the following form.

(**define-language** *name* { **over** *tspec*$^+$ }
  where *production*$^+$)

The optional *tspec* declarations specify the terminals of the language and introduce metavariables ranging over the various terminals. Each *tspec* is of the form

(*metavariable*$^+$ **in** *terminal*)

where the *terminal* categories are declared externally. A metavariable declaration for x implicitly specifies metavariables of the form xn, where n is a numeric suffix. Each *production* corresponds to a production in the grammar of the intermediate language.

A production pairs a nonterminal with one or more alternatives, with an optional set of metavariables ranging over the nonterminal.

({ *metavariable*$^+$ **in** } *nonterminal alternative*$^+$)

Productions may also specify elements that are common to all alternatives using the following syntax.

({ *metavariable*$^+$ **in** } (*nonterminal common*$^+$)
  alternative$^+$)

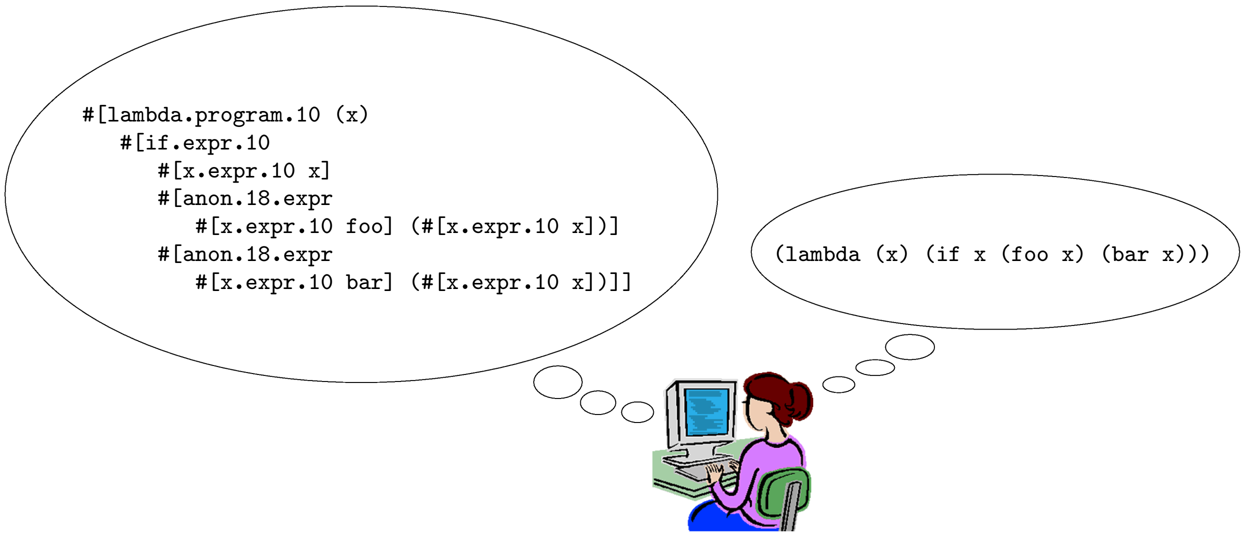
#[lambda.program.10 (x)
   #[if.expr.10
      #[x.expr.10 x]
      #[anon.18.expr
         #[x.expr.10 foo] (#[x.expr.10 x])]
      #[anon.18.expr
         #[x.expr.10 bar] (#[x.expr.10 x])]]
(lambda (x) (if x (foo x) (bar x)))

Illustration 1: All compiler-writer interactions are via the s-expression syntax.

Common elements may be used to store annotations, e.g., source information or analysis byproducts, that are common to all subforms of the intermediate language.

Each *alternative* is a metavariable or parenthesized form declaring an intermediate language construct, followed by an optional set of production properties *property*$^+$. Parenthesized forms usually begin with a keyword and contain substructure that usually includes metavariables specifying the language category into which each subform falls. At most one alternative of a production may be a parenthesized form that does not begin with a keyword, allowing the intermediate language to include applications using the natural s-expression syntax. Each *property* is a *key*, *value* pair. Properties are used to specify semantic, type, and flow information for the associated alternative.

Figure 1 shows a simple language definition and the grammar it implicitly defines. It defines metavariables `x`, `b`, and `n` ranging over variables, booleans, and integers, and defines three productions. The first production defines `Program` as an `Expr`. The second defines metavariables `e` and `body` ranging over `Expr` and declares that `Expr` is a boolean, integer, variable reference, `if` expression, `seq` expression, `lambda` expression, or application. The third defines metavariable `c` ranging over `Command` and declares that `Command` is a `set!` command or `seq` command.

