

Improving the Static Analysis of Embedded Languages via Partial Evaluation

David Herman
dherman@ccs.neu.edu

Philippe Meunier
meunier@ccs.neu.edu

College of Computer and Information Science
Northeastern University
360 Huntington Ave #202 WWH
Boston, MA 02115

Abstract

Programs in embedded languages contain invariants that are not automatically detected or enforced by their host language. We show how to use macros to easily implement partial evaluation of embedded interpreters in order to capture invariants encoded in embedded programs and render them explicit in the terms of their host language. We demonstrate the effectiveness of this technique in improving the results of a value flow analysis.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging—*debugging aids*; D.3.3 [Programming Languages]: Language Constructs and Features—*macros*; D.3.4 [Programming Languages]: Processors—*code generation, debuggers, interpreters*

General Terms

Reliability, Languages

Keywords

Partial evaluation, macros, embedded languages, value flow analysis

1 One Language, Many Languages

Every practical programming language contains small programming languages. For example, C's `printf` [18] supports a string-based output formatting language, and Java [3] supports a declarative sub-language for laying out GUI elements in a window. PLT Scheme [9] offers at least five such languages: one for formatting console output; two for regular expression matching; one for sending queries to a SQL server; and one for laying out HTML pages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.
Copyright 2004 ACM 1-58113-905-5/04/0009 ...\$5.00

In many cases, though not always, programs in these embedded special-purpose programming languages are encoded as strings. Library functions consume these strings and interpret them. Often the interpreters consume additional arguments, which they use as inputs to the little programs.

Take a look at this expression in PLT Scheme:

```
(regexp-match "http://([a-z.]*)/([a-z]*)/" line)
```

The function `regexp-match` is an *interpreter* for the regular expression language. It consumes two arguments: a string in the regular expression language, which we consider a program, and another string, which is that program's input. A typical use looks like the example above. The first string is actually specified at the call site, while the second string is often given by a variable or an expression that reads from an input port. The interpreter attempts to match the regular expression and the second string.

In PLT Scheme, the regular expression language allows programmers to specify subpatterns via parentheses. Our running example contains two such subexpressions: `([a-z.]*)` and `([a-z]*)`. If the regular expression interpreter fails to match the regular expression and the string, it produces `false` (`#f`); otherwise it produces a list with $n + 1$ elements: the first one for the overall match plus one per subexpression. Say `line` stands for

```
"http://aaa.bbb.edu/zzz/"
```

In this case, the regular expression matches the string, and `regexp-match` produces the list

```
(list "http://aaa.bbb.edu/zzz/"  
      "aaa.bbb.edu"  
      "zzz")
```

The rest of the Scheme program extracts the pieces from this list and computes with them.

The `regexp-match` expression above is a simplified excerpt from the PLT Web Server [12]. Here is a slightly larger fragment:

```
(let ([r (regexp-match  
        "http://([a-z.]*)/([a-z]*)/" line)])  
  (if r  
      (process-url (third r) (dispatch (second r)))  
      (log-error line)))
```

Notice how the `then`-clause of the `if`-expression extracts the second

and third elements from `r` without any checks to confirm the length of the list. After all, the programmer knows that if `r` is not false, then it is a list of three elements. The embedded program says so; it is a regular expression and contains two subexpressions.

Unfortunately, the static analysis tools for PLT Scheme cannot reason on both levels. MrFlow [20], a static debugger, uses a constraint-based analysis [22], a version of set-based analysis [2, 13, 10], to analyze the program and discover potential errors. If it finds one it can draw a flow graph from the source of the bad value to the faulty primitive operation. For the `let`-expression above, MrFlow finds that both `(second r)` and `(third r)` may raise runtime errors because `r` may not contain enough elements.

In this paper, we show how using Scheme macros to partially evaluate calls to embedded interpreters such as `regexp-match` greatly increases the precision of the static analysis. Since we use macros, library designers can easily implement the partial evaluation, rather than relying on the host language implementor as they must for ad-hoc solutions.

In Section 2 we give a brief overview of set-based analysis and MrFlow. In the next section we explain three examples of embedded languages and the problems they cause for MrFlow’s static analysis. We then present in Section 4 our general approach to solving those problems, based on macros. An overview of the macro system we use is given in Section 5. Section 6 then presents a general technique for translating embedded interpreters into macros. In Section 7, we explain the properties of the static analysis that enable it to find more results in partially evaluated code. Finally, in Section 8, we show how partially evaluating Scheme programs that contain embedded programs helps MrFlow in our three examples. Section 9 presents related work and we conclude in Section 10.

2 Set-Based Analysis

To explain how the results of a static analysis can be improved by using partial evaluation of embedded languages, we first need to describe such an analysis. MrFlow, a static analyzer for DrScheme, uses a set-based value flow analysis to compute an approximation of the values that each subexpression of a program might evaluate to at runtime [22]. The approximation computed for each expression is a set of abstract values that can be displayed on demand. The debugger can also draw arrows showing the flow of values through the program.

Figure 1 displays an example of analyzing a simple program. In the box next to the term `3` is the abstract value for that term, meaning that at runtime the term `3` might evaluate to the value `3`. The arrow starting from the term `3` shows that at runtime the value `3` might flow into the argument `x` of the function `f` and from there flow into the reference to the variable `x` in the body of `f`. There is a second reference to `x` in `f`—the corresponding arrow is not shown in this example. In the box next to the call to the Scheme primitive `gcd` is the abstract value for the result of that call. Since the analysis never tries to evaluate expressions, it uses the abstract value *integer* to represent the result of the primitive call, if any, which is a conservative approximation of the actual value that that call might compute at runtime.

The biggest box displays the type of the adjacent `if`-expression, which is the union of the *integer* abstract value computed by the `gcd` primitive and of the string `"hello"`. Arrows show that the result of the `if`-expression can come from both the then- and else-branches:

the analysis does not attempt to apply the `number?` predicate to the variable `x`, so it conservatively assumes that both branches of the `if`-expression may be evaluated at runtime.

