A pattern matcher for miniKanren
or
How to get into trouble with CPS macros

Andrew W. Keep Michael D. Adams Lindsey Kuper William E. Byrd Daniel P. Friedman

Indiana University, Bloomington, IN 47405

{akeep,adamsmd,lkuper,webyrd,dfried}@cs.indiana.edu

Abstract

CPS macros written using Scheme's **syntax-rules** macro system allow for guaranteed composition of macros and control over the order of macro expansion. We identify a limitation of CPS macros when used to generate bindings from a non-unique list of user-specified identifiers. Implementing a pattern matcher for the miniKanren relational programming language revealed this limitation. Identifiers come from the pattern, and repetition indicates that the same variable binding should be used. Using a CPS macro, binding is delayed until after the comparisons are performed. This may cause free identifiers that are symbolically equal to be conflated, even when they are introduced by different parts of the source program. After expansion, this leaves some identifiers unbound that should be bound. In our first solution, we use **syntax-case** with *bound-identifier=?* to correctly compare the delayed bindings. Our second solution uses eager binding with **syntax-rules**. This requires abandoning the CPS approach when discovering new identifiers.

1. Introduction

Macros written in continuation-passing style (CPS) [4, 6] give the programmer control over the order of macro expansion. We chose the CPS approach for implementing a pattern matcher for miniKanren, a declarative logic programming language implemented in a pure functional subset of Scheme [1, 3]. This approach allows us to generate clean miniKanren code, keeping bindings for logic variables in as narrow a scope as possible without generating additional binding forms. During the expansion process, the pattern matcher maintains a list of user-specified identifiers we have encountered, along with the locations in which bindings should be created for them. We accomplish this by using a macro to compare an identifier with the elements of one or more lists of identifiers. Each clause in the macro contains an associated continuation that is expanded if a match is found. The macro can then determine when a unification is unnecessary, when an identifier is already bound, or when an identifier requires a new binding.

While CPS and conditional expansion seemed, at first, to be an effective technique for implementing the pattern matcher, we discovered that the combination of delayed binding of identifiers and conditional expansion based on these identifiers could cause free variables that are symbolically equal to be conflated, even when they are generated from different positions in the source code. The result of conflating two or more identifiers is that only the first will receive a binding. This leaves the remaining identifiers unbound in the final expression, resulting in unbound variable errors.

This issue with delaying identifier binding while the CPS macros expand suggests that some care must be taken when writing macros in CPS. In particular, CPS macros written using Scheme's **syntax-rules** macro system are limited in their ability to compare two identifiers and conditionally expand based on the result of the comparison. The only comparison available to us under **syntax-rules** is an auxiliary keyword check that is the operational equivalent of **syntax-case**'s *free-identifier=?* predicate. Unfortunately, when we use such a comparison, identifiers that are free and symbolically equal may be incorrectly understood as being lexically the same.

In our implementation, the pattern matcher exposes its functionality to the programmer through the $\boldsymbol{\lambda}^{\boldsymbol{e}}$ and **match**$^{\boldsymbol{e}}$ forms. We begin by describing the semantics of $\boldsymbol{\lambda}^{\boldsymbol{e}}$ and **match**$^{\boldsymbol{e}}$ and giving examples of their use in miniKanren programs in section 2. In section 3, we present our original implementation of the pattern matcher, and in section 4 we demonstrate how the issue regarding variable binding can be exposed. We follow up in section 5 by presenting two solutions to the variable-binding issue, the first using **syntax-case** and the second using eager binding with **syntax-rules**.

2. Using $\boldsymbol{\lambda}^{\boldsymbol{e}}$ and match$^{\boldsymbol{e}}$

Our aim in implementing a pattern matcher was to allow automatic variable creation similar to that found in the Prolog family of logic programming languages. In Prolog, the first appearance of a variable in the definition of a logic rule leads to a new logic variable being created in the global environment. The $\boldsymbol{\lambda}^{\boldsymbol{e}}$ and **match**$^{\boldsymbol{e}}$ macros described below allow the miniKanren programmer to take advantage of the power and concision of Prolog-style pattern matching with automatic variable creation, without changing the semantics of the language.