The semantics of each intermediate language form may be specified implicitly via its natural translation into the host language, if one exists. In Figure 1, this implicit translation suffices for booleans, numbers, variable references, `lambda`, `set!`, `if`, and applications. For `seq` expressions, the translation is specified explicitly using the `=>` (translates-to) property. Implicit and explicit translation rules establish the meaning of an intermediate language program in terms of the host language, which is an aid to understanding intermediate language programs and provides a mechanism whereby the output of each pass can be verified to produce the same results as the original input program while a compiler is being debugged. Explicit translations can become complex, in which case we often express the translation in terms of a syntactic abstraction (macro) defined in the host language.

From a compiler writer's point of view, a language definition specifies the structure of an intermediate language in terms of the familiar s-expression syntax (Illustration 1). All of the compiler writer's interactions with the intermediate language occur via this syntax. Internally, however, intermediate language programs are represented more securely and efficiently as record structures.

Intermediate language programs are also evaluable in the host language, using the translation properties attached to production alternatives. To support these differing views of intermediate language programs, a language defined via **define-language** implicitly defines the following items:

1. a set of record types representing the abstract syntax trees (ASTs) of intermediate-language programs,
2. a mapping from s-expressions to record structure,
3. a mapping from record structure to s-expressions, and
4. a mapping from s-expression patterns to record structure

These products are packaged within a module and may be imported where they are needed. The remainder of this section describes these products in more detail.

3.1.1 Record-type definitions

The language definition automatically generates a set of record definitions as shown in Figure 2. A base record type is constructed for the language along with a subtype for each nonterminal. The subtype for each nonterminal declares the common elements for that nonterminal. A new record type is also created for each alternative as a subtype of the corresponding nonterminal. The **define-language** form given in Figure 1 defines a record type for `L0`, subtypes of this type for `Program`, `Expr`, and `Command`, and subtypes of these for each of their alternatives. For example, the record type for `(if e1 e2 e3)` is a subtype of the record type for `Expr`, which is in turn a subtype of the record type for `L0`. The record type for `if` contains three `Expr` fields. Where ellipses are used in the **define-language** syntax, the field contains a list of elements. For example, an alternative

```
(define-language L0 over
  (x in variable)
  (b in boolean)
  (n in integer)
 where
  (Program
    Expr)
  (e body in Expr
    b
    n
    x
    (if e1 e2 e3)
    (seq c1 e2) => (begin c1 e2)
    (lambda (x ...) body)
    (e0 e1 ...))
  (c in Command
    (set! x e)
    (seq c1 c2) => (begin c1 c2)))
```

```
⟨L0⟩        ⟶   ⟨Program⟩
⟨Program⟩   ⟶   ⟨Expr⟩

⟨Expr⟩      ⟶   ⟨boolean⟩
             |   ⟨integer⟩
             |   ⟨var⟩
             |   (if ⟨Expr⟩ ⟨Expr⟩ ⟨Expr⟩)
             |   (seq ⟨Command⟩ ⟨Expr⟩)
             |   (lambda (⟨var⟩*) ⟨Expr⟩)
             |   (⟨Expr⟩ ⟨Expr⟩*)

⟨Command⟩   ⟶   (set! ⟨var⟩ ⟨Expr⟩)
             |   (seq ⟨Command⟩ ⟨Command⟩)
```

Figure 1: A simple language definition and the corresponding grammar

```
(define-record L0 ())
(define-record program L0 ())
(define-record expr L0 ())
(define-record command L0 ())
(define-record b.expr expr (b))
(define-record n.expr expr (n))
(define-record x.expr expr (x))
(define-record if.expr expr (e1 e2 e3))
(define-record seq.expr expr (c1 c2))
(define-record lambda.expr expr (xs body))
(define-record app.expr expr (e0 es))
(define-record set!.command command (x e))
(define-record seq.command command (c1 c2))
```

Figure 2: Record definitions generated for L0

```
(define (parser-lang.L0 s-exp)
 (define (parse-program s-exp) ...)
 (define (parse-expr s-exp)
  (if (pair? s-exp)
      (cond
       ...
       [(and (eq? 'seq (car s-exp))
             (= 3 (length s-exp)))
        (make-seq.expr.L0.6
         (parse-command (cadr s-exp))
         (parse-expr (caddr s-exp)))]
       ...
       [else
         (make-anon.7
         (parse-expr (car s-exp))
         (map parse-expr (cdr s-exp)))])
       (cond
        [((boolean? s-exp)
          (make-b.expr.L0.1 s-exp)]
        ...
        [else (error ---)]))))
 (define (parse-command s-exp) ...))
```

Figure 3: Parser for a language L0

`(let ((x e) ...) body)` would declare a record type with two fields, the first of which contains a list of `(x e)` records associating program variables with `Expr` records. Strong typing of record constructors ensures that ASTs are well-formed by construction.

3.1.2 Parser

Each language definition produces a parser capable of transforming s-expressions to the corresponding record structure representing the same abstract syntax tree. The parser for the first input language serves as the first pass of the compiler (after lexical analysis and parsing) and provides the record-structured input required by subsequent passes of the compiler. Providing parsers for each intermediate language aids debugging by allowing the compiler-writer to obtain inputs suitable for arbitrary passes of the compiler.