3 Three Embedded Languages

We now turn to embedded languages, which are a useful technique for establishing abstraction layers for a particular design space. Functional languages are well-suited to writing interpreters for embedded languages, in which the higher-level embedded language is implemented as a set of functions in the general purpose host language and has access to all of its features [15, 16, 24]. But these abstractions come at a cost for program analysis. In particular, tools built to examine programs of the host language cannot derive information for the programs in the embedded languages because they do not understand the semantics of those languages.

In this section we demonstrate three examples of practical embedded languages for Scheme and show their negative effects on static analysis. In the first example, properties of the embedded language create the possibility of errors that can go undetected by the analysis. In the next two examples, undetected properties lead to analyses that are too conservative, resulting in many false positives; that is, the analysis reports errors that can never actually occur.

3.1 Format Strings

The PLT Scheme library provides a `format` function, similar to C’s `sprintf`, which generates a string given a format specifier and a variable number of additional arguments. The format specifier is a string containing some combination of literal text and formatting tags. These tags are interpreted along with the remaining arguments to construct a formatted string. The `format` function is thus an interpreter for the format specifier language. The format specifier is a program in this language and the additional arguments are its inputs.

To construct its output, the `format` function requires the number of extra arguments to match the number of format tags, and these arguments must be of the appropriate type. Consider the example of displaying an ASCII character and its encoding in hexadecimal:

```
(format "~c = 0x~x" c n)
```

In this example, the format specifier, which contains the format tags `"~c"` and `"~x"` and some literal text, expects to consume exactly two arguments. These arguments must be a character and an integer, respectively. An incorrect number of arguments or a type mismatch results in a runtime error.

Unfortunately analysis tools for Scheme such as MrFlow have no *a priori* knowledge of the semantics of embedded languages. The analysis cannot infer any information about the dependencies between the contents of the format string and the rest of the arguments without knowledge of the syntax and semantics of the `format` language. As a result the analysis cannot predict certain categories of runtime errors, as shown in Figure 2. The application of `format` is not underlined as an error, even though its arguments appear in the wrong order and the analysis correctly computes the types of both `c` and `n`.

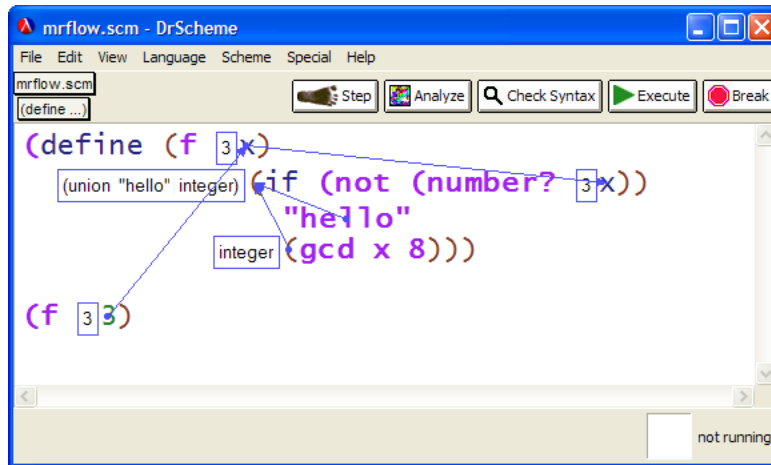


Figure 1. Analyzing a simple program with MrFlow.

3.2 Regular Expressions

Regular expressions are used in all kinds of Scheme programs. The language of regular expression patterns is embedded in Scheme as strings. A library of functions interpret these strings as programs that consume additional arguments as input strings and return either a list of matched subpatterns or #f to indicate failure.

Consider again the excerpt from the PLT Web Server from Section 1. Programmers know that if the match succeeds, then the result list contains exactly three elements: the result of the entire match, and the results of the two subpattern matches. Again the analysis is unable to discover this invariant on its own. Figure 3 shows the results of analyzing the sample code with MrFlow. The list accessors `second` and `third` are underlined in red because the analysis cannot prove that their arguments are sufficiently long lists.

Programmers then must either go through each of these false positives and prove for themselves that the errors can never occur, or else learn to ignore some results of MrFlow. Neither option is desirable. The former creates *more* work for the programmer, rather than less; the latter is unsafe and easily leads to overlooked errors.

3.3 SchemeQL

SchemeQL [28] is an embedded language for manipulating relational databases in Scheme. Unlike the string-based `format` language, SchemeQL programs consist of special forms directly embedded inside Scheme. The SchemeQL implementation provides a set of macros that recognize these forms and expand them into Scheme code. A typical database query in SchemeQL might look like this:

```
(direct-query (name age phone) directory)
```

corresponding to the SQL statement

```
SELECT name, age, phone FROM directory
```

The result of executing a query is a lazy stream representing a cursor over the result set from the database server. Each element in the stream is a list of values representing a single row of the result set. The cursor computes the rows by need when a program selects the next sub-stream.

Programmers know that the number of elements in each row of a cursor is equal to the number of columns in the original request. Our analysis, however, cannot discover this fact automatically. Figure 4 shows the results of an analysis of a SchemeQL query in the context of a trivial Scheme program. The example query consists of exactly three columns, and the code references the third element of the first row. This operation can never fail, but the analysis is unable to prove this. Instead, it conservatively computes that `row` is a list of unknown length: *rec-type* describes a recursive abstract value, which in the present case is the union of `null` and a pair consisting of any value (*top*) and the abstract value itself, creating a loop in the abstract value that simulates all possible list lengths. MrFlow therefore mistakenly reports an error by underlining the primitive `third` in red, since, according to the analysis, `row` might have fewer than three elements at runtime.

4 Macros for Partial Evaluation

All the embedded languages presented in the previous section have one thing in common: they can encode invariants that are not visible to any analysis of the general purpose language in which they are embedded. These invariants can be exposed to analyses in two ways:

- by extending the analyses in an ad-hoc manner for each embedded language so that they understand its semantics, or
- by partially evaluating the embedded interpreters with regard to the embedded programs to make the invariants in the embedded programs explicit as invariants in the host language, whenever possible.