2.1 Writing the *append* relation with $\boldsymbol{\lambda}^{\boldsymbol{e}}$

Before describing $\boldsymbol{\lambda}^{\boldsymbol{e}}$ and **match**$^{\boldsymbol{e}}$ in detail, we motivate our discussion of pattern matching by looking at a common operation in logical and functional programming languages—appending two lists. In Prolog, the definition of *append* is very concise:

2009 Workshop on Scheme and Functional Programming

```
append([],Y,Y).
append([A|D],Y2,[A|R]) :- append(D,Y2,R).
```

We first present a version of *append* in miniKanren without using $\boldsymbol{\lambda}^e$ or **match**^{e}. Without pattern matching, the *append* relation in miniKanren is surprisingly verbose when compared with the Prolog equivalent:

```
(define append
  (λ (x y z)
    (cond^e
      ((≡ '() x) (≡ y z))
      ((exist (a d r)
         (≡ '(,a . ,d) x)
         (≡ '(,a . ,r) z)
         (append d y r))))))
```

Using $\boldsymbol{\lambda}^e$, the miniKanren version can be expressed almost as succinctly as the Prolog equivalent:

```
(define append
  (λ^e (x y z)
    ((() __ ,y))
    (((,a . ,d) __ (,a . ,r)) (append d y r))))
```

The two match clauses of the $\boldsymbol{\lambda}^e$ version of *append* correspond to the two rules in the Prolog version. In the first match clause, x is unified with () and z with y. In the second clause, x is unified with a pair that has a as its *car* and d as its *cdr*, and z is unified with a pair that has the same a as its *car* and a fresh r as its *cdr*. The *append* relation is then called recursively to finish the work.

No new variables need be created in the first clause, since the only variable referenced, y, is already in the $\boldsymbol{\lambda}^e$ formals list. In the second clause, $\boldsymbol{\lambda}^e$ is responsible for creating bindings for a, d, and r. In both clauses, the double underscore __ indicates a position in the match that has a value we do not care about. No unification is needed here, since no matter what value y has, it will always succeed and need not extend the variable environment. We also have the option of using ,y instead of __ because $\boldsymbol{\lambda}^e$ recognizes a variable being matched against itself and avoids generating the unnecessary unification.

With the *append* relation defined we can now use miniKanren's **run** interface to test the relation.

```
(run 1 (t) (append '(a b c) '(d e f) t)) ⇒
((a b c d e f))
```

where 1 indicates only one answer is desired and t is the logic variable bound to the result. Because *append* is a relation we can also use it to generate the input lists that would give us (a b c d e f).

```
(run 5 (t)
  (exist (x y)
    (append x y '(a b c d e f))
    (≡ '(,x ,y) t))) ⇒
((() (a b c d e f))
 ((a) (b c d e f))
 ((a b) (c d e f))
 ((a b c) (d e f))
 ((a b c d) (e f)))
```

where 5 indicates five answers are desired and x and y are uninstantiated variables used to represent the first and second lists. *append* then returns the first five possible input list pairs that when appended yield (a b c d e f).

2.2 Syntax and semantics of $\boldsymbol{\lambda}^e$

Having seen $\boldsymbol{\lambda}^e$ in action, we now formally describe its syntax and semantics. The syntax of a $\boldsymbol{\lambda}^e$ expression is:

```
(λ^e formals
  (pattern1 goal1 ...)
  (pattern2 goal2 ...)
  ...)
```

where *formals* may be any valid λ formal arguments expression, including those for variable-length argument lists. *formals* is the expression to be matched against in the match clauses that follow. Each match clause begins with a pattern followed by zero or more user-supplied goals. The pattern and user-supplied goals represent a conjunction of goals that must all be met for the clause to succeed. Taken together, the clauses represent a disjunction and expand into the clauses of a miniKanren **cond**^{e} (disjunction) expression [3], hence the name $\boldsymbol{\lambda}^e$. The pattern within each clause is then further expanded into a set of variable bindings using miniKanren's **exist** and unification operators as necessary.