Figure 3 shows part of the code for the parser that is generated by the language definition for `L0` in Figure 1. The code for the parser mirrors the language definition. The language definition produces a set of mutually recursive procedures, `parse-program`, `parse-expr` and `parse-command`, each handling all the alternatives of the corresponding nonterminal. The nonterminal parsers operate by recursive descent on list structured input following the grammar. For example, the `parse-expr` case for `seq` first verifies whether the list has three elements and begins with the keyword `seq`, then calls directly to `parse-command` for first subform and to `parse-expr` for the second subform. An error is signaled if there is no match.

Since parenthesized forms are disambiguated by the beginning keyword, at most one parenthesized form per nonterminal can begin with a metavariable, i.e., a nonkeyword.

3.1.3 Unparser

The unparser converts the AST records to their corresponding host-language executable forms. Like the parser this also serves as a good debugging aid by allowing the compiler-writer to view the output of any pass in host-language form. It enables the compiler-writer to trace and manually translate programs, e.g., during the exploratory phase of the development of a new optimization. Each record-type definition stores the parenthesized form and the host-language form for the alternative. The host-language form, if different from the parenthesized form, is expressed as the translates-to production property in the language definition, as in the case of `(seq c1 e2) => (begin c1 e2)` in Figure 1. Each record type stores the information required to unparse instances of itself. As a result, all languages share one unparse procedure.

Since the record type stores both the parenthesized form and the host-language form of the alternative, the unparser can also translate the record structures into their parenthesized forms, thus allowing the compiler writer to pretty-print the output.

3.1.4 Partial Parser

The partial parser is used to support input pattern matching and output construction, which are described in Section 3.3.1 and Section 3.3.2.

The partial parser translates s-expression syntax representing an input pattern or output template into its corresponding record structure. The variable parts of the s-expression syntax are converted to a special list representation that is later used to generate code for the pass.

For example, the partial parser parses the s-expression syntax `(let ((,x ,e) ...) (foo ,x ...))` into the following.

```
#[let-record
  #[anon.1-record
    (maplist (list (x variable)(e expr)))]
  #[app-record
   #[ref-record (foo variable)]
   #[ref-record (x variable)]]]
```

The actual list representations generated are slightly more elaborate than the ones shown above and will be discussed later in Section 4.2. The structures produced by the partial parser are not visible to the compiler-writer, but are used to generate the code for matching the given pattern and constructing the desired output.

3.2 Language inheritance

Consecutive intermediate languages are often closely related due to the fine granularity of the intervening passes. To permit more concise specification of these languages, the **define-language** construct supports a simple form of inheritance via the **extends** keyword, which must be followed by the name of a base language, already defined.

(**define-language** *name* **extends** *base*
  { **over** { *mod tspec* }$^+$ }
  { **where** { *mod production* }$^+$ })

The terminals and productions of the base language are copied into the new language subject to modifications in the **over** and **where** sections of the definition, either of which may be omitted if no modifications to that section are necessary. Each *mod* is either `+`, which adds a new terminal or production, or `-`, which removes the corresponding terminal or production(s). The example below defines a new language `L1` derived from `L0` (Figure 1) by removing the `boolean` terminal and `Expr` alternative and replacing the `Expr if` alternative with an `Expr case` alternative.

```
(define-language L1 extends L0
  over
   - (b in boolean)
  where
   - (Expr b (if e1 e2 e3))
   + (default in Expr
       (case x (n1 e1) ... default)))
```

Language `L1` could serve as the output of a conversion pass that makes language-specific details explicit en route to a language-independent back end. For example, C treats zero as false, while Scheme provides a distinct boolean constant `#f` representing false. Conditional expressions of either language could be translated into `case` expressions in `L1` with language-specific encodings of false made explicit.

Language inheritance is mainly a notational convenience. A new language definition is generated from the definition of the parent language and the implementation from that point is the same as that for **define-language**. The parent and child languages do not share any record definitions.

3.3 Defining passes

Passes are specified using a `define-pass` construct that names the input and output languages and specifies transformation functions that map input-language forms to output-language forms.

(`define-pass` *name input-language* `->` *output-language*
  transform ...)

Some passes are run purely for effect, e.g., to collect and record information about variable usage. For such passes, the special output language `void` is used. Similarly, the special output language `datum` is used when a pass traverses an AST to compute some more general result, e.g., an estimate of object code size.

Each *transform* specifies the transformer's name, a signature describing the transformer's input and output types, and a set of clauses implementing the transformation.

(*name* : *nonterminal arg* ... `->` *val val* ...
  [*input-pattern* { *guard* } *output-expression*] ...)

The input portion of the signature lists a nonterminal of the input language followed optionally by the types of any additional arguments expected by the transformer. The output portion lists one or more result types. Unless `void` or `datum` is specified in place of the output language, the first result type is expected to be an output-language nonterminal.