The first solution requires modifying each analysis to support each embedded language. The second solution can simply be implemented from within the host language through the old Lisp trick of using “compiler macros” [25] as a light-weight partial evaluation mechanism. In the present case, instead of using partial evaluation to optimize programs for speed, we use it to increase the precision of program analyses.

While Lisp’s compiler macros are different from regular Lisp macros, Scheme’s macro system is powerful enough that the equivalent of Lisp’s compiler macros can be implemented as regular Scheme macros. The partial evaluation of embedded interpreters then simply involves replacing the libraries of functions imple-

```

format.scm
(define ...)

(let* ([c #\a]
      [n (char->integer c)])
  (format "~c=0x~x" exact-integer n #\a c))

```

Figure 2. Imprecise analysis of the format primitive.

```

url.scm
(define ...)

(define (download address)
  (let ([r (regexp-match
          "http://([a-z.]+)/([a-z]+)/" address)])
    (if r
        (process-url (third r)
                     (dispatch (second r)))
        (log-error "invalid url" address))))

```

Figure 3. Imprecise analysis of regexp-match.

```

schemeql.scm
(define ...)

(let* ([conn (connect-to-database "directory" "" "")]
      [result (direct-query conn
                            (name age phone)
                            directory)]
      [cursor (result-cursor result)]
      [(rec-type ((a0 (union (cons top a0) null))) a0) row (cursor-car cursor)])
  (disconnect-from-database conn)
  (third row))

```

Figure 4. Imprecise analysis of a SchemeQL query.

menting the interpreters with libraries of semantically equivalent macros¹. This has the additional advantage that it can be done by the author of the library of functions, as opposed to the compiler's or analyzer's implementor in the case of ad-hoc extensions.

Of course, the partial evaluation of embedded interpreters is only possible when their input programs are known statically. For example, it is not possible to expand a call to `format` if the formatting string given as its first argument is computed at runtime. The programmer therefore makes a trade-off between the precision of analyses and how dynamic the code can be. In practice, though, the embedded programs are often specified statically in user code. Combined with the simplicity of implementing partial evaluation with macros, this makes for a useful technique for improving the precision of analyses at a low cost.

In the next two sections, we describe some of the important features of the Scheme macro system and then explain how we make use of this system to partially evaluate the interpreters of these embedded languages to improve the results of static analysis.

5 Macros in Scheme

Scheme has a powerful macro system for extending the language with derived expression forms that can be rewritten as expressions in the core language. Macros serve as a means of syntactic abstraction. Programmers can generalize syntactic patterns in ways that are not possible with functional abstraction. This technology also provides a hook into the standard compiler tool chain by allowing programmers to implement additional program transformations before compilation. In this section we describe the basics of standard Scheme macros and introduce identifier macros, a generalization of the contexts in which macros can be matched.

5.1 Rule-Based Macros

The `define-syntax` special form allows the programmer to extend Scheme with derived expression forms. Before compilation or execution of a Scheme program, all occurrences of these derived forms are replaced with their specified expansions.

The `syntax-rules` form specifies macro expansions as rewrite rules. Consider the following simple macro, which defines a short-circuit logical `or` as a derived form:

```
(define-syntax or
  (syntax-rules ()
    [(or e1 e2)
     (let ([tmp e1])
       (if tmp tmp e2))]))
```

The macro defines a single rewrite rule, consisting of a *pattern* and a *template*. The pattern matches the `or` keyword in operator position followed by two pattern variables `e1` and `e2`, each matching an arbitrary subexpression in argument position. The template directs the macro expansion to replace occurrences of the matched pattern with a `let`-expression constructed from the matched subexpressions.

¹The transformation is not strictly speaking partial evaluation: the reductions performed by the macros are not exactly the ones performed by the embedded interpreters. However, the macros share the techniques and issues of partial evaluation since they simulate parts of the interpreters, and it is therefore useful to describe them as such.

Notice that this `or` form cannot be defined as a regular function in Scheme. The second argument is only evaluated if the first argument evaluates to false. Since Scheme has a strict evaluation semantics, a functional `or` would necessarily evaluate both of its arguments before computing a result. Controlling the evaluation of expressions is an important use of Scheme macros. Macros can also abstract over other syntactic forms in ways that functions cannot by expanding into second-class language constructs such as `define`.

5.2 Lexical Scope

Macros written with the standard Scheme `syntax-rules` mechanism are both *hygienic* and *referentially transparent*. Hygienic macro expansion guarantees that binding forms inside the definition of the macro template do not capture free variables in macro arguments. Consider the following use of our `or` macro:²

```
(or other tmp)
⇒ (let ([tmp1 other])
    (if tmp1 tmp1 tmp))
```

Hygienic expansion automatically renames the variable bound inside the expanded macro template to avoid capturing the free variable in the macro argument.

Referential transparency complements hygiene by ensuring that free variables inside the macro template cannot be captured by the context of the macro call site. For example, if the context that invokes `or` rebinds the `if` name, the expansion algorithm renames the binding in the *caller's* context to avoid capturing the variable used in the template body:

```
(let ([if 3])
  (or if #f))
⇒ (let ([if1 3])
    (let ([tmp if1])
      (if tmp tmp #f)))
```

The combination of hygiene and referential transparency produces macros that are consistent with Scheme's rules of lexical scope and can be invoked anywhere in a program without the danger of unexpected variable capture.³

5.3 Identifier Macros

The `syntax-rules` form only matches expressions in which the macro name occurs in "application position," i.e., as the operator in an application expression. References to a `syntax-rules` macro in other contexts result in syntax errors:

```
(fold or #f 1s)
⇒ syntax error
```

PLT Scheme's `syntax-id-rules` form is similar to `syntax-rules` but matches occurrences of the macro keyword in arbitrary expression contexts: in operator position, operand position, or as the target of an assignment.