If no additional goals are supplied by the programmer, then the unifications generated by the pattern will comprise the body of the generated **cond**^{e} clause. Otherwise, the user-supplied goals will be evaluated in the scope of the variables created by the pattern. The first match clause of *append* requires no user-supplied goal, while the second clause uses a user-supplied goal to provide the recursion. It is important to note that $\boldsymbol{\lambda}^e$ does not attempt to identify unbound identifiers in user-supplied goals, only those in the pattern. Any variables needed in the user-supplied goals not named in the formals list or pattern will need to be bound with an **exist** explicitly by the user.

The pattern matcher recognizes the following forms:

() The null list.

__ Similar to Scheme's _, the double underscore __ represents a position where an expression is expected, but its value can be ignored.

,x A logic variable x. If this is the first appearance of x in the pattern and it does not appear in the formals list of $\boldsymbol{\lambda}^e$, a new logic variable will be created.

'e Preserves the expression e. This is provided as an escape for special forms where the exact contents should be preserved. For example, if we wish to match the symbol __ rather than having it be treated as an ignored position, we could use '__ in our pattern. $\boldsymbol{\lambda}^e$ would then know to override the special meaning of __.

sym Where *sym* is any Scheme symbol, other then those assigned special meaning, such as __. These will be preserved in the unification as Scheme symbols.

(a . d) Arbitrarily nested pairs and lists are also allowed, where a and d are stand-ins for the *car* and *cdr* positions of the pair. This also allows us to create arbitrary list structures, as is normally the case with pairs in Scheme.

When processing the pattern for each clause, $\boldsymbol{\lambda}^e$ breaks the pattern down into parts which correspond to the members of the *formals* list. The list of parts is then processed from left to right, with *formals* as the initial list of known variables. As $\boldsymbol{\lambda}^e$ encounters fresh variable references in each part, it adds them to the known-variables list. If a part is __, or if it is the variable appearing in the corresponding position in *formals*, no unification is necessary. Otherwise, a unification between the processed pattern and the appropriate *formals* variable will be generated.

2.3 Syntax and semantics of match^e

match^e is similar to $\boldsymbol{\lambda}^e$ in syntax, and it recognizes the same patterns. Unlike $\boldsymbol{\lambda}^e$, however, there is no *formals* list, so the list of known variables starts out effectively empty. Strictly speaking, the known-variables list contains the temporary variable introduced to bind the expression in **match**^{e}, which simplifies the implementation of **match**^{e} by making it possible to use the same helper macros

as $\boldsymbol{\lambda}^{e}$. However, since this temporary variable is introduced by a **let** expression generated by **match**^{e}, hygiene ensures that it will never inadvertently match a variable named in the pattern.

match^{e} has the following syntax:

```
(match^e expr
  (pattern_1 goal_1 ...)
  (pattern_2 goal_2 ...)
  ...)
```

where *expr* is any Scheme expression. Similar to other pattern matchers, **match**^{e} **let**-binds *expr* to a temporary variable to ensure it is only computed once. Unlike $\boldsymbol{\lambda}^{e}$, which may generate multiple unifications for each clause, **match**^{e} only generates one unification per clause, since it matches each pattern with the variable bound to *expr* as a whole.

Since **match**^{e} can be used on arbitrary expressions, it provides more flexibility then $\boldsymbol{\lambda}^{e}$ in defining the matches. For instance, we may want to define the *append* relation using only one of the formal arguments in the match. Consider the following definition of *append*.

```
(define append
  (λ (x y z)
    (match^e x
      (() (≡ y z))
      ((,a . ,d)
       (exist (r)
         (≡ `(,a . ,r) z)
         (append d y r))))))
```

Here we have chosen to match against only the first list in the relation, supplying the unifications necessary for the other formal variables. The first clause matches x to () and unifies y and z. The second clause decomposes the list in x into a and d, then uses **exist** to bind r and unifies `(,a . ,r)` with z. Finally it recurs on the *append* relation to finish calculating the appended lists. This clause requires an explicit **exist** be used to bind r since it is not a formal or pattern variable.

Both implementations of $\boldsymbol{\lambda}^{e}$ and **match**^{e} were designed for use in R^{5}RS, but can be ported to an R^{6}RS library with relative ease, as long as care is taken to ensure that the `_` auxiliary keyword is exported with the library.

