Reasonable macros through explicit bindings

Frederic Kettelhoit
Recurse Center
fred@fkettelhoit.com

***Abstract*—We present macros that can be statically reasoned about without requiring a macro expansion phase. This is achieved by distinguishing variables that are being *bound* from variables that are being *used*, in combination with *explicit block scopes*. The resulting macro system enables *local reasoning*, by supporting local evaluation without requiring full knowledge of the macros in scope.**

Introduction

Modern programming languages face a fundamental tension between expressiveness and static analyzability. While macro systems provide the flexibility to define custom control structures and extend language syntax, they often compromise the ability to reason about programs statically. This paper introduces a novel approach to resolve this tension through *explicit bindings*, a mechanism that enables macro-like expressiveness while preserving static reasoning capabilities.

Motivation

The design of extensible programming languages has long grappled with the challenge of allowing user-defined constructs without sacrificing analyzability. Consider the requirement that every function or control structure in a language should be redefinable, including fundamental operations such as variable assignment. While a language may provide built-in syntax for expressions like `a = 5`, the goal is to enable user-defined functions to achieve equivalent functionality, such as through a function `let` invoked as `let(a, 5)`.

Traditional approaches to this problem have relied on Lisp-style macro systems, dating back to Lisp 1.5 [1], [2]. In such systems, `let` can be implemented as a macro that treats its first argument `a` as an unevaluated symbol rather than a variable reference, evaluates the second argument `5`, and then invokes the built-in assignment operation to bind the symbol `a` to the value 5. However, this approach introduces significant complications:

1. **Complexity**: Macro systems add substantial complexity to the language, particularly regarding macro hygiene: the challenge of preventing unintended variable capture and ensuring that macro expansions do not interfere with the lexical scoping of the surrounding code [3].
2. **Static analysis impediments**: The presence of macros fundamentally undermines static reasoning about program behavior. Given an expression such as `foo(2 + 2)`, one cannot determine whether this can be simplified to `foo(4)` without first resolving whether `foo` is a macro. Macros can observe and act upon syntactic differences between expressions that are semantically equivalent, necessitating macro resolution before any program optimization or analysis can occur.

Contribution

This work proposes an alternative approach that achieves most of the expressiveness of traditional macro systems while maintaining the ability to perform static analysis. Our key insight is to explicitly distinguish between variables that are being **bound** (newly introduced into scope) and variables that are being **used** (referenced from existing scope). This distinction, combined with explicit block scoping, enables what we term **reasonable macros**: extensible language constructs that can be analyzed statically without prior macro expansion.

The fundamental guiding principle of our design is:

Every construct in the language can be redefined without privileged language constructs, while the scope and binding structure of variables remains immediately apparent from the source code alone, without requiring evaluation of any function or macro.

Technical approach

Our approach bridges the gap between fexprs and traditional macros through *selective evaluation* based on syntactic markers. The core insight is that evaluation behavior can be determined purely syntactically: expressions containing explicit binding markers remain unevaluated for structural manipulation, while unmarked expressions are evaluated normally. This guarantees that macros can observe syntactic differences only in the presence of explicit binding markers, in all other cases semantically equivalent expressions can be freely substituted for each other, ensuring that referential transparency is preserved.

Explicit bindings

We introduce a syntactic distinction that governs evaluation behavior:

- **Variable Usage**: Variables resolved from existing scope use standard notation (e.g., `x`)
- **Variable Binding**: Variables that introduce fresh bindings are prefixed with a marker (e.g., `:x`)

This marking scheme enables what we term *reasonable macros*: functions that can access syntactic structure when needed while preserving static reasoning for unmarked expressions:

```
// no bindings, equivalent to f(4)
f(2 + 2)

// :x remains unevaluated
f(:x)

// :x unevaluated, y + z evaluated
f(:x, y + z)
```

The presence of binding markers provides a syntactic guarantee about evaluation behavior, eliminating the need for runtime type checking (as in fexprs) or complete macro expansion (as in traditional macro systems).

Explicit block scope

To support static reasoning while allowing user-defined binding constructs, variable scope must be tracked syntactically. We employ explicit block syntax `{ ... }` with the innovation that binding declarations are separated from their scope blocks. Such a block is equivalent to a lambda abstraction that does not specify its bound variables explicitly but rather determines them based on the explicit bindings that appear to its left in the abstract syntax tree. This separation enables precise control over variable binding while maintaining syntactic clarity about scope boundaries.

As a more concrete example, let us consider the following function call:

```
foo(:x, y, { bar(x) })
//  |       \________/
//  |       scope of x
//  |
//  '-- binding of x
```

Both the scope and the origin of the variable `x` in the block `{ bar(x) }` are determined syntactically, without knowing the definition of `foo` (and thus without knowing whether `foo` is a macro or a regular function). Since the explicit binding `:x` appears to the left of the block `{ bar(x) }`, the block is desugared to a lambda abstraction with the bound variable `x`.

More precisely, the association between explicit bindings and blocks works as follows: A function call $f(f_1, \ldots, f_{m-1}, \{\text{body}\}, f_{m+1}, \ldots, f_n)$ where $\{\text{body}\}$ is a block argument at position m desugars the block $\{\text{body}\}$ to a lambda abstraction that binds all explicit bindings occurring in the arguments $f_0, \ldots, f_{m-1}$ that have not been consumed by other blocks appearing earlier in those arguments. The block $\{\text{body}\}$ is transformed into $\lambda x_1 \ldots x_k.\text{body}$ where $x_1, \ldots, x_k$ are the unconsumed explicit bindings from the preceding arguments, and these bindings are marked as consumed for subsequent blocks in the same function call or enclosing expressions.

Nested blocks

Sequential binding operations using the explicit form can lead to deeply nested structures that impair readability. Consider a series of variable bindings:

```
let(:a, x, {
  let(:b, y, {
    let(:c, z, {
      f(a, b, c)
    })
  })
})
```

While this nesting clearly shows the scope structure, it becomes unwieldy for longer sequences. To address this, we introduce syntactic sugar that allows blocks to consume bindings by enclosing them:

```
{
  let(:x, y),
  f(x)
}
```

This block-enclosed syntax is equivalent to the more explicit form `let(:x, y, { f(x) })`. The desugaring process recognizes that the `let` construct within the block contains an explicit binding `:x` and is followed by another element in the enclosing block, so the binding is automatically consumed by the enclosing block. This transformation preserves the static analyzability of the binding structure while providing more natural syntax for common patterns.

The approach scales naturally to multiple nested bindings:

```
{
  let(:a, x),
  let(:b, y),
  f(a, b)
}
```

This desugars to `let(:a, x, { let(:b, y, { f(a, b) }) })`, creating the expected nested scope structure. Each binding construct that contains explicit bindings automatically receives the remainder of the block as its block argument.

To handle expressions that do not introduce bindings (such as side effects), the system treats non-binding block elements differently. When a block element contains no explicit bindings that could be consumed by the enclosing block, the element is evaluated as an argument to an anonymous function that returns the rest of the block:

```
{
  let(:x, Foo),
  print("..."),
  use_x(x)
}
```

This desugars to an expression where the `print` call executes its side effect before the remainder of the block continues with access to the binding `x`. This mechanism allows natural mixing of binding constructs and effectful computations while maintaining the clear separation between binding and usage.

The nested block syntax preserves all the static reasoning properties of the explicit form. Variable bindings remain syntactically apparent, scope boundaries are clearly delineated, and the desugaring process produces standard lambda calculus constructs that can be analyzed and optimized using conventional techniques.

Macro definitions

To complete the macro system, we need mechanisms for defining constructs that can observe and manipulate the syntactic structure of explicitly marked bindings.

Macro definitions are distinguished from regular function definitions through a syntactic marker. A macro definition uses the `#` prefix (e.g., `#f = ...`), while regular function definitions use standard syntax (e.g., `:f = ...`). The environment tracks both the names in scope and their classification as either macros or regular values. This distinction matters only during the desugaring phase when translating to call-by-value lambda calculus; it does not impact the evaluation rules, which can continue to use standard lambda calculus environments.

When a macro is applied to arguments, its arguments undergo a static transformation that makes syntactic structure observable. Arguments are wrapped in data structures that preserve the distinction between evaluated expressions and syntactic elements that contain explicit bindings. This transformation occurs during the desugaring phase, before any evaluation takes place, and relies only on syntactic information.

$$\begin{aligned}
x &\to \text{Value}(x) && (1)\\
:x &\to \text{Binding}(\text{"x"}) && (2)\\
\{\ldots\} &\to \text{Block}(\ldots) && (3)\\
f(x,y) &\to \text{Value}(f(x,y)) && (4)\\
f(x,:y) &\to \text{Call}(\text{Value}(f),[\text{Value}(x),\text{Binding}(\text{"y"})]) && (5)\\
:f(x,y) &\to \text{Call}(\text{Binding}(\text{"f"}),[\text{Value}(x),\text{Value}(y)]) && (6)
\end{aligned}$$

In the case of `f(x, :y)`, the presence of the explicit binding `:y` prevents the entire expression from being evaluated. Instead, it is preserved as a `Call` structure that contains both evaluated components (`Value(x)`) and syntactic components (`Binding("y")`). This selective preservation allows macros to observe syntactic structure precisely where it is explicitly marked, while maintaining referential transparency for unmarked subexpressions.

More formally, when a macro f is applied to arguments $a_1, a_2, \ldots, a_n$, each argument a_i is transformed according to the function $\text{wrap}(a_i)$ defined as:

$$\text{wrap}(a) = \begin{cases} \text{Binding(name)} \\ \quad \text{if } a \text{ is a binding expression } :name \\ \text{Block(content)} \\ \quad \text{if } a \text{ is a block expression } \{\ldots\} \\ \text{Call}(\text{wrap}(f), [\text{wrap}(a_1), \ldots, \text{wrap}(a_n)]) \\ \quad \text{if } a = f(a_1, \ldots, a_n) \\ \quad \text{and wrap}(f) \neq \text{Value}(\ldots) \\ \text{Call}(\text{wrap}(f), [\text{wrap}(a_1), \ldots, \text{wrap}(a_n)]) \\ \quad \text{if } a = f(a_1, \ldots, a_n) \\ \quad \text{and } f \text{ is not a macro} \\ \quad \text{and any wrap}(a_i) \neq \text{Value}(\ldots) \\ \text{Value}(a) \\ \quad \text{otherwise} \end{cases}$$

Multi-level bindings

The basic explicit binding mechanism supports bindings that are active in the immediately following scope, but many programming constructs require bindings that persist across multiple scope levels. A prominent example is the definition of a recursive function, where the function being defined must be available both within its own definition (for recursive calls) and in the scope following the definition (for external use).

Multi-level bindings extend the explicit binding syntax to support this pattern through repeated markers. A binding `::x` remains active for the next two scopes, `:::x` for three scopes, and so on.

Consider the definition of a recursive function:

```
{
  ::factorial(:n) = {
    if(n == 0, 1, n * factorial(n - 1))
  },
  factorial(5)
}
```

The `::factorial` binding with two markers indicates that the function will be available both within its own definition (enabling the recursive call `factorial(n - 1)`) and in the subsequent scope (enabling the call `factorial(5)`).

The multi-level binding mechanism preserves static analyzability by making the scope lifetime explicit in the syntax. A static analyzer can determine the availability of any identifier by counting binding markers and tracking scope nesting levels, without requiring knowledge of the specific constructs being used.

Related work

The challenge of balancing expressiveness with static analyzability in metaprogramming has been explored through several distinct approaches, each with particular trade-offs between power and reasoning capabilities.

Fexprs and operatives

An alternative to macro-based metaprogramming emerged through the development of *fexprs*: functions that receive their arguments unevaluated and can selectively evaluate them in controlled environments [4]. This approach was later refined in Shutt's Kernel language, which distinguishes between *operatives* (functions that do not evaluate their arguments by default) and *applicatives* (functions that do evaluate their arguments) [5].

The fundamental insight of fexprs is that operatives represent a more primitive abstraction than applicatives, since any applicative can be constructed by wrapping an operative with automatic argument evaluation. However, this generality comes at the cost of static reasoning: because operatives can selectively evaluate or ignore their arguments, expressions like `f(2 + 2)` cannot be optimized to `f(4)` without first determining whether `f` is an operative or applicative.

Mitchell [6] and Wand [7] identified this limitation in their foundational work on fexprs, noting that the ability to observe syntactic structure necessarily impedes equational reasoning. While fexprs provide semantic abstraction without requiring phase separation between compile-time and run-time, they sacrifice the compiler's ability to perform optimizations based on expression equivalence.

Our approach represents a deliberate restriction of fexpr-style operatives, trading some expressive power for static analyzability. Unlike Kernel's operatives, which can dynamically choose whether to evaluate any argument, our approach determines evaluation behavior syntactically through binding markers. This restriction enables translation to standard call-by-value lambda calculus while preserving the ability to define custom binding constructs.

Restricted metaprogramming approaches

Several researchers have explored restricted forms of metaprogramming that preserve some static reasoning capabilities. These approaches generally involve constraining when and how syntactic structure can be observed, though they differ in their specific mechanisms and the extent of their restrictions.

The approach presented in this paper builds upon insights from both macro systems and fexprs while introducing novel syntactic constraints. By making variable bindings explicit through syntactic markers, we enable selective access to syntactic structure: arguments containing explicit bindings remain unevaluated for structural manipulation, while other arguments are evaluated normally. This provides a middle ground between the full power of fexprs and the static analyzability of conventional function calls.

Comparison with existing approaches

Our explicit binding approach differs from traditional macros in that it eliminates the need for complete macro expansion before optimization can occur. Unlike fexprs, it provides syntactic guarantees about when evaluation occurs, enabling static reasoning about expression equivalence. The key innovation is that evaluation behavior is determined syntactically by the presence of binding markers rather than by runtime type checking or compile-time macro resolution.

References

[1] T. P. Hart, "MACRO definitions for LISP," *AI Memos*, 1963.

[2] G. L. Steele and R. P. Gabriel, "The evolution of lisp," in *History of programming languages—II*, 1996, pp. 233–330.

[3] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba, "Hygienic macro expansion," in *Proceedings of the 1986 ACM conference on LISP and functional programming*, 1986, pp. 151–161.

[4] K. M. Pitman, "Special forms in lisp," in *Proceedings of the 1980 ACM conference on LISP and functional programming*, 1980, pp. 179–187.

[5] J. N. Shutt, PhD thesis, PhD thesis, Worcester Polytechnic Institute, September 2010, 2010.

[6] J. C. Mitchell, "On abstraction and the expressive power of programming languages," *Science of Computer Programming*, vol. 21, no. 2, pp. 141–163, 1993.

[7] M. Wand, "The theory of fexprs is trivial," *Lisp and Symbolic Computation*, vol. 10, no. 3, pp. 189–199, 1998.