Each clause pairs an *input-pattern* with a host-language *output-expression* that together describe the transformation of a particular input-language form. Input patterns are specified using an s-expression syntax that extends the syntax of alternatives in the production for the corresponding input-language nonterminal as described in Section 3.3.1. Output expressions may contain *templates* for constructing output-language forms using syntax which extends that of alternatives in the production for the corresponding output-language nonterminal. The extended syntax of input patterns and output templates is described in sections 3.3.1 and 3.3.2. The optional *guard*, if present, is a host-language expression that imposes additional constraints on matching.

Often, a pass performs nontrivial transformation for just a few forms of the input language. In such cases, the two intermediate languages are closely related and the new language can be expressed using language inheritance. When two intermediate languages can be related by inheritance, a pass definition can specify transformers for only those forms that have undergone change, leaving the implementation of other transformers to a *pass expander*. The pass expander completes the implementation of a pass by consulting the definitions of the input and output languages. Strong typing of passes, transformers, and intermediate languages helps the pass expander to automate these simple transformations. The pass expander is an important tool for keeping pass specifications concise.

3.3.1 Matching input

When invoked, a transformer matches its first argument, which must be an AST, against the input pattern of each clause until a match is found. Clauses are examined in order, with user-specified clauses preceding any clauses inserted by the pass expander. This process continues until the input pattern of some clause is found to match the AST and the additional constraints imposed by guard expressions and pattern variables, described below, are satisfied. When a match is found, the corresponding output expression is evaluated to produce a value of the expected result type. An error is signaled if no clause matches the input.

Input patterns are specified using an s-expression syntax that extends the syntax of alternatives for the corresponding nonterminal with support for pattern variables. Subpatterns are introduced by commas, which indicate, by analogy to `quasiquote` and `unquote` [5], portions of the input form that are not fixed. For example, `(seq (set! ,x ,n) ,e2)` introduces three subpatterns binding pattern variables `x`, `n`, and `e2`. Metavariables appearing within patterns impose further constraints on the matching process. Thus the preceding pattern matches only those inputs consisting of a `seq` form whose first subform is a `set!` form that assigns a number to a variable, and whose second subform is an `Expr`.

Pattern variables are used within input patterns to constrain the matching of subforms of the input AST. Within subpatterns, pattern variables are used to bind matching subforms of the input to program variables that may be referenced within output expressions and to match the results of structural recursion on subforms of the input. The various forms that subpatterns may take are summarized below, where the metavariable a ranges over alternate forms of an input-language nonterminal A, and the metavariable b ranges over alternate forms of an output-language nonterminal B.

1. The subpattern `,`a matches if the corresponding input subform is a form of A, and binds the pattern variable a to the matching subform.

2. The subpattern `,[`f `:` a `->` b`]` matches if the corresponding input subform is a form of A, and the result obtained by invoking the transformer f on that subform is a form of B. If the match succeeds, the pattern variable a is bound to the matching input subform, and b is bound to the result that is obtained by invoking f.

3. The subpattern `,[`a `->` b`]` is equivalent to the subpattern `,[`f `:` a `->` b`]` if f is the sole transformer mapping $A \to B$.

4. The subpattern `,[`b`]` is equivalent to the subpattern `,[`a `->` b`]` if the corresponding input subform is a form of A.

Transformers may accept multiple arguments and return multiple values. To support these transformers, the syntax `,[`f `:` a x `...` `->` b y `...]` may be used to supply additional arguments x ... to f and bind program variables y ... to the additional values returned by f. The first argument must be an AST as must the first return value, unless `void` or `datum` is specified as the output type. Subpatterns 3 and 4 are extended in the same way to support the general forms `,[`a x `...` `->` b y `...]` and `,[`b y `...]`. Pattern variables bound to input subforms may be referenced among the extra arguments x ... within structural-recursion patterns 2 and 3 above. Metavariables are used within patterns to guide the selection of appropriate transformers for structural recursion.

When present, the optional guard expression imposes additional constraints on the matching of the input subform prior to any structural recursion specified by the subpattern. Pattern variables bound to input forms are visible within guard expressions. The mechanism just described is consistent with the behavior of transformers, although, in fact, transformers locate candidate clauses using efficient type dispatch supported by the AST record structures. To avoid duplicate evaluation, the pass expander commonizes cases that scrutinize the results of structural recursion.

3.3.2 Constructing output

When the input to a transformer matches the input pattern of one of the clauses, the corresponding output expression is evaluated in an environment that binds the subforms matched by pattern variables to like-named program variables. For example, if an input language record representing `(set! y (f 4))` matches the pattern `(set! ,x ,e)`, the corresponding output expression is evaluated in an environment that binds the program variables `x` and `e` to the records representing `y` and `(f 4)`. When a pattern variable is followed by an ellipsis (`...`) in the input pattern, the corresponding program variable is bound to a matching list of records.