3. Implementation

Our primary objective in adding pattern-matching capability to miniKanren is to provide convenience to the programmer, but we would prefer that convenience not come at the expense of efficiency. Indeed, we would like to generate the cleanest correct programs possible, so that we can get good performance from the results of our macros.

Since relational programming languages like miniKanren return all possible results from a relation, we would like goals that will eventually reach failure to do so as quickly as possible. In keeping with this "fail fast" principle, we follow two guidelines. First, we limit the scope of logic variables as much as possible. While introducing new logic variables is not an especially time-consuming process, we would still prefer to avoid creating logic variables we will not be using. Second, we generate as few **exist** forms as possible. Minimizing the number of **exist** forms in the code generated by $\boldsymbol{\lambda}^{e}$ and **match**^{e} aids efficiency. **exist** wraps its body in two functions. The first is a monadic transform to thread miniKanren's substitution through the goals in its body. The second generates a thunk to allow miniKanren's interleaving search to work through the goals appropriately. This means that each **exist** may cause multiple closures to be generated, and we would like to keep these to a minimum.

To illustrate the benefit of keeping the scope of logic variables as tight as possible, consider the following example:

```
(exist (x y z) (≡ `(,x . ,y) `(a . b)) (≡ x y) (≡ z `c))
```

Here, we create bindings for x, y, and z, even though z will never be used. $(\equiv x\ y)$ will fail since $(\equiv$ `(,x . ,y)` `(a . b))` binds x to `a` and y to `b`, so z is never encountered. However, we can tighten the lexical scope for z as follows:

```
(exist (x y) (≡ `(,x . ,y) `(a . b)) (≡ x y) (exist (z) (≡ z `c)))
```

The narrower scope around z helps the **exist** clauses to fail more quickly, cutting off miniKanren's search for solutions. This example illustrates the trade-off inherent in our twin goals of keeping each variable's scope as narrow as possible and minimizing the overall number of **exist** clauses. Our policy has been to allow more **exist** clauses to be generated when it will tighten the scope of variables. As we continue to explore various performance optimizations in miniKanren, the pattern matcher could benefit from more detailed investigation to determine if the narrowest-scope-possible policy wins more often then it loses.

3.1 $\boldsymbol{\lambda}^{e}$ and match^{e}

All three of our implementations for the pattern matcher expose their functionality to the programmer via the $\boldsymbol{\lambda}^{e}$ and **match**^{e} macros. $\boldsymbol{\lambda}^{e}$ and **match**^{e} are implemented as follows:

```
(define-syntax λ^e
  (syntax-rules ()
    ((_ args c c* ...)
     (λ args (handle-clauses args (c c* ...))))))


(define-syntax match^e
  (syntax-rules ()
    ((_ e c c* ...)
     (let ((t e)) (handle-clauses t (c c* ...))))))
```

The interface to these two macros is shared by all three implementations. In all three cases, $\boldsymbol{\lambda}^{e}$ and **match**^{e} use the same set of macros to implement their functionality.

In general, the CPS macro approach [4, 6] seems well-suited for our purposes in implementing a pattern matcher in that parts of the pattern must be reconstructed for use during unification and bindings for variables must be generated outside these unifications. Since the CPS macro approach gives us the ability to control the order of expansion, we decided to take an "inside-out" approach: clauses are processed first, and the **cond**^{e} form is then generated around all processed clauses, rather than first expanding the **cond**^{e} and then expanding clauses within it. This inside-out expansion allows us to process patterns from left to right without needing to worry about nesting later unifications and user-supplied goals into the **exist** clauses as we go. Patterns must be processed from left to right to ensure we are always generating an **exist** binding form for the outermost occurrence of an identifier. The entire pattern of a clause is processed, with each part of the pattern being transformed into a unification; any variables that require bindings to be generated for them are put into a flat list of unifications in the order they occur.