²We use the convention of representing macro expansion with a double-arrow (\Rightarrow) and ordinary (runtime) evaluation with a single-arrow (\rightarrow).

³Macros can also be defined in and exported from modules in PLT Scheme [11].

The following macro demonstrates a hypothetical use of `syntax-id-rules`:

```
(define-syntax clock
  (syntax-id-rules (set!)
    [(set! clock e) (set-clock! e)]
    [(clock e) (make-time-stamp (get-clock) e)]
    [clock (get-clock)]))
```

The list of identifiers following `syntax-id-rules`, which was empty in our previous examples, now includes the `set!` identifier, indicating that `set!` is to be treated as a keyword rather than a pattern variable. The first rewrite rule matches expressions in which the `clock` name occurs as the target of an assignment. The second rule is familiar, matching the macro in application position. The final rule matches the identifier `clock` in any context not matched by the previous two rules. In addition to the usual application context, we can use the `clock` macro in an argument position:

```
(+ clock 10)
⇒ (+ (get-clock) 10)
```

or as a `set!` target:

```
(set! clock 5)
⇒ (set-clock! 5)
```

5.4 Programmatic Macros

The language of patterns and templates recognized by `syntax-rules` and `syntax-id-rules` is actually a special case of Scheme macros. In general, the `define-syntax` form binds a transformer procedure

```
(define-syntax name
  (lambda (stx)
    etc.)
```

The argument to the transformer procedure is a *syntax object*, which is similar to an S-expression representing quoted code, but which also encapsulates information about the lexical context of the code, such as source file location and variable bindings. This context information is essential in allowing DrScheme's language tools to trace errors and binding relationships back to the original source location in the user's code where a macro is invoked. Because syntax objects are so similar to quoted data, the standard library includes the `syntax-object->datum` procedure, which strips the lexical information from a syntax object and returns its corresponding datum. For example, the datum corresponding to a syntax object representing a literal number is its numeric value, the datum corresponding to an identifier is a symbol representing the identifier's name, and so on.

A syntax transformer procedure accepts as its argument a syntax object representing the expression that invoked the macro, and produces a new syntax object, which the macro expansion algorithm uses to replace the original expression. All Scheme macros are syntax transformers; although the `syntax-rules` and `syntax-id-rules` forms do not use the `lambda` notation, they are themselves implemented as macros that expand to syntax transformer procedures.

The `syntax-case` facility allows the construction of macros with pattern matching, as with `syntax-rules` and `syntax-id-rules`, but with arbitrary expressions in place of templates for the result expressions. For example, the above `or` macro would be defined as:

```
(define-syntax or
  (lambda (stx)
    (syntax-case stx ()
      [(or e1 e2)
       #'(let ([tmp e1])
           (if tmp tmp e2))]))))
```

The macro is almost the same as before, but for two refinements. First, the `syntax-case` form takes the argument `stx` explicitly, whereas `syntax-rules` implicitly defines a transformer procedure and operates on the procedure argument. Second, the result expression is prefixed by the syntax-quoting `#'` operator, which is analogous to Scheme's quote operator `'`. Whereas an expression prefixed with `'` evaluates to a quoted S-expression, a `#'` expression becomes a quoted syntax object that also includes lexical information. Similarly, the `quasisyntax` operator `#'` and `unsyntax` operator `#,` behave for syntax objects like the `quasiquote` and `unquote` operators for S-expressions, respectively.

The use of arbitrary computations in the result expression allows macros to expand differently based on the results of actual computations:

```
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(swap a b)
       (if (and (identifier? #'a)
                (identifier? #'b))
           #'(let ([tmp b])
               (set! b a)
               (set! a tmp))
           (raise-syntax-error
            'swap "expects identifiers"
            stx))]))))
```

In this example, if `swap` is not given identifiers as arguments, the `raise-syntax-error` function uses the lexical information in the `stx` syntax object to highlight the original `swap` expression in the user's code.

Conditional matching can also be achieved using *pattern guards*, which can inspect a matched expression and determine whether to accept a match:

```
(define-syntax swap
  (lambda (stx)
    (syntax-case stx ()
      [(swap a b)
       (and (identifier? #'a)
            (identifier? #'b))
       #'(let ([tmp b])
           (set! b a)
           (set! a tmp))]))))
```

The pattern guard is a new expression, inserted between the pattern and the result expressions. A guarded match only succeeds if its guard does not evaluate to false; when a guard fails, the pattern matcher falls through to attempt the next pattern in the list.

6 Macros for Interpreters

In this section, we present a general technique for specializing embedded interpreters with macros, and explain how we apply this technique to the three embedded languages described in Section 3.

The technique can be summarized in the following steps:

1. Write the interpreter compositionally as a module of library functions.
2. Replace the interpreter's main function with a macro that unfolds the case dispatch on the input (the embedded program) when it is known statically.
3. Default to the original function when the input is not known at compile time.

Writing the interpreters compositionally serves two purposes. First, by delegating the interpretation of the program constructs that make up an embedded program to separate functions, it becomes possible to share code between the original interpreter and the macro that replaces it. This effectively limits the macro's responsibility to a simple dispatch. Second, compositionality makes it easier to guarantee that unfolding terminates, since the recursive macro calls always operate on smaller terms.

6.1 Format Strings

The implementation of a string formatter involves a number of simple library functions to convert each possible type of argument to strings. Each formatting tag corresponds to one of these combinators. For example, the `"~c"` tag corresponds to a combinator, `format/char`, which accepts a character and converts it to a string, the `"~x"` tag corresponds to `format/hex`, which converts integers to their hexadecimal representation, and so forth. The string formatter then simply dispatches to these combinators based on the content of the formatting string:

```
(define (format s . args)
  (cond
    [(string=? s "") ""]
    [(string=? (substring s 0 2) "~c")
     (string-append (format/char (car args))
                    (apply format
                          (substring s 2)
                          (cdr args)))]
    etc. ))
```

The interpreter accepts the formatting string `s` and, based on formatting tags like `"~c"` that it finds, decomposes the string into a series of applications of the corresponding combinators to successive arguments of `format` (represented by `args`). It reassembles the transformed pieces with the standard `string-append` function.