New abstract syntax trees are constructed via output templates specified using an overloaded `quasiquote` syntax that constructs record instances rather than list structure. Where commas (i.e., `unquote` forms) do not appear within an output template, the resulting AST has a fixed structure. An expression prefixed by a comma within an output template is a host-language expression that must be evaluated to obtain an AST to be inserted as the corresponding subform of the new AST being produced. For example, `` `(if (not ,e1) ,e2 ,e3) `` constructs a record representing an `if` expression with an application of the primitive `not` as its test and the values of program variables `e1`, `e2`, and `e3` inserted where indicated. The constructor applications implied by this output template are essentially the following.

```
(make-if (make-primapp 'not e1) e2 e3)
```

The record constructors available within output templates are determined by the output language specified in the pass definition. Instantiating the output template must produce an AST representing a form of the output nonterminal for the transformer containing the clause.

A subtle point arises when writing a pass that maps programs in one language to programs in another closely related language. When the input and output languages are different, a clause `[,x x]` would produce a record of the wrong type, since it matches and returns an input-language record. The clause must instead be written `[,x `,x]` so that the output-language constructors are invoked to produce an AST of the expected type. Fortunately, the pass expander automates such trivial transformations. By nature, more substantial transformations virtually always employ output templates and so are seldom affected by this distinction.

Where ellipses follow an `unquote` form in an output template, the host-language expression must evaluate to a list of objects. For example, `` `(begin ,e ... ,c) `` requires that `e` be bound to a list of `Expr` forms. Lists of records are often convenient when interfacing with common subroutines, e.g., to partition a set of expressions according to specific properties. Because `quasiquote` is overloaded to construct record structures of the output language, an expression `` `(,e ...) `` may return a record representing a procedure application, rather than a list of `e` records. Thus it is convenient that an input pattern `(let ((,x ,e) ...) ,body)` binds the program variables `x` and `e` to lists of `variable` and `Expr` records. When more complex list structures are desired, list-processing tools of the host language are convenient. For example, within an output expression associated with the preceding `let` pattern, `(map list x e)` could be used to build a list structure that could not otherwise be constructed via the overloaded `quasiquote`.

4. EXAMPLES

This section shows a few sample languages and compiler passes defined via **define-language** and `define-pass`. It also shows portions of the code generated by the pass expander for representative passes and briefly discusses how they are generated. For clarity, the following examples use fully qualified names of the form *name*`.`*language*.

4.1 Removing a primitive

Language `L2` defined in Figure 4 is a simple language of expressions. The objective of the pass `remove-not` in Figure 5 is to eliminate the primitive `not` from programs in this language. When `not` is used as the test part of an `if` expression, we express the inversion in flow of control simply by swapping the consequent and alternative expressions. In all other contexts, we replace calls to `not` with an equivalent `if` expression that does the conversion explicitly. This pass operates as a source-to-source transformation. The pass definition deals with only those language forms that undergo transformation and relies upon the pass expander to supply code for the remaining cases. In particular, the pass expander supplies code for the case where the test part of an `if` expression is not a `primapp` or is an application of a primitive other than `not`.

```
(define-language L2 over
  (x in variable)
  (b in boolean)
  (n in integer)
  (pr in primitive)
 where
  (Program
    Expr)
  (e body in Expr
    b
    n
    x
    (if e1 e2 e3)
    (lambda (x ...) body)
    (primapp e0 e1 ...)
    (e0 e1 ...)))
```

Figure 4: A simple language of expressions

```
(define-pass remove-not L2 -> L2
  (process-expr : Expr () -> Expr ()
    [(if (not ,[e1]) ,[e2] ,[e3])
     `(if ,e1 ,e3 ,e2)]
    [(primapp ,pr ,[e])
     (eq? 'not pr)
     `(if ,e #f #t)]))
```

Figure 5: A pass that performs source-to-source transformation

As mentioned in Section 3.1.4, we generate the pattern-matching code by first parsing the input pattern with the partial parser of the input language in order to identify the input and output constraints that govern the pattern-matching process. For example, when given the input pattern

```
(if (not ,[e1]) ,[e2] ,[e3])
```

the partial parser generates the following record.

```
#[if.L2
  #[primapp.L2
    #[var.L2 'not]
    (e1 #f expr.L2 process-expr)]
  (e2 #f expr.L2 process-expr)
  (e3 #f expr.L2 process-expr)]
```

By comparing this record with the `if` alternative in the definition of `L2`, we see that the first subpattern imposes a constraint on the input, since `primapp` is a subtype of `expr`. The list structures contained within this record impose additional constraints on the matching of the results from implicit recursion, and they specify the targets of these recursive calls, as well as the names of the pattern variables to which the results are to be bound. For example, the pattern variable `e2` is to be bound to the result of calling `process-expr` on `(primapp.L2-e (if.L2-e1 ir))`, provided that the result satisfies the `expr.L2?` predicate. Here `ir` is the input to the pass and *record-fieldname* is the syntax for field accessors. Normally, the input to a pass is generated by either a parser or another pass and is thus well-formed by construction. Therefore, the individual fields of the input record need not be checked unless they indicate a type or pattern more specific than what is expected according to the input-language definition.

