Obstacles to Compilation of Rebol Programs

Viktor Pavlu

TU Wien
Institute of Computer Aided Automation, Computer Vision Lab
A-1040 Vienna, Favoritenstr. 9/183-2, Austria
pavlu@caa.tuwien.ac.at

Abstract. Rebol's syntax has no explicit delimiters around function arguments; all values in Rebol are first-class; Rebol uses *fexprs* as means of dynamic syntactic abstraction; Rebol thus combines the advantages of syntactic abstraction and a common language concept for both meta-program and object-program. All of the above are convenient attributes from a programmer's point of view, yet at the same time pose severe challenges when striving to compile Rebol into reasonable code. An approach to compiling Rebol code that is still in its infancy is sketched, expected outcomes are presented.

Keywords: first-class macros, dynamic syntactic abstraction, $vau calculus, fexpr, Kernel, Rebol

1 Introduction

A meta-program is a program that can analyze (read), transform (read/write), or generate (write) another program, called the object-program.

Static techniques for syntactic abstraction (macro systems, preprocessors) resolve all abstraction during compilation, so the expansion of abstractions incurs no cost at run-time. Static techniques are, however, conceptionally burdensome as they lead to staged systems with phases that are isolated from each other. In systems where different syntax is used for the phases (e. g., C++), it results in a multitude of programming languages following different sets of rules.

In systems where the same syntax is shared between phases (e. g., Lisp), the separation is even more unnatural: two syntactically largely identical-looking pieces of code cannot interact with each other as they are assigned to different execution stages.

While static approaches to syntactic abstraction try to alleviate the burden on the programmer by lowering the barriers between phases (e. g., `constexprs` introduced in C++11 can be used at compile-time and in the compiled program) we are interested in making conceptually simple dynamic abstractions more efficient.

Recent work [7] has shown that fexprs can in fact be used to bring the two phases together in a single dynamic syntactic abstraction system.

Vau expressions as defined by Shutt create operative combiners (operatives) based on statically scoped fexprs. An operative combiner is a function that does not evaluate its operands before application. Instead, operatives work directly with their operands,

not on their argument values. Operands are passed unevaluated together with access to the calling environment, so that the operands may be evaluated explicitly when needed by the operative. This shifts from an implicit-evaluation environment to an environment where the evaluation of operands is controlled explicitly.

Operative combiners have access to both the static (or lexical) environment where the combiner was created and the dynamic environment where the combiner is applied. When using the dynamic environment to evaluate the body of the combiner, we get dynamic scoping as in LISP. When using the static environment to evaluate the body of the combiner, we get static scoping as in Scheme. Both can be implemented using Shutt's operative combiners as the vau abstraction provides explicit access to both environments.

The shift from an implicit-evaluation environment to an environment where evaluation of operands is explicitly requested avoids frequent difficulties with (naive) macro implementations. Consider the running example in Figs. 1–4. `or` shall be defined with short-circuit evaluation, so that it returns the value of its first operand if it evaluates to true and otherwise returns the result of evaluating the second operand.

A naive approach to define `or` as applicative (or lambda) is shown in Fig. 1. The problem with this definition is that the second operand will be evaluated already when `or` is applied, no matter what the value of the first operand is. The definition therefore lacks the required short-circuit evaluation.

```
>> (define (or x y)
     (if x
         x
         y))

>> (or 1 something-undefined)
reference to undefined identifier: something-undefined
```

Fig. 1. Flawed defintion of `or` as an applicative in Racket. Operands to an applicative are evaluated to argument values during application, so the required short-circuiting cannot be provided by applicatives.

Using pattern-based macros we can define `or` as a macro that rewrites to a conditional (cf. Fig. 2) and offers the required short-circuiting but the first operand is evaluated twice in the expanded code which in the least creates bloat but also causes unwanted results when the evaluation has side-effects.

By the use of a local variable to cache the value of the first operand, the macro can avoid duplicate evaluation (cf. Fig. 3). The introduction of a new local variable, however, introduces another set of problems with accidental name captures that hygienic macro implementations solve for the programmer [1]. Still, the macro definition is obfuscated with code that circumvents multiple evaluation as there is no means of referring to a parameter without triggering its evaluation in an implicit-evaluation environment.

In the context of explicit-evaluation operatives this is different: referring to a parameter and evaluation of a parameter are distinct. Shutt's Kernel programming language [6]

```
>> (define-syntax-rule (or x-exp y-exp)
     (if x-exp x-exp y-exp))

>> (or (print "first") (print "second"))
"first""first"
```

Fig. 2. Flawed defintion of a short-circuiting `or` as a pattern-based macro in Racket. The first operand is evaluated twice.

```
>> (define-syntax-rule (or x-exp y-exp)
     (let ([x x-exp])
       (if x x y-exp)))

>> (or ((get-print) "first") ((get-print) "second"))

>> (or (print "first") (print "second"))
```

Fig. 3. Definition of a short-circuiting `or` as a pattern-based macro in Racket. Caching the result of the evaluated first operand avoids multiple evaluation.

based on vau expressions thus allows for a straight-forward definition of the `or` macro as shown in Fig. 4. No code to circumvent multiple evaluation obfuscates the actual algorithm. The calls to `eval` make it explicit where evaluation is performed and within which environment.

```
(define-vau (or x-exp y-exp env) env
  (let ([x (eval x-exp env)])
    (if x x (eval y-exp env))))
```

Fig. 4. Definition of a short-circuiting `or` operative in Kernel (with Racket Syntax).

Operatives can choose to implement any evaluation strategy and may even choose not to evaluate but analyze its operands. An operative may then compose code based on the syntactic structure of its operands, which is what macro systems offer, only at run-time. As a result, the core language is drastically simplified, as many language features usually built-in to the language can be constructed from vau expressions, e. g., Macros, special forms, applicative and operative combiners can all be constructed from vau expressions. Applicative combiners or lambdas, i. e., functions that evaluate operands to arguments, are just operative combiners where all operands are evaluated to argument values.

Shutt has demonstrated that a language based on vau expressions can use a small axiomatic set of primitives for both the macro language and the target language.

Dynamic syntactic abstraction using vau expressions thus promises both, the general advantages of syntactic abstraction in e. g., crafting domain-specific (sub-)languages and

a common language concept. The phase separation is overcome as syntactic abstraction is then an equal among the other abstraction mechanisms, increasing the use of syntactic abstraction and making it possible for interactive languages to have syntactic abstraction in the first place.

2 Problem

When operatives are primitive data types of the language (first-class operatives) the separation in a metaprogramming- and a programming stage is lifted from the language. What would otherwise be conceptually isolated may now freely interact, i. e., not only may the same language and primitives be used in the meta-program and the object program (homogenous metaprogramming), they may also share the same data as the phase separation is overcome.

This unique feature of dynamic syntactic abstraction, however, comes at a price: when operatives are deliberately indistinguisable from applicatives, it is generally no longer possible to determine whether a combiner is an applicative (works on operands evaluated to arguments) or an operative (works on unevaluated operands), until immediately before evaluation (when the operator of a combination is evaluated to an applicative or operative combiner). This is not an unwanted side-effect but the direct result of treating operative combiners like any other value in the language. Still, it presents a practical difficulty.

Expressions in a program with first-class operatives can no longer be grouped into expressions that will be evaluated to argument values and expressions that must remain unmodified. Hence, two expressions that evaluate to the same value are no longer interchangeable in any context because an operative combiner may distinguish between those expressions on purely syntactic grounds, e. g., `(print (add 3 1))` must not be replaced with the shorter `(print 4)` since `print` may or may not be an operative that distinguishes between combinations and literals.

In effect, all expressions must remain unmodified in order to not alter the meaning of the program as long as it is not known whether an expression is to be evaluated by an applicative or operative combiner. This precludes all program optimization since there is no way to statically distinguish between applicative and operative combiners in the general case.

2.1 The Kernel Programming Language

Kernel [6] is the programming language implemented by Shutt to demonstrate the practicability of fexpr-based dynamic syntactic abstractions.

Even without the transformation-adverse properties introduced by first-class operatives, Kernel is not an easy target for program analysis:

- We cannot statically determine the value of any variable; all combiners are first-class and kept in variables.
- We cannot statically compute the value of even the simplest arithmetic expressions (e. g., $1 + 1$) because mathematical operators are also combiners and may have been redefined to a non-standard binding.

- Non-standard bindings of combiners may also alter the number of operands the combiner uses, resulting in a different program structure.
- We cannot statically determine the list of free variables in a code block.

Kernel is thus implemented as a simple non-optimizing interpreter due to the properties of the language.

2.2 The Rebol Programming Language

Rebol [4] has the same concept of operands being passed unevaluated but calls them *get arguments* and the choice of evaluation or non-evaluation lies with each individual operand rather than with the type of combiner (applicative or operative). So while Kernel only allows purely applicative or purely operative combiners, Rebol programs may have mixed combiners where some values are passed as unevaluated operands and some as evaluated arguments.

Further, Rebol has no syntax to denote the list of operands in a combination but instead each combiner has a fixed number of operands known from the combiner's definition. Before applying a function, the exact number of operand expressions is evaluated and then passed as argument values.

With the potential for redefinitions to a different number of operands this is an obstacle for static analysis in its own right, that is orthogonal to the problem of discerning applicative and operative operands.

The example in Fig. 5 illustrates the practical implications of this. It shows how a programmer would interpret the application of three pre-defined functions in Rebol. Given this example, a Rebol programmer knows that `replace` consumes the three values and replaces all occurrences of "bar" with "baz" in the string "foo". The call to `append` returns a new string "foobar" and is followed by the (unconsumed) literal "baz", and the third function application using `print` is read as `(print "foo")` followed by two string literals. With the information on the number of arguments a function consumes, a program analysis would be able to interpret this program fragment in the same manner. If, however, the number of arguments a function consumes is not known statically, as is the case in the example given in Fig. 5 where the arity of function `f` depends on a value only known at runtime, neither programmer nor analysis are able to statically discern between operand values and further values that follow the function application.

While making the definition of a function's arity depend on some completely random value is clearly an artificial example, it is definitely of practical relevance to a language as dynamic as Rebol that the definition of a function depends on a value only known at runtime. This is always the case when an identifier is used to abstract over two or more implementations of a function, e. g., when a generic `open-db` function is used to establish the connection to a database, and depending on the particular database system used, one or the other `open-db`-implementation is assigned to `open-db`. Then, at least, the case of a change in arity is ruled out for practical reasons, although in theory is still possible.

The absence of syntax to delimit the operand list has an additional effect in Rebol. Applicatives evaluate their arbitrarily nested operand expressions to single argument values while operatives only consume their first operand unevaluated as single value

```
>> replace "foo" "bar" "baz"
;; read as: (replace "foo" "bar" "baz")

>> append "foo" "bar" "baz"
;; read as: (append "foo" "bar") ("baz")

>> print "foo" "bar" "baz"
;; read as: (print "foo") ("bar") ("baz")

>> switch random 3 [
     1 [f: :replace]
     2 [f: :append ]
     3 [f: :probe  ]
   ]

>> f "foo" "bar" "baz"
;; read as: ???
```

Fig. 5. Function application syntax in Rebol has no clue to the number of arguments a funtion will consume. Without the implicit information on the arity of functions, it is impossible to parse the tokens following the application of `f` into arguments and non-arguments.

with no attention to the arity of sub-expressions. The same code `f g x` where `f` and `g` are combiners with arity 1 can therefore result in different parse trees depending on the kind of combiner in `f`: with an applictive combiner, the code reads as `(f (g x))`, whereas an operative combiner results in `(f g) (x)`.

An example is given in Fig. 6. In the applicative combiner, the operand expression `add1 1` is reduced to 2 and passed as argument value. `42` remains as the next value to be consumed. Meanwhile, in the operative example, the operand expression consists of `add1` only and `1` remains as the next value to be consumed.

So Rebol, too, has very limited room for static optimization and is implemented as a simple interpreter.

3 Expected Results

By introducing a strict import/export module system to a language with dynamic syntactic abstraction, we essentially replace the implicit stage boundaries common to static syntactic abstraction techniques with explicit module boundaries. We expect three main results of this change.

3.1 Increased Flexibility Inside

We believe that the creation of domain-specific languages is a very powerful tool that is best used within a closely confined part of a program. Within a module there will be no separation between meta- and object program and no phase separation when using

```
>> add1: func [x][ x + 1 ]

>> applicative: func [x][ probe x ]
>> operative: func  ['x][ probe x ]

>> print-next: func [a b][
     print ["next value:" b]
   ]

>> print-next applicative add1 1 42
2
next-value: 42

>> print-next operative   add1 1 42
add1
next-value: 1
```

Fig. 6. Two combiners are evaluated. The applicative reduces a nested expression to a single value while the operative passes a single operand without reduction, resulting in different associations of operands.

first-class operatives. A single base language can be used to both extend the language and write programs using the base or extended languages.

Between modules, the flexibility is curtailed. Any modifications to the base language are restricted to module boundaries. We believe this to be the level of flexibility that is practical when using such a powerful tool. For small problems, domain-specific sub-languages have successfully been used as effective tools. For programming in the large, however, we do not believe that it is of advantage if all combiners, operators and functions as well as names can be redefined at a single place and all other, entirely unrelated places that use the same entity, are affected.

3.2 Modularization of Modifications

While stage boundaries prevent program phases from all interactions with each other (even wanted interactions of seemingly compatible parts), the module boundaries

- separate different program parts and prevent *accidental* interaction of distant program parts with each other,
- explicitly state the interface between separate program parts,
- document the scope of semantic abstractions,
- aid with selective imports of syntactic abstractions and thus encourage reuse of abstractions.

This simplifies the creation of domain-specific languages as accidental side-effects of language modifications are better isolated. Errors due to these side-effects are then easier to locate and debug.

All of which fosters locality and modularization of language extensions, therefore increasing clarity of a program with syntactic abstraction and adding to hygiene. This will add to the attractiveness of domain-specific languages and dynamic syntactic abstraction.

3.3 Room for Static Analysis

Explicity stated imports/exports between modules confine the flexibility of the self-modifying language within module boundaries to a degree where static analysis and optimization becomes useful again.

When the flexibility that code from one module leaks into another module is ruled out and the only interaction between modules is documented through the imports, we anticipate that modules are self-contained to the degree that all analysis is essentially whole-program analysis, so static analysis becomes useful again. The interplay with other modules need not be analyzed at all, as information on external objects is already available through the imports each module is instructed to follow.

From the division into libraries we expect that separate ahead-of-time compilation to machine code is feasible and allows efficient implementation of syntactic abstractions without stage boundaries.

4 Methods

Aim of this work is to research practical methods to achieve separate compilation of subexpressions at the module level in a language with first-class operatives as dynamic syntactic abstraction.

We will primarily focus our efforts on the Kernel programming language as several problems with Rebol are avoided there while the metaprogramming characteristics that make Rebol interesting are retained. In particular, Kernel has a clean definition of the language (not just a reference implementation), several implementations are available and it does not share the implicit arity of combiners found in Rebol, so we can concentrate on the problem of separate compilation of first-class operatives.

The first steps are:

- We analyze library systems that allow the selective importation and exportation of symbols for their suitability to a language based on vau expressions. As starting point we use the Scheme library system defined in R7RS small language [5].
- Leveraging the customized library system we add devices to the language that separate programs into independent modules with an explicitly defined interface. We expect that tying down all flexibility between modules to what is explicity declared in the imports/exports will allow us to conservatively treat each module as a complete program, enabling whole-program analysis at the module level.
- A static data-flow analysis will be formulated as whole-program analysis to find stable bindings of operatives and applicatives. With this information in place, safe optimizing transformations can be implemented.
- For pathological programs that abuse the flexibility of the language (i. e., repeated redefinition of core operatives that confuses static analysis) the analysis will produce

warnings as we think that those cases are rarely of practical use. Compilation of programs with such constructs must be treated in a convservative way, however. Fallback to interpretation at run-time is an option. The rationale here is that programs are analyzed and optimized where possible and interpreted at run-time where necessary.
- Shutt comments briefly on possible inlining transformations for hygienic and unhygienic fexprs. This seems a viable starting point for optimizations orthogonal to module-based optimization and approximate typing.

The benefits of Shutt's vau expressions can be evaluated by direct comparison of programs with first-class operatives to a similar program written in a language without this language construct. A more rigorous evaluation of vau expressions is difficult as the number of real-world programs in Kernel is very small. The motivation for vau expressions is, however, not the main focus of this work. Our goal is to investige restrictions on programs with vau expression that do not impede their usefulness as a means of syntactic abstraction and, at the same time, create room for static analysis and optimization.

The fact that first-class operatives are not encumbered by our restrictions will be validated by creating an implementation of Kernel under these restrictions. In Kernel, the operatives are not merely a device added on top of the languge to enhance programmer productivity (which, in addition, would be laborious to evaluate), but are the basic abstraction of the language from which all other abstractions are built, i. e., `$lambda`, the primitive to define functions, is an application of the operative `$vau`. Consequently, if it is possible to implement Kernel with our restrictions in place, the viability of the restrictions is demonstrated.

The other part to be validated is that the restrictions actually introduce a potential for analysis and optimization and to a lesser degree, that the optimizations are in fact beneficial to the efficiency of programs. The feasibility of optimizing ahead-of-time compilation will be established by implementing a system capable of modular compilation of a language with first-class operatives. This is an open question left for future work in Shutt's thesis [7] and the central point of this work.

Once the potential for optimization is in place, it is then interesting to evaluate the quality of optimizations in terms of fast execution. This is primarily a question of quality regarding the analyses and transformations but also regarding the potential for optimizations attained through restrictions on the flexibility of the language, which, again, is the center of interest in our work. Evaluation of performance will be done using benchmarks derived from the (few) existing Kernel and (more) Rebol programs. Roughly 1200 programs are available through the Rebol Script Library [3] that can be used for this purpose.

5 Related Work

Macros are the predominant form to implement syntactic abstraction. They range from simple token-based substitutions over pattern-based substitution systems (syntax-rules macros used in R5RS [2]) to meta-programs that can use arbitrary functions to create their object programs (syntax-case macros used in R6RS [1]). They have in common

that they operate in an implicit-evaluation environment so that naively written macros may suffer from undesired multiple evaluation of macro operands. A way around this is to have the programmer manually ensure that macro parameters are at most evaluated once and the results be kept in a variable local to the macro expansion for use inside the expanded code. Hygienic macro expansion automatically takes care that local names of such variables do not inadvertedly capture bindings at the macro expansion site. Circumventing multiple evaluation obfuscates the actual algorithm but is necessary as there is no means of referring to a parameter without triggering its evaluation in an implicit-evaluation environment.

C++ Templates can be seen as a macro system that expands code to cater for different types. It can also be used to perform integer and pointer computations at compile-time.

All these syntactic abstraction mechanisms, being static techniques, share the same conceptual divide in a generating phase (macro expansion) and a phase where the generated code is executed. Sometimes the phases share the same language (e. g., Lisp) resulting in an even more unnatural separation as two syntactically largely identical-looking pieces of code cannot interact with each other because they are assigned to different execution stages.

Dynamic syntactic abstraction using operatives was pioneered by Shutt in his thesis [7]. His operatives are implemented using statically-scoped fexprs and work like normal functions (i. e., applicatives), except that the operand expressions are passed unevaluated.

Wand [8] demonstrated that the equational theory of fexprs is trivial which means that two expressions in the language can only be used interchangeably (are contextually equivalent) if they are syntactically identical (α-congruent). In essence, this observation precludes all optimizations as expressions cannot be replaced by anything except themselves without possibly altering the meaning of the program.

Shutt addresses this result and traces back the seeming contradiction with his thesis to differences in the modeled language. If there is any difference between two expressions that is observable by a fexpr, the two expressions are no longer contextually equivalent. In Wand's language, everything was an S-expression and could thus be deconstructed by fexprs. As a result, only S-expressions that were identical had contextual equivalence and the equational theory was indeed trivial. In Shutt's language, however, not every entity is a decomposable S-expression. There are encapsulated objects (environments, compound operatives) and computational states (active terms) which have a non-trivial equational theory and leave potential for optimizing transformations.

6 Conclusions

We believe that adapting a rigorous library system to an otherwise hardly restricted language is a viable first step to attain optimizing ahead-of-time compilation of programs with syntactic abstractions based on first-class operatives. Aim of this work is to demonstrate the feasibility of this approach.

References

1. Dybvig, R., Hieb, R., Bruggeman, C.: Syntactic abstraction in scheme. LISP and Symbolic Computation 5(4), 295–326 (1993), http://dx.doi.org/10.1007/BF01806308
2. Kohlbecker, E.E., Wand, M.: Macro-by-example: Deriving syntactic transformations from their specifications. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 77–84. POPL '87, ACM, New York, NY, USA (1987), http://doi.acm.org/10.1145/41625.41632
3. Rebol Special Interest Group: Rebol Script Library. http://www.rebol.org/script-index.r
4. Sassenrath, C.: Rebol design objectives for computer scientists. http://www.rebol.com/docs/design-objectives.html
5. Shinn, A., Cowan, J., Gleckler, A.A.: Revised^7 report on the algorithmic language Scheme. Tech. rep., Scheme Steering Committee (2013)
6. Shutt, J.N.: Revised^{-1} report on the kernel programming language. Tech. rep., Worcester Polytechnic Institute (2009)
7. Shutt, J.N.: Fexprs as the basis of Lisp function application or $vau: the ultimate abstraction. Ph.D. thesis, Worcester Polytechnic Institute (2010)
8. Wand, M.: The theory of fexprs is trivial. Lisp and Symbolic Computation 10(3), 189–199 (1998)