In order to specialize the `format` interpreter, we replace it with a macro that re-uses its associated combinators:

```
(define (format/dynamic s . args)
  as before )
(define-syntax format
  (lambda (stx)
    (syntax-case stx ()
      [(format s-exp a1 a2 ...)
       (string? (syntax-object->datum #'s-exp))
       (let ([s (syntax-object->datum #'s-exp)])
         (cond
          [(string=? s "") #'""]
          [(string=? (substring s 0 2) "~c")
           #'(string-append
              (format/char a1)
              (format #,(substring s 2) a2 ...))]]
```

```
etc. )]]
[(format s-exp a1 a2 ...)
 #'(format/dynamic s-exp a1 a2 ...)]
[format
 (identifier? #'format)
 #'format/dynamic]]))
```

The partial evaluation works by unfolding the interpreter's top-level case dispatch on the program text. Rather than delaying the inspection of the string to runtime, the macro precomputes the result of the decomposition statically whenever the string is given as a literal. We can identify literal strings through the use of a pattern guard. More precisely, the macro can inspect the syntax object `s-exp`, corresponding to `format`'s first argument, and determine whether it can be converted to a string via `syntax-object->datum`. When the conversion succeeds, the pattern guard allows the match to succeed, and partial evaluation proceeds.

After the macro expansion, the resulting program text consists of the application of `string-append` to the calls to the library functions, with no references to the interpreter:

```
(format "~c = 0x~x" c n)
⇒ (string-append (format/char c)
                 " = 0x"
                 (format/hex n))
```

In order for the replacement of the original function with a macro to be unobservable, the macro must behave exactly like the original function in all contexts. When `format` is applied to a dynamic formatting string, the macro defaults to the original functional implementation. Similarly, when `format` is passed as an argument to a higher-order function, we use the technique of identifier macros to refer to the original function.⁴

6.2 Regular Expressions

One of PLT Scheme's regular expression engines uses the two-continuation model of backtracking [1]. A regular expression "matcher" is represented as a function that accepts a success continuation and a failure continuation. When a matcher succeeds in matching its input, it applies its success continuation to the accepted input, and when it fails to match, it invokes its failure continuation. This allows the interpretation of the alternation operator `"|"` to try each alternate pattern sequentially: an alternation matcher tries to match its first pattern with a failure continuation to try the second pattern. Thus if the first pattern fails, the matcher invokes the failure continuation, which tries the second pattern. Otherwise, the failure continuation is disregarded and the matcher applies its success continuation, which skips the second pattern and returns the result of the first match.

Each of the regular expression constructions corresponds to a functional combinator that produces a matcher. These combinators can express the standard operators of regular expressions: success, failure, alternation, concatenation, and repetition (i.e., Kleene star). There is also a `submatch` combinator for the parenthesized subpatterns in the original regular expression. A successful `regexp-match` returns a list with the entire matched string followed by each submatch corresponding to a parenthesized subpattern. Any subpattern that does not match corresponds to an entry of `false` (`#f`) in the result list. For example, the following successful

⁴The case of `set!` is not critical since, in PLT Scheme, imported module references cannot be the target of an assignment.

match contains a failed submatch:

```
(regexp-match "a((b)|(c))" "ac")  
→ (list "ac" "c" #f "c")
```

Regardless of the contents of the second argument, there is always exactly one element in the result list for each parenthesized sub-pattern in the regular expression. The `submatch` operator accomplishes this by wrapping a given matcher with continuations that add either the result of a successful match or false to a list of indexed submatches accumulated during the match. The initial (success) continuation for `regexp-match` sorts the accumulated list of indexed submatches, adding false entries for all submatches that were never reached because of backtracking.

Partial evaluation of the regular expression library works by unfolding the definitions of the combinators as well as the contents of the initial continuation. Each application of a combinator gets replaced by an application of a copy of the body of the combinator's definition.⁵ The recursive code that constructs the result list in the success continuation gets expanded into an explicit chain of `cons` expressions:

```
(regexp-match "a((b)|(c))" input)  
⇒ ((build-matcher input)  
   (lambda (subs)  
     (cons (lookup subs 0)  
           (cons (lookup subs 1)  
                 (cons (lookup subs 2)  
                       (cons (lookup subs 3) null))))))  
   (lambda () #f))
```

Since the size of the result list is known, it is possible to unfold recursive definitions, such as the initial continuation that constructs the match result, to make the structure of the result explicit.

Finally, in the cases where the embedded program is not known statically, or when `regexp-match` is used in non-application contexts, the macro expands to the original functional definition.

6.3 SchemeQL

The SchemeQL language differs from the other examples in that its programs are not embedded as strings but rather as special forms recognized by a library of macros. This means that for queries that select from a fixed set of columns, the length of cursor rows is always known statically; the column names are specified as a sequence of identifiers in the syntax of the query form.

Just as the interpreters for the string-based embedded programs perform a case dispatch on the contents of program strings, the SchemeQL macros dispatch on the shape of the query expressions. The cases where partial evaluation is possible can be captured by inserting additional rules into the original library's macros.

Partial evaluation of SchemeQL queries uses the same technique as for the regular expression library: the recursive function that constructs a cursor row is unfolded into an explicit chain of `cons` expressions. Since we know the length of the cursor row statically, the unfolding is guaranteed to terminate.

⁵It is convenient to define the Kleene star operator recursively by $p^* = (pp^*)\epsilon$. However, this non-compositional definition leads to an infinite macro expansion, so the macro must carefully avoid unfolding such a definition.

Since the SchemeQL library is implemented as macros, there is no need to capture the cases where the query forms are used in non-application contexts. Adding special cases to the existing macro does not affect its set of allowable contexts. Similarly, the cases where the row length is not known statically are already handled by the existing SchemeQL macros.