A guard expression, if present, is evaluated only if the input record satisfies the input constraints of the pattern. If the guard condition is satisfied, pattern matching continues with the input and output pattern variables bound to the corresponding pieces of the input or results from implicit recursion. Input pattern variables can be referenced within guard expressions, as shown below.

```
[(primapp ,pr ,[e] ...)
 (eq? pr 'not)
 `(if ,e #f #t)]
```

The guard expression, `(eq? pr 'not)` is evaluated in an environment where `pr` is bound to `(primapp.L2-pr ir)`.

To generate the code that is responsible for constructing the output record, we parse the output template with the partial parser of the output language and examine the resulting record structure to identify the record-constructors that need to be called. For the pattern in the output template `(if ,e1 ,e3 ,e2)`, the partial parser generates the following record.

```
#[if.L2 e1 e3 e2]
```

Since output templates impose no constraints, the partial parser suppresses that information for the pattern variables `e1`, `e2`, and `e3`.

The following excerpt shows the expansion of the `if` clause specified by the programmer in Figure 5.

```
(and (if.L2? ir)
     (primapp.L2? (if.L2-e1 ir))
     (let ([e1 (process-expr
                 (primapp.L2-e (if.L2-e1 ir)))]
           [e2 (process-expr (if.L2-e2 ir))]
           [e3 (process-expr (if.L2-e3 ir))])
       (or (and (expr.L2? e1)
                (expr.L2? e2)
                (expr.L2? e3)
                (make-if.L2 e1 e3 e2))
           (error ---))))
```

The following shows the expansion of the `primapp` clause in Figure 5.

```
(and (primapp.L2? ir)
     (let ([pr (primapp.L2-pr ir)])
       (and (primitive? pr)
            (eq? 'not pr)
            (let ([e (process-expr
                       (primapp.L2-e ir))])
              (or (and (expr.L2? e)
                       (make-if.L2 e '#f '#t))
                  (error ---))))))
```

4.2 Beta reduction

The objective of the `optimize-direct-call` pass in Figure 7 is to replace what would otherwise be a call to an anonymous `lambda` with a simple `let` expression. In practical terms, this transformation avoids an unnecessary heap allocation, an indirect jump, and, when there are free variables in the body, some additional indirect memory references. If the expression is evaluated frequently, the savings can be significant. This pass translates expressions in language `L2` to expressions in language `L3`, which differs from `L2` only by the addition of a `let` form. `L3` is concisely defined in Figure 6 by extending the previously defined language `L2`.

```
(define-language L3 extends L2
  where
  + (Expr (let ((x e) ...) body)))
```

Figure 6: A language derived from another language

```
(define-pass optimize-direct-call L2 -> L3
  (process-expr : Expr () -> Expr ()
    [((lambda (,x ...) ,[body]) ,[e] ...)
     `(let ((,x ,e) ...) ,body)]))
```

Figure 7: A pass that performs beta reduction.

The input and output constraints and the output constructors in this example are generated from partially parsed records in the same way as described in Section 4.1. However, the patterns in this case are more involved. Consider the following clause from Figure 7.

```
[((lambda (,x ...) ,[body]) ,[e] ...)
 `(let ((,x ,e) ...) ,body)]
```

For the input pattern, the partial parser produces the following record.

```
#[app.L2
  #[lambda.L2
    (maplist-of (x variable.L2 #f #f))
    (body expr.L2 expr.L3 process-expr)]
  (maplist-of
    (e expr.L2 expr.L3 process-expr))]
```

`(maplist-of (x variable #f #f))` indicates that the pattern is to be matched against an input list containing an arbitrary number of items, where each item is a `variable` record that needs no implicit recursion and imposes no output type constraints. From this record, we generate code like the following to bind the pattern variables to the appropriate portions of the input and to the results of processing the input.

```
(let ([x (lambda.L2-x (app.L2-e0 ir))])
  ...
  (let ([e (map process-expr (app.L2-e1 ir))]
        [body (process-expr
                (lambda.L2-body ir))])
    ...))
```

For the output template, the partial parser produces the following record.

```
#[let.L3 (maplist-of (list-of x e)) body]
```

The list `(maplist-of (list-of x e))` representing the pattern ((,x ,e) ...) is also processed to generate code to reconstruct the list from the instantiated pattern variables x and e as shown below.

```
(let ([opfield.1
       (map (lambda (g0 g1)
              (cons g0 (cons g1 '())))
            x e)]
      [opfield.2 body])
  (make-let.L3 opfield1 opfield2))
```

4.3 Assignment conversion

The process of assignment conversion involves two passes. The first pass, mark-assigned, locates assigned variables i.e., those appearing on the left-hand side of an assignment, and marks them by setting a flag in one of the fields of the variable record structure. A second pass, convert-assigned, rewrites references and assignments to assigned variables as explicit structure accesses or mutations.