As an example, consider the $\boldsymbol{\lambda}^{e}$ version of the *append* relation from the previous section. At expansion time, the pattern in the second clause is processed into the following flat list of unifications (with embedded indicators of where new variables need to be bound):

```
((ex a d) (≡ (cons a d) x) (ex r) (≡ (cons a r) z))
```

Here (**ex** *a d*) and (**ex** *r*) indicate the places where new variables need to be bound with an **exist** clause. The **build-clause** macro, described below, then takes this list, along with user-specified goals (if any) and a continuation, and calls the continuation on the completed clause, which looks like this after expansion:

```
(exist (a d)
  (≡ (cons a d) x)
  (exist (r)
    (≡ (cons a r) z)
    (append d y r)))
```

where The **exist** forms and unifications were generated as a result of matching the pattern with the λ^e formals list, and (*append d y r*) was the user-specified goal. When both clauses of the *append* relation have been processed and wrapped in a single **cond**^{e}, *append* expands to

```
(define append
  (λ (x y z)
   (cond^e
     ((≡ '() x) (≡ y z))
     ((exist (a d)
        (≡ (cons a d) x)
        (exist (r)
          (≡ (cons a r) z)
          (append d y r)))))))
```

In this example, the first clause does not require any **exist** clauses, since it does not introduce any new bindings.

3.2 CPS macro implementation

Aside from the user-facing λ^e and **match**^{e}, the CPS macro implementation of the pattern matcher comprises ten macros: two macros for decomposing clauses and patterns; two helper macros for constructing continuation expressions; five macros for building up clauses, unifications, and expressions; and one macro for matching identifiers to determine when bindings have been seen before. As a guide to the reader, the macros used to decompose clauses and patterns have names starting with **handle**; the helper macros for constructing continuations have names starting with **make**; and the macros used to build up discovered parts of clauses, unifications, and expressions have names starting with **build**. Finally, the **case-id** macro is used to match identifiers in much the same way Scheme's **case** is used to match symbols. We have also endeavoured to use consistent naming conventions for the variables used in the **handle**, **make**, and **build** macros, as follows:

***a*, *a*∗** indicate an argument (*a*) or list of arguments (*a*∗).

***p*, *p*∗, *pr*∗** indicate a part (*p*), parts (*p*∗), or the patterns remaining to be processed (*pr*∗) from the initial pattern.

***g*∗, *g*∗∗** indicate goals from a clause (*g*∗) or the remaining clauses (*g*∗∗).

***pc*∗, *pp*∗, *pg*∗** indicate a list of processed clauses (*pc*∗), processed pattern parts (*pp*∗), and processed goals (*pg*∗).

***k*∗** indicates the continuation for the macro.

***svar*∗** indicates a list of variables we have already seen in processing the pattern.

***evar*∗** indicates a list of variables that need to be bound with **exist** for the unification currently being worked on.

pa*, *pd indicate the *car* (*pa*) and *cdr* (*pd*) positions of a pattern pair.

3.2.1 The *handle* macros

The **handle-clauses** and **handle-pattern** macros implement the forward phase of pattern processing and are responsible for breaking the λ^e and **match**^{e} clauses and patterns down into parts for the **build** macros to reconstruct. The **handle-clauses** macro is implemented as follows:

```
(define-syntax handle-clauses
  (syntax-rules ()
    ((_ a* () . pc*) (cond^e . pc*))
    ((_ (a . a*) (((p . p*) . g*) (pr* . g**) ...) . pc*)
     (make-clauses-cont
       (a . a*) a a* p p* g* ((pr* . g**) ...) . pc*))
    ((_ a ((p . g*) (pr* . g**) ...) . pc*)
     (make-clauses-cont a a () p () g* ((pr* . g**) ...) . pc*))))
```

handle-clauses transforms the list of λ^e and **match**^{e} clauses into a list of **cond**^{e} clauses. The first rule recognizes when the list of λ^e clauses to be processed is empty and generates a **cond**^{e} to wrap the processed clauses *pc*∗. The second and third rules both serve to decompose the clauses, processing each one in order using the **make-clauses-cont** macro described below. The second rule processes clauses of λ^e expressions where the *formals* start with a pair. The third rule handles **match**^{e} clauses where the expression to be matched is **let**-bound to a temporary and λ^e clauses where the formal is a single identifier rather than a list.