7 Static Analysis for Scheme

MrFlow's value flow analysis is an extension of an ordinary set-based closure analysis like Palsberg's [22]. For every expression in a program, MrFlow statically computes a conservative approximation of the set of values to which the expression might evaluate at runtime. From a given expression it creates a graph that simulates the flow of values inside the expression. The analysis simulates evaluation by propagating abstract values in this graph until reaching a fixed point. From the set of abstract values that propagate to a given node, the analysis reconstructs a type that is then displayed to the user through DrScheme's graphical interface.

Extensions to the basic analysis include, among other things: analyzing functions that can take any number of arguments, analyzing assignments to variables (`set!`), and analyzing generative data structure definitions. MrFlow also supports all the primitives defined in R⁵RS [17]. The vast majority of these primitives are defined using a special, type-like language embedded inside the analyzer. For a given primitive, the corresponding type translates to a graph that simulates the primitive's internal flows. The analysis then proceeds just like for any other expression. The few remaining primitives need special handling because of their imperative nature (`set-car!` or `vector-fill!`) and are analyzed in an ad-hoc manner.

By default, MrFlow analyzes the `format` primitive based on the following pseudo-type description:

```
(string top *-> string)
```

The `*` in the `*->` constructor means that the primitive is a function that can take any number of arguments as input beyond the ones explicitly specified. In the present case, the function must receive a string as its first argument, followed by any number of arguments of any type (represented by the pseudo-type `top`), and returns a string. Given such a description, the only errors MrFlow detects are when the primitive is given something other than a string as first argument, or if it is given no argument at all.

After partial evaluation, the application of `format` is replaced by calls to its individual library functions such as `format/char` and `format/hex`. These functions have respectively the pseudo-types

```
(char -> string)
```

and

```
(integer -> string)
```

Using this more precise information, MrFlow can detect arguments to the original `format` call that have the wrong type. Checking that the `format` primitive receives the right number of arguments for a given formatting string happens during partial evaluation, so the analyzer never sees arity errors in the expanded code.

Since DrScheme's syntax object system keeps track of program terms through the macro expansions [11], MrFlow is then able to trace detected errors back to the original guilty terms in the user's

program and flag them graphically. Arrows representing the flow of values can also be displayed interactively in terms of the original program, allowing the user to track in the program the sources of the values that triggered the errors. In essence, the only requirement for MrFlow to analyze the partially evaluated code of `format` is to specify the pseudo-types for the library functions introduced by the transformations, like `format/char`⁶.

Similarly, it is enough to define pseudo-types for the functions used in the partially evaluated form of SchemeQL's `query` to have MrFlow automatically compute precise results without any further modifications.

The partial evaluation for regular expressions is more challenging. Consider the example from Section 1:

```
(let ([r (regexp-match
         "http://([a-z.]*)/([a-z]*)/" line)])
  (if r
      (process-url (third r) (dispatch (second r)))
      (log-error)))
```

After the call to `regexp-match`, the variable `r` can be either a list of three elements or `false`. Based on its conservative pseudo-type specification for `regexp-match`, MrFlow computes that `r` can be either a list of unknown length or `false`. This in turn triggers two errors for each of the `second` and `third` primitives: one error because the primitive might be applied to `false` when it expected a list, and one error because it might be applied to a list that is too short.

The second kind of false positives can be removed by partially evaluating `regexp-match` to make the structure of the result more explicit to MrFlow, as described in Section 6.2. The analysis then determines that the primitive returns either a list of three elements or `false` and in turn checks that `second` and `third` are applied to a list with enough elements.

Still, the possible return values of `regexp-match` may contain `false`. Indeed, `false` will be the value returned at runtime if the line given to `regexp-match` does not match the pattern. The programmer has to test for such a condition explicitly before processing the result any further. The only way for MrFlow not to show a false positive for `second` and `third`, because of the presence of this false value, is to make the analysis aware of the dependency between the test of `r` and the two branches of the `if`-expression. This form of flow-sensitive analysis for `if`-expressions is difficult to implement in general since there is no bound to the complexity of the tested expression. In practice, however, an appreciable proportion of these tests are simple enough that an ad-hoc solution is sufficient.

In the case where the test is simply a variable reference it is enough to create two corresponding ghost variables, one for each branch of the `if`, establish *filtering flows* between the variable `r` and the two ghost variables, and make sure each ghost variable binds the `r` variable references in its respective branch of the `if`-expression. The filtering flows prevent the false abstract value from flowing into the then branch of the `if`-expression and prevent everything but the false value from flowing into the else branch. Only the combination of this flow sensitivity for `if`-expressions with the partial evaluation of `regexp-match` gives analysis results with no false positives.

⁶Specifying such pseudo-types will not even be necessary once MrFlow knows how to analyze PLT Scheme contracts. This is the subject of a forthcoming paper.

Once flow-sensitive analysis of `if`-expressions is added and pseudo-type descriptions of the necessary primitives are provided to the analysis, partial evaluation makes all the false positives described in Section 3 disappear, as we illustrate in the next section.

8 Improvement of Static Analysis

Partially evaluating `format` eliminates the possibility of runtime arity errors, since the macro transformations can statically check such invariants. It also allows MrFlow to detect type errors that it could not detect before, since the corresponding invariants were described only in the embedded formatting language. These invariants are now explicit at the Scheme level in the transformed program through the use of simpler primitives like `format/char` or `format/integer`. Figure 5 shows the same program as in Figure 2, but after applying partial evaluation. The `format` primitive is now blamed for two type errors that before could be found only at runtime. The error messages show that the user simply gave the arguments `n` and `c` in the wrong order.

Similarly, specializing the regular expression engine with respect to a pattern eliminates false positives. The length of the list returned by `regexp-match` cannot be directly computed by the analysis since that information is hidden inside the regular expression pattern. As a result, the applications of `second` and `third` in Figure 3 are flagged as potential runtime errors (we have omitted the fairly large error messages from the figure). After specialization, the structure of the value returned by `regexp-match` is exposed to the analysis and MrFlow can then prove that if `regexp-match` returns a list, it must contain three elements. The false positives for `second` and `third` disappear in Figure 6.

Of course, `regexp-match` can also return `false` at runtime, and the analysis correctly predicts this regardless of whether partial evaluation is used or not. Adding flow sensitivity for `if`-expressions as described in Section 7 removes these last spurious errors in Figure 6.

Partial evaluation now allows the precise analysis of SchemeQL queries as well. Figure 7 shows the precise analysis of the same program as in Figure 4, this time after partial evaluation. As with `regexp-match`, the analysis previously computed that `cursor-car` could return a list of any length, and therefore flagged the call to `third` as a potential runtime error. This call is now free of spurious errors since the partial evaluation exposes enough structure of the list returned by `cursor-car` that MrFlow can compute its exact length and verify that `third` cannot fail at runtime.

While the results computed by the analysis become more precise, partially evaluating the interpreters for any of the three embedded languages we use in this paper results in code that is bigger than the original program. Bigger code in turn means that analyses will take more time to complete. There is therefore a trade-off between precision and efficiency of the analyses. We intend to turn that trade-off into a user option in MrFlow. The user might also exercise full control over which embedded languages are partially evaluated and where by using either the functional or macro versions of the embedded languages' interpreters, switching between the two through the judicious use of a module system, for example [11].

Note that partial evaluation does not always benefit all analyses. In the `regexp-match` example from Figure 6, spurious errors disappear because MrFlow has been able to prove that the list `r` is of length three and therefore that applying the primitives `second` or

The screenshot shows the DrScheme IDE with the file 'format.scm' open. The code defines a function that takes a character 'c' and an integer 'n', and returns a string formatted as '~c=0x~x'. The analysis shows that the character 'c' is of type '#\a' and the integer 'n' is of type 'integer'. The 'format' primitive is analyzed as taking a string and a character, and returning a string.

```

(format.scm)
(define ...)

(let* ([c #\a]
       [n (char->integer c)])
  (format "~c=0x~x" n c))

```

not running

Figure 5. Precise analysis of the `format` primitive.

The screenshot shows the DrScheme IDE with the file 'url.scm' open. The code defines a function 'download address' that uses 'regexp-match' to parse a URL. The analysis shows that the 'regexp-match' primitive is analyzed as taking a string and a regular expression, and returning a list of strings. The 'process-url' primitive is analyzed as taking a string and returning a string.

```

(url.scm)
(define ...)

(define (download address)
  (let ([r (regexp-match
            "http://([a-z.]*)/([a-z]*)/" address)])
    (if (union (cons (union #f string) (cons (union #f string) (cons (union #f string) null))) #f) r
        (process-url (union #f string) (third r))
          (dispatch (second r)))
      (log-error "invalid url" address))))

```

not running

Figure 6. Precise analysis of `regexp-match`.

The screenshot shows the DrScheme IDE with the file 'schemeql.scm' open. The code defines a function that connects to a database, queries for a directory, and returns the first row of the result. The analysis shows that the 'connect-to-database' primitive is analyzed as taking a string and returning a connection object. The 'direct-query' primitive is analyzed as taking a connection object and a list of strings, and returning a result object. The 'result-cursor' primitive is analyzed as taking a result object and returning a cursor object. The 'cursor-car' primitive is analyzed as taking a cursor object and returning a row. The 'disconnect-from-database' primitive is analyzed as taking a connection object and returning nothing.

```

(schemeql.scm)
(define ...)

(let* ([conn (connect-to-database "directory" "" "")]
       [result (direct-query conn
                             (name age phone)
                             directory)]
       [cursor (result-cursor result)]
       [row (cons top (cons top (cons top null)))])
  (disconnect-from-database conn)
  (third row))

```

not running

Figure 7. Precise analysis of a SchemeQL query.

third to `r` cannot fail. If the analysis were a Hindley-Milner-like type system, though, no difference would be seen whether partial evaluation were used or not. Indeed, while such a type system could statically prove that the arguments given to `second` or `third` are lists, it would not attempt to prove that they are lists of the required length and a runtime test would still be required. Using partial evaluation to expose such a property to the analysis would therefore be useless. Simply put, making invariants from embedded programs explicit in the host language only matters if the system analyzing the host language cares about those invariants.

This does not mean partial evaluation is always useless when used in conjunction with a Hindley-Milner type system, though. Partially evaluating `format`, for example, would allow the type system to verify that the formatting string agrees with the types of the remaining arguments. This is in contrast to the ad-hoc solution used in OCaml [19] to type check the `printf` primitive, or the use of dependent types in the case of Cayenne [4].

9 Related Work

Our work is analogous to designing type-safe embedded languages such as the one for `printf` [21, 4]. Both problems involve determining static information about programs based on the values of embedded programs. In some cases, designers of typed languages simply extend the host language to include specific embedded languages. The OCaml language, for example, contains a special library for `printf` [19] and uses of `printf` are type-checked in an ad-hoc manner. Similarly, the GCC compiler for the C language uses ad-hoc checking to find errors in `printf` format strings. Danvy [7] and Hinze [14] suggest implementations of `printf` in ML and Haskell, respectively, that obviate the need for dependent types by recasting the library in terms of individual combinators. In our system, those individual combinators are automatically introduced during macro expansion. The C++ language [26] likewise avoids the problem of checking invariants for `printf` by breaking its functionality into smaller operations that do not require the use of an embedded formatting language.

A work more closely related to ours is the Cayenne language [4]. Augustsson uses a form of partial evaluation to specialize dependent types into regular Haskell-like types that can then be used by the type system to check the user’s program. Our macro system uses macro-expansion time computation to specialize expressions so that the subsequent flow analysis can compute precise value flow results. Augustsson’s dependent type system uses computation performed at type-checking time to specialize dependent types so that the rest of the type checking can compute precise type information. The specialization is done in his system through the use of type-computing functions that are specified by the user and evaluated by the type system.

The main difference is that his system is used to compute specialized types and verify that the program is safe. Once the original program has been typed it is just compiled as-is with type checking turned off. This means that in the case of `format`, for example, the formatting string is processed twice: once at type checking time to prove the safety of the program, and once again at run time to compute the actual result. Our system is used to compute specialized expressions. This means that the evaluation of the `format`’s string needs to be done only once. Once specialized, the same program can either be run or analyzed to prove its safety. In both cases the format string will not have to be reprocessed since it has been completely replaced by more specialized code.

Another difference is that in our system, non-specialized programs are still valid programs that can be analyzed, proved safe, and run (though the result of the analysis will probably be more conservative than when analyzing the corresponding partially evaluated program, so proving safety might be more difficult). This is not possible in Cayenne since programs with dependent types cannot be run without going through the partial evaluation phase first.

Much work has gone into optimization of embedded languages. Hudak [15], Elliott *et al* [8], Backhouse [5], Christensen [6], and Veldhuizen [27] all discuss the use of partial evaluation to improve the efficiency of embedded languages, although none makes the connection between partial evaluation and static analysis. In Backhouse’s thesis he discusses the need to improve error checking for embedded languages, but he erroneously concludes that “*syntactic* analyses cannot be used due to the embedded nature of domain-specific embedded languages.”

The Lisp programming language ([25], Section 8.4) provides for “compiler macros” that programmers can use to create optimized versions of existing functions. The compiler is not required to use them, though. To our knowledge, there is no literature showing how to use these compiler macros to improve the results of static analyses. Lisp also has support for inlining functions, which might help monovariant analyses by duplicating the code of a function at all its call sites, thereby simulating polyvariant analyses.

Bigloo [23] is a Scheme compiler that routinely implements embedded languages via macros and thus probably provides some of the benefits presented in this paper to the compiler’s internal analyses. The compiler has a switch to “enable optimization by macro expansion,” though there does not seem to be any documentation or literature describing the exact effect of using that switch.

10 Conclusion

Programs in embedded languages contain invariants that are not automatically enforced by their host language. We have shown that using macros to partially evaluate interpreters of little languages embedded in Scheme with respect to their input programs can recapture these invariants and convey them to a flow analysis. Because it is based on macros, this technique does not require any ad-hoc modification of either interpreters or analyses and is thus readily available to programmers. This makes it a sweet spot in the programming complexity versus precision landscape of program analysis. We intend to investigate the relationship between macros and other program analyses in a similar manner.

Acknowledgments

We thank Matthias Felleisen, Mitchell Wand, and Kenichi Asai for the discussions that led to this work and for their helpful feedback. Thanks to Matthew Flatt for his help with the presentation of Scheme macros. Thanks to Dale Vaillancourt for proofreading the paper and to Ryan Culpepper for his macrological wizardry.

11 References

- [1] H. Abelson and G. J. Sussman. *The Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.
- [2] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35:79–111, 1999.

- [3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3d edition, 2000.
- [4] L. Augustsson. Cayenne—a language with dependent types. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250. ACM Press, 1998.
- [5] K. Backhouse. *Abstract Interpretation of Domain-Specific Embedded Languages*. PhD thesis, Oxford University, 2002.
- [6] N. H. Christensen. *Domain-specific languages in software development – and the relation to partial evaluation*. PhD thesis, DIKU, Dept. of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark, July 2003.
- [7] O. Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.
- [8] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. In *SAIG*, pages 9–27, 2000.
- [9] R. B. Findler, J. Clements, M. F. Cormac Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.
- [10] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. on Programming Languages and Systems*, 21(2):369–415, Feb. 1999.
- [11] M. Flatt. Composable and compilable macros: you want it when? In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 72–83. ACM Press, 2002.
- [12] P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen. Programming the web with high-level programming languages. In *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001, Proceedings*, volume 2028 of *Lecture Notes in Computer Science*, pages 122–136, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.
- [13] N. Heintze. *Set Based Program Analysis*. PhD thesis, Carnegie-Mellon Univ., Pittsburgh, PA, Oct. 1992.
- [14] R. Hinze. Formatting: a class act. *Journal of Functional Programming*, 13(5):935–944, 2003.
- [15] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.
- [16] S. N. Kamin. Research on domain-specific embedded languages and program generators. In R. Cleaveland, M. Mislove, and P. Mulry, editors, *Electronic Notes in Theoretical Computer Science*, volume 14. Elsevier, 2000.
- [17] R. Kelsey, W. Clinger, and J. R. [editors]. Revised⁵ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–104, August 1998. Also appeared in *SIGPLAN Notices* 33:9, September 1998.
- [18] B. W. Kernighan and D. M. Ritchie. *The C programming language*. Prentice Hall Press, 1988.
- [19] X. Leroy. The Objective Caml System, release 3.07, 2003. <http://caml.inria.fr/ocaml/htmlman>.
- [20] P. Meunier. <http://www.plt-scheme.org/software/mrflow>.
- [21] M. Neubauer, P. Thiemann, M. Gasbichler, and M. Sperber. Functional logic overloading. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 233–244. ACM Press, 2002.
- [22] J. Palsberg. Closure analysis in constraint form. *Proc. ACM Trans. on Programming Languages and Systems*, 17(1):47–62, Jan. 1995.
- [23] M. Serrano and P. Weis. Bigloo: A portable and optimizing compiler for strict functional languages. In *Static Analysis Symposium*, pages 366–381, 1995.
- [24] O. Shivers. A universal scripting framework, or Lambda: the ultimate “little language”. In *Proceedings of the Second Asian Computing Science Conference on Concurrency and Parallelism, Programming, Networking, and Security*, pages 254–265. Springer-Verlag, 1996.
- [25] G. L. Steele. *COMMON LISP: the language*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1984. With contributions by Scott E. Fahlman and Richard P. Gabriel and David A. Moon and Daniel L. Weinreb.
- [26] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., 1997.
- [27] T. L. Veldhuizen. C++ templates as partial evaluation. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 13–18, 1999.
- [28] N. Welsh, F. Solsona, and I. Glover. SchemeUnit and SchemeQL: Two little languages. In *Proceedings of the Third Workshop on Scheme and Functional Programming*, 2002.