The mark-assigned pass runs for effect only on an input in language L4, which is may be derived from L0 by adding the forms let, primapp, and the terminal primitive, as shown below.

```
(define-language L4 extends L0
  over
   + (pr in primitive)
  where
   + (Expr
       (let ((x e) ...) body)
       (primapp pr e ...))
   + (Command
       (primapp pr e ...)))
```

Only one language form, set!, need be handled explicitly by this pass. If the input is a set! expression, the pass simply sets the assigned flag in the record structure representing the assigned variable.

```
(define-pass mark-assigned L4 -> void
  (process-command : Command -> void
    [(set! ,x ,[e])
     (set-variable-assigned! x #t)]))
```

Since convert-assigned removes the set! form, its output language, L5, is derived from its input language L4.

```
(define-language L5 extends L4
  where
   - (Command (set! x e)))
```

As shown in Figure 8, the convert-assigned pass introduces a let expression binding each assigned variable to a pair whose car is the original value of the variable, replaces each reference to an assigned variable with a call to car, and replaces each assignment with a call to set-car!. split-vars is an auxiliary procedure that introduces temporaries for assigned variables. This pass converts the L4 program

```
(lambda (a b)
  (seq
    (set! a b)
    a))
```

into the following L5 program

```
(lambda (t b)
  (let ((a (cons t #f)))
    (seq
      (set-car! a b)
      (car a))))
```

5. RELATED WORK

The Zephyr Abstract Syntax Description Language (ASDL) describes tree-like intermediate languages in a format that can be used to generate language-specific data structure declarations as well as language-specific functions that read and write these structures [16]. These functions provide functionality similar to the parsers and unparsers generated by our language definitions. ASDL focuses solely on intermediate representation and therefore does not integrate support for defining passes. We have introduced our own language for describing intermediate representations so that the pass expander can refer to these descriptions when filling in the details of passes.

The TIL compiler performs all optimizations on typed intermediate languages [11]. Tarditi, et. al, found that type-checking the output of each optimization pass helped to identify and eliminate bugs in the compiler. In our framework, type information could be encoded via properties in the language definition, and a verification pass could be inserted after each pass to typecheck its output. Our language definitions produce expanders that translate intermediate representations to semantically equivalent host-language programs. During development we often use this mechanism to evaluate the output of each pass and compare the results with those produced by a reference implementation. We could instead compare the results of static analysis, including type information, on the input and output programs after each pass. We already use verification passes during development to check other static properties.

Polyglot simplifies construction of compilers for source-level language extensions of Java [9]. A key design goal of Polyglot is to ensure that the work required to add new passes or new AST node types is proportional to the number of node types or passes affected. This goal is achieved through the use of some fairly involved OOP syntax and mechanisms. Our pass expander and our support for language inheritance approach the same goal with less syntactic and conceptual overhead.

Tm is a macro processor in the spirit of m4 that takes a source code template and a set of data structure definitions and generates source code [12]. Tree-walker and analyzer templates that resemble define-pass have been generated using Tm [13]. These templates are low-level relatives of define-pass, which provides convenient input pattern syntax for matching nested record structures and output template syntax constructing nested record structures. Tm data-structure definitions do not support extensible properties described in Section 3.1.

Some similarities also exist between the nanopass approach and approaches taken in the PFC and SUIF compilers. Like the Nanopass compiler, the PFC compiler [3] uses macro expansion to fill in boilerplate transformations. The SUIF system [1, 2] provides object-oriented tools for specifying intermediate language programs, including tools that allow compiler passes to focus on different aspects of an intermediate language program. The SUIF compiler has

```
(define-pass convert-assigned L4 -> L5
  (process-expr : Expr -> Expr
    [,x (variable-assigned x) `(primapp car ,x)]
    [(lambda (,x ...) ,[body])
     (let-values ([(xi xa xr) (split-vars x)])
       `(lambda (,xi ...)
          (let ((,xa (primapp cons ,xr #f)) ...)
            ,body)))])
  (process-command : Command -> Command
    [(set! ,x ,[e]) `(primapp set-car! ,x ,e)]))

(define split-vars
  (lambda (vars)
    (if (null? vars)
        (values '() '() '())
        (let-values ([(ys xas xrs) (split-vars (cdr vars))])
          (if (variable-assigned (car vars))
              (let ([tmp (make-variable 'tmp)])
                (values (cons tmp ys) (cons (car vars) xas) (cons tmp xrs))
                (values (cons (car vars) ys) xas xrs)))))))
```

Figure 8: Assignment conversion

one intermediate language format as opposed to the many intermediate languages encouraged by the Nanopass compiler. The nanopass approach achieves effects similar to these systems but extends them by formalizing the language definitions, including sufficient information in the language definitions to allow automated conversion to and from the host language, and separating traversal algorithms from intermediate language and pass definitions.

6. CONCLUSIONS

The nanopass methodology supports the decomposition of a compiler into many small pieces. This decomposition simplifies the task of understanding each piece and, therefore, the compiler as a whole. Adding new optimizations is easier; there is no need to "shoe-horn" an analysis or transformation into an existing monolithic pass. The methodology also simplifies the testing and debugging of a compiler, since each task can be tested independently, and bugs are easily isolated to an individual task.

The nanopass tools enable a compiler student to focus on concepts rather than implementation details while having the experience of writing a complete and substantial compiler. While it is useful to have students write out all traversal and rewriting code for the first few passes to understand the process, the ability to focus only on meaningful transformations in later passes reduces the amount of tedium and repetitive code. The code savings is significant for many passes, including the passes shown in Section 4. With our old tools, `remove-not` was the smallest pass at 25 lines; it is now 7 lines. Similarly, `convert-assigned` was 55 lines and is now 20 lines. On the other hand, the sizes of a few passes cannot be reduced. The code generator, for example, must explicitly handle every grammar element.

At present, the pass expander can fill in missing details only for passes implementing algorithms that are insensitive to the order in which the pass recurs on subforms. This is suitable for most passes, but not for passes that perform a flow-sensitive analysis or transformation, such as a live analysis. Such passes must presently be written out in full. A focus of our future research will be to extend language definitions to allow flow information to be incorporated and to extend pass definitions to allow a forward or backward flow-sensitive traversal algorithm to be specified.

Our experience indicates that fine-grained passes work extremely well in an educational setting. We are also interested in using the nanopass technology to construct production compilers, where the overhead of so many repeated traversals of the code may be unacceptable. Another focus of our future research will be to develop a pass combiner that can, when directed, fuse together a set of passes into a single pass, using deforestation techniques [15] to eliminate rewriting overhead.

The fine-grained nature of the passes may also tend to exacerbate *phase ordering problems* [7, 18]. A phase ordering problem occurs when no ordering of a set of improvement passes takes advantage of all optimization opportunities, because each optimization may lead to opportunities for some of the others. This problem can often be solved by iterating individual improvement passes until a fixed point is reached, but this solution may result in significant compile-time overhead. A more efficient solution is to combine many optimizations, or the analyses that enable them, into a single "super optimizer" that produces the same or better residual code in fewer iterations due to the synergy among optimizations at the subexpression level [4, 8, 10, 14, 17]. Unfortunately, super optimizers exhibit the undesirable monolithic structure that the nanopass framework is designed to avoid, and, in our experience, adding new optimizations to an existing super optimizer requires considerably more effort than writing them individually. The pass combiner hypothesized above may be able to exploit this synergy and allow improvement passes to be developed independently yet run as a single super optimizer.

Acknowledgments

Dan Friedman suggested the use of small, single-purpose passes and contributed to an earlier compiler based on this principle. Erik Hilsdale designed and implemented the matcher we used to implement micropass compilers. Our input-pattern subpatterns were inspired by Erik's matcher. Jordan Johnson implemented an early prototype of the nanopass framework. Dipanwita Sarkar was funded by a gift from Microsoft Research University Relations.

7. REFERENCES

[1] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Technical report, Stanford University, 2000.

[2] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. The SUIF2 compiler infrastructure. Technical report, Stanford University, 2000.

[3] J.R. Allen and K. Kennedy. Pfc: A program to convert fortran to parallel form. Technical Report MASC-TR82-6, Rice University, Houston, TX, 1982.

[4] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2), 1995.

[5] R. Kent Dybvig. *The Scheme Programming Language*. MIT Press, third edition, 2002.

[6] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[7] Wilf R. LaLonde and Jim des Rivieres. A flexible compiler structure that allows dynamic phase ordering. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 134–139, 1982.

[8] Sorin Lerner, David Grove, and Craig Chambers. Composing dataflow analyses and transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 270–282, 2002.

[9] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer-Verlag, 2003.

[10] Anthony Pioli and Michael Hind. Combining interprocedural pointer analysis and conditional constant propagation. Research Report 21532, IBM T. J. Watson Center, 1999.

[11] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 181–192, 1996.

[12] C. van Reeuwijk. Tm: a code generator for recursive data structures. *Software Practice and Experience*, 22(10):899–908, 1992.

[13] C. van Reeuwijk. Rapid and robust compiler construction using template-based metacompilation. In *Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 247–261. Springer-Verlag, 2003.

[14] Oscar Waddell and R. Kent Dybvig. Fast and effective procedure inlining. In Pascal Van Hentenryck, editor, *Fourth International Symposium on Static Analysis*, volume 1302 of *Lecture Notes in Computer Science*, pages 35–52. Springer-Verlag, 1997.

[15] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88: European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer-Verlag, 1988.

[16] D.C. Wang, A.W. Appel, J.L. Korn, and C.S. Serra. The zephyr abstract syntax description language. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, pages 213–228, 1997.

[17] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 3(2):181–210, 1991.

[18] D. Whitfield and M. L. Soffa. An approach to ordering optimizing transformations. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 137–146, 1990.