handle-pattern is where the main work of the pattern matcher takes place. It is responsible for deciding when new logic variables need to be introduced and generating the expressions to be unified against in the final output.

```
(define-syntax handle-pattern
  (syntax-rules (quote unquote top __)
    ((_ top a __ (k* ...) svar* evar* pp* ...)
     (k* ... svar* evar* pp* ...))
    ((_ tag a __ (k* ...) svar* evar* pp* ...)
     (k* ... (t . svar*) (t . evar*) pp* ... t))
    ((_ tag a () (k* ...) svar* evar* pp* ...)
     (k* ... svar* evar* pp* ... ()))
    ((_ tag a (quote p) (k* ...) svar* evar* pp* ...)
     (k* ... svar* evar* pp* ... (quote p)))
    ((_ tag a (unquote p) (k* ...) svar* evar* pp* ...)
     (case-id p
       ((a) (k* ... svar* evar* pp* ...))
       (svar* (k* ... svar* evar* pp* ... p))
       (else (k* ... (p . svar*) (p . evar*) pp* ... p))))
    ((_ tag a (pa . pd) k* svar* evar* pp* ...)
     (handle-pattern inner t1 pa
       (handle-pattern inner t2 pd
         (build-cons k*)) svar* evar* pp* ...))
    ((_ tag a p (k* ...) svar* evar* pp* ...)
     (k* ... svar* evar* pp* ... 'p))))
```

The first two rules both match the __ "ignore" pattern. However, the first rule is distinguished by its use of the **top** auxiliary keyword indicating that it is at the top level of the pattern, i.e., it will be matched directly with an input variable, either a λ^e formal or **let**-bound temporary variable for the **match**^{e} expression. In either case, no unification is needed, so we do not extend the list of processed pattern parts *pp*∗. In the second rule, we know that __ must be nested within a pair, so a new logic variable is generated to indicate that an expression is expected here, even though we do not care what the value of the expression is. Since the logic variable is generated as a temporary, it will not clash with any other variable already bound, thanks to hygienic macro expansion.

The remaining rules do not require this special handling around the top element, and so they ignore the "tag" supplied as the first part of the pattern. The third, fourth, and seventh rules handle the null, quoted expression, and bare symbol cases, respectively. In all of these cases, the continuation is invoked with either a null list or a quoted expression. If we are at the top level of the

pattern, the continuation builds a unification directly using **build-goal**; otherwise, it builds a more complex expression using **build-cons**. The CPS nature of the macro, however, frees us from having to concern ourselves with the kind of expression generated.

The sixth rule handles pairs. Here, **handle-pattern** is called on the *car* of the pair, *pa*, with a continuation that processes the *cdr* of the pair, *pd*, which in turn calls the **build-cons** continuation to build the *cons* pair. The continuations are each created with the part of the current state required for them to finish their jobs, relying on the application sites for the continuation to fill in any extra arguments. This is why expressions of the form ($k* \ldots$ *args* $\ldots$) are so prevalent in our macros. Note that in both calls to **handle-pattern**, *inner* is specified to ensure that if _ is encountered, it will recognize that it is no longer at the top level of the pattern.

Finally, the fifth rule in **handle-pattern** determines if a unification can be skipped, because it is unifying a variable with itself; if the identifier is already a bound variable; or if a new binding is needed for this variable. **case-id** provides the functionality for this conditional expansion. The first case of the **case-id** expression checks to see if the formal argument *a* matches the pattern variable just discovered, and skips the unification if they do. Note that if this is not a top-level match, then *a* will be a temporary variable generated by the calls to **handle-pattern** in rule six. The second case checks if *p* occurs in the list of encountered variables *svar*$*$; if so, it simply extends the list of pattern parts with *p*. If the **else** case is triggered, it means that we need both a new binding for the logic variable and a pattern for the unification. In this case *p* is added to the overall list of encountered variables *svar*$*$ as well as the list of variables to be bound for this unification *evar*$*$. The list of pattern parts is also extended with *p*. Here *svar*$*$ and *evar*$*$ are kept distinct, because *svar*$*$ records all of the variables we have encountered in processing this clause, while *evar*$*$ records only those needed for the current unification, so that the **exist** clause can bind the logic variables close to their first use.

3.2.2 The *make* helper macros

One of our design principles in implementing the pattern matcher was to write several smaller macros, each with one relatively simple task to accomplish, rather than writing a few monolithic ones. This "small pieces" approach relies on the ability to compose macros as continuations to accomplish more complex actions. Therefore, we often find ourselves needing to construct continuations within continuations. The **make-clauses-cont** and **make-pattern-cont** macros help streamline the code by factoring this continuation-building behavior out into its own macros.

```
(define-syntax make-clauses-cont
  (syntax-rules ()
    ((_ args a a* p p* g* ((pr* . g**) ...) . pc*)
     (handle-pattern top a p
       (make-pattern-cont a a* p* ()
         (build-clause g*
           (handle-clauses args ((pr* . g**) ...) . pc*)))
       (a . a*) ()))))
```

The **make-clauses-cont** macro is used by **handle-clauses** when we begin processing a pattern. The continuation uses **handle-pattern** to match the first part of a pattern to the first part of the formals list. In addition to handing **handle-pattern** the list of items to work on, a fairly deeply nested chain of continuations is passed along. The outermost continuation, **make-pattern-cont**, is used to construct the continuations that build up a unification goal from the results of **handle-pattern**. Once the unification goal is built, the **build-clause** continuation is used to build the completed clause, and finally the **handle-clauses** continuation is used to begin working on the remaining clauses. This computation is responsible for driving the recursion for **handle-clauses**, the macro that both initiates the computation and finally generates the **cond**^{e} expression.

```
(define-syntax make-pattern-cont
  (syntax-rules ()
    ((_ a a* p* u* k* svar* evar* . pp*)
     (build-goal a
       (build-var evar*
         (build-clause-part a* p* u* svar* k*)) . pp*))))
```

Similar to **make-clauses-cont**, **make-pattern-cont** builds a list of nested continuations. Both **make-clauses-cont** and **build-clause-part** use **make-pattern-cont** to provide a continuation for turning the pattern built during **handle-pattern** into a proper unification goal. The outermost continuation, **build-goal**, wraps the result in a unification with its matching formal argument, *a*. It is passed a continuation of **build-var** that is responsible for turning the *evar*$*$ list into an (**ex** *evar*$*$ $\ldots$) part in the flattened list of unifications. Finally, the innermost continuation, **build-clause-part**, drives the recursion through **handle-pattern** so that the entire pattern will be processed into unifications and **ex** indicators before the completed list is passed off to **build-clause** to build the final clause.

3.2.3 The *build* macros

At various stages in the expansion, the **build** macros serve both to build some part of the final expression from parts processed through the **handle** macros, and to drive the recursion through the **handle** macros to bring the expansion to completion. The **build-goal** macro is responsible for generating unifications, when necessary. **build-var** generates **ex** indicators from the *evar*$*$ list of variables if the list is not null. The **build-clause-part** macro drives the recursion around **handle-pattern** to finish processing the pattern into a set of unification goals and **ex** indicators. **handle-pattern** uses the **build-cons** macro to rebuild pairs discovered in the pattern. Finally, the **build-clause** macro combines the flattened list of unifications and **ex** indicators with the user-supplied goals and creates a clause for use in the final **cond**^{e} expression.

```
(define-syntax build-goal
  (syntax-rules ()
    ((_ a (k* ...)) (k* ...))
    ((_ a (k* ...) p) (k* ... (≡ p a)))))
```

The two rules in **build-goal** correspond to whether **handle-pattern** has supplied a pattern to it. If _ occurs at the top level, or a formal matches a variable referenced in the same position in the match pattern, **handle-pattern** does not create a pattern; therefore, **build-goal** simply calls its continuation. Otherwise, **build-goal** calls its continuation with the unification of the provided pattern and the argument.

```
(define-syntax build-var
  (syntax-rules ()
    ((_ () (k* ...) . g*) (k* ... . g*))
    ((_ evar* (k* ...) . g*) (k* ... (ex . evar*) . g*))))
```

Likewise, **handle-pattern** may provide an empty list of discovered variables to **build-var**. Therefore, **build-var** need only create a new **ex** indicator if it receives a non-null list. Otherwise, **build-var** simply calls its continuation on the list of goals *g*$*$.

```
(define-syntax build-clause-part
  (syntax-rules ()
    ((_ () () (u* