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 and maintaining compatibility with conventional variable scoping patterns found in mainstream programming languages.

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 and preserving familiar variable scoping semantics. 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), using syntactic markers that align closely with how variable scopes work in mainstream programming languages.

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.*

Intuition

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 that are resolved use standard notation (e.g., `x`)
- **Variable Binding**: Variables that introduce fresh bindings are marked syntactically (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:

Consider destructuring assignment as our running example:

```
// assuming point is in scope
(:x, :y) = point  // x and y are bound
use_point(x, y)   // x and y are used
```

Here, the assignment operator = can be implemented as a user-defined infix macro that observes the binding structure `(:x, :y)` while evaluating `point` normally. The scope of the newly bound variables `x` and `y` extends through the remainder of the enclosing block.

The syntactic difference between bound and used variables makes it immediately clear which subexpressions can be evaluated, even in the presence of macros. The presence of binding markers provides a syntactic guarantee about evaluation behavior, eliminating the need for runtime evaluation (as in fexprs) or complete macro expansion (as in traditional macro systems).

Without knowing how the assignment macro `=` is defined, it is immediately obvious which argument (sub-)expressions are evaluated:

```
// x and y are bound
(:x, :y) = point

// x is bound, z is resolved
(:x, z) = point

// x is bound, 2 + 2 is evaluated
(:x, 2 + 2) = point

// x is bound
(:x, :x) = point
```

If `=` implements standard pattern matching with unification (and throws an exception if the pattern on the left does not match the value on the right), the first example would be an irrefutable match, whereas the second and third examples would match only if the second element of the pair has a particular value, while the last example would only match if both elements can be unified.

Crucially, however, it is safe to evaluate `2 + 2` even if `=` is implemented differently and e.g., does not implement pattern matching at all. In contrast to traditional macro systems, the syntactic distinction between bound and used variables guarantees that evaluation behavior remains statically tractable and exposes only explicitly annotated expressions as syntactically observable macro arguments.

Explicit scope

It is common for binding constructs in traditional languages to bind variables not just in the *enclosing* scope, but also in *explicit block scopes* that are used as part of a binding construct. Examples are constructs for declaring anonymous functions, which bind function arguments in the body of the function, as well as pattern matching constructs, which bind pattern variables in the body of a match clause.

We will mark explicit block scope syntactically by enclosing a sequence of expressions in `{...}`, with the innovation that binding declarations are separated from their scope blocks. 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.

Whenever a function is called with an explicit block as one of its arguments, the evaluation behavior of the arguments preceding the block is defined as follows:

- **Variable Binding**: Variables that introduce fresh bindings use standard notation (e.g., `x`)
- **Variable Usage**: Variables that are resolved are marked syntactically (e.g., `^x`)

This marking scheme is thus dual to the marking scheme used for the enclosing scope: In the enclosing scope, bound variables are explicitly marked, whereas for explicitly marked block scopes, used variables are explicitly marked.

Consider the pattern matching operator `->` as an example involving explicit block scope:

```
// assuming point and z are in scope
match (point) [
    // x bound, z used
    (x, ^z) -> {
      f(x)
    }
]
```

As in the case of explicit bindings used within the enclosing scope, it is immediately obvious which argument (sub-)expressions of `->` are evaluated:

```
match (point) [
  (x, x) -> {
    // x is bound
  }
  (x, ^z) -> {
    // x is bound, z is resolved
  }
  (x, y) -> {
    // x and y are bound
  }
]
```

Combining explicit and implicit bindings

Some programming constructs require bindings that are bound both in the enclosing scope and inside of an explicit block argument. 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). This is easily supported by combining explicit bindings and block scope arguments.

Consider the definition and application of a recursive function:

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

The explicit `:factorial` binding indicates that the variable will be bound both within its own definition (since it is followed by the explicit block argument of `=`, enabling the recursive call `factorial(n - 1)`) and in the enclosing scope (since it is explicitly marked, enabling the call `factorial(5)`).

Macro resolution through usage

Unlike traditional macro systems that require explicit macro definitions, our approach determines whether a function should be treated as a macro based on how it is used. This usage-based resolution eliminates the need for separate macro definition syntax while preserving static analyzability.

A function call is treated as a macro (with arguments wrapped in syntactic annotations) if any of the following conditions hold:

1. **Explicit bindings**: The function is called with explicit binding arguments that can be used in the enclosing scope: `f(:x, y)` or `:result = f(args)`
2. **Block arguments**: The function is called with block arguments: `f(args, {...})`

When a function is identified as a macro call, its arguments undergo a static transformation that preserves syntactic structure where needed. We first give an informal intuition by listing several examples and formalize the algorithm in the next section. Syntactic annotations add a runtime-observable tag to their wrapped data and are written in smallcaps:

For the enclosing scope:

$$x \to \textsc{Val}(x) \tag{1}$$
$$: x \to \textsc{Bind}(\text{"x"}) \tag{2}$$
$$\{\dots\} \to \textsc{Block}(\lambda \dots) \tag{3}$$
$$f(x, y) \to \textsc{Val}(f(x, y)) \tag{4}$$
$$f(: x, y) \to \textsc{Call}(\textsc{Val}(f), [\textsc{Bind}(\text{"x"}), \textsc{Val}(y)]) \tag{5}$$

For explicit scope arguments:

$$^\wedge x \to \textsc{Val}(x) \tag{6}$$
$$x \to \textsc{Bind}(\text{"x"}) \tag{7}$$
$$\{\dots\} \to \textsc{Block}(\lambda \dots) \tag{8}$$
$$^\wedge f(^\wedge x, ^\wedge y) \to \textsc{Val}(f(x, y)) \tag{9}$$
$$^\wedge f(x, ^\wedge y) \to \textsc{Call}(\textsc{Val}(f), [\textsc{Bind}(\text{"x"}), \textsc{Val}(y)]) \tag{10}$$

This selective preservation allows macros to observe syntactic structure precisely where it is explicitly marked, while maintaining referential transparency for unmarked subexpressions.

Lambda Transformation

We will now formalize the macro system by presenting an algorithm that translates bindings and blocks to call-by-value lambda calculus. The algorithm consists of three individual translations:

1. A translation of macro arguments into arguments wrapped with syntactic annotations.
2. A translation of a sequence of expressions that might include explicit bindings inside of an enclosing scope into nested lambda abstractions.
3. A translation of function calls with implicit bindings and explicit block arguments into lambda abstractions.

Since the last two translations depend on the first translation, we will begin by formalizing the notion of wrapping macro arguments with runtime tags that was outlined informally in the last section.

Macro argument wrapping

To be able to use the same algorithm for both enclosing scopes and explicit block arguments, we abstract over the specific syntax by distinguishing *binding names* from other names and define binding names as an explicit binding in the context of an enclosing scope or an implicit binding in the context of explicit block arguments, which captures the duality of these contexts.

When a macro f is applied to arguments $a_1, a_2, \dots, a_n$, each argument a_i is transformed according to the function wrap(a_i) defined as:

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

We consider built-in data structures such as tuples as function calls that build up these data structures, so that a tuple such as `(x, y)` can be treated as the call `tuple(x, y)` for the purposes of macro argument wrapping.

Notice that our algorithm supports bindings being used in the position of a function that is applied to arguments. This is useful if the goal is to expose and match against the structure of nested data structures, similar to the polymorphism supported in pattern calculus [4].

Another possibility would be to disallow bindings to be used as functions in the context of an enclosing scope and to treat a function without a ^ marker as a *value* instead of a binding name in the context of block arguments. This would lead to fewer annotations at the price of breaking the duality between the two contexts.

Enclosing scope

For enclosing scope contexts, expressions containing explicit bindings are converted to lambda abstractions that bind variables for the remainder of the scope.

```
(:x, :y) = point
use_point(x, y)
```

With `WRAP` as the translation described in the last section and `x => y` as a lambda abstraction with argument `x` and body `y`, the above desugars to:

```
(=)(
  WRAP((:x, :y)),
  point,
  x => y => use_point(x, y)
)
```

If an expression does not contain any explicit bindings, it is treated as a side effect and passed as an argument to a lambda abstraction that ignores its argument, which effectively sequences the side effects in the expected order:

```
f(x, y)
g(z)
```

Desugars to:

```
(_ => g(z))(f(x, y))
```

The translation of a whole block is then a fold over the sequence of expressions, starting with the last expression of the block as the initial element:

```
(:x, :y) = point
f(x, y)
use_point(x, y)
```

Desugars to:

```
(=)(
  WRAP((:x, :y)),
  point,
  x => y => ((_ => g(z))(f(x, y)))
)
```

More formally, let $E = [e_1, e_2, \ldots, e_n]$ be a sequence of expressions within an enclosing scope. The translation $\mathcal{T}(E)$ is defined recursively using a right fold operation:

$$\mathcal{T}([e]) = e \quad \text{(base case)}$$

$$\mathcal{T}([e_1, e_2, \ldots, e_n]) = \begin{cases} \mathcal{M}(e_1, \mathcal{T}([e_2, \ldots, e_n])) \\ \quad \text{if } e_1 \text{ contains expl. bindings} \\ (\lambda\_.\mathcal{T}([e_2, \ldots, e_n]))(e_1) \\ \quad \text{otherwise} \end{cases}$$

where $\mathcal{M}(e, \text{cont})$ represents the macro transformation of expression e with continuation cont, and is defined as:

$$\mathcal{M}(f(a_1, \ldots, a_k), \text{cont}) =$$

$$f(\text{wrap}(a_1), \ldots, \text{wrap}(a_k), \lambda x_1 \ldots x_m.\text{cont})$$

where $x_1, \ldots, x_m$ are the variables bound by the explicit bindings in $a_1, \ldots, a_k$.

For expressions without function calls but containing explicit bindings (such as standalone binding declarations), the transformation follows the same pattern by treating the binding construct as a macro that introduces variables into the continuation.

Notice that as a consequence of using enclosing blocks as the mechanism for both binding variables and sequencing effects, it is not possible to use a macro without binding any variables. For example, it is not possible to use destructuring assignment to match the value `x` against the value `y` by writing it as `x = y`, because it would be interpreted as a side effect due to its lack of explicit bindings. We consider this acceptable but note that explicit syntax could be introduced to distinguish these two cases.

Block arguments

For block argument contexts, the lambda translation is determined by the implicit bindings that precede the block argument in the abstract syntax tree.

```
(x, y) -> { use_point(x, y) }
```

Desugars to:

```
(->)(
  WRAP((x, y)),
  x => y => use_point(x, y)
)
```

More precisely, implicit bindings and explicit uses in the presence of block scope arguments are translated to lambda terms as follows: A function call $f(a_1, \ldots, a_{m-1}, \{\text{body}\}, a_{m+1}, \ldots, a_n)$ where $\{\text{body}\}$ is a block argument at position m is transformed as:

$$f(\text{wrap}(a_1), \ldots, \text{wrap}(a_{m-1}), \lambda x_1 \ldots x_k.\text{body},$$

$$\text{wrap}(a_{m+1}), \ldots, \text{wrap}(a_n))$$

where $x_1, \ldots, x_k$ are the implicit bindings extracted from the preceding arguments $a_1, \ldots, a_{m-1}$.

The implicit bindings are determined by the function bind(a) defined as:

$$\text{bind}(a) = \begin{cases} \{a\} & \text{if } a \text{ is a binding name} \\ \bigcup_{i=1}^{n} \text{bind}(a_i) & \text{if } a = g(a_1, \ldots, a_n) \\ \emptyset & \text{otherwise} \end{cases}$$

The order of the lambda parameters $x_1, \ldots, x_k$ follows the left-to-right traversal order of their first occurrence in the abstract syntax tree of the preceding arguments.

For multiple block arguments, each block receives the bindings from the unconsumed arguments that precede it:

$$f(a_1, \ldots, a_i, \{\text{body}_1\}, a_{i+1}, \ldots, a_j, \{\text{body}_2\}, \ldots)$$

becomes:

$$f(\text{wrap}(a_1), \ldots, \text{wrap}(a_i), \lambda\vec{x}.\text{body}_1,$$
$$\text{wrap}(a_{i+1}), \ldots, \text{wrap}(a_j), \lambda\vec{y}.\text{body}_2, \ldots)$$

where $\vec{x}$ represents the bindings from $a_1, \ldots, a_i$ and $\vec{y}$ represents the bindings from $a_{i+1}, \ldots, a_j$.

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 [5]. 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) [6].

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 [7] and Wand [8] 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] C. B. Jay, “The pattern calculus,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 6, pp. 911–937, 2004.

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

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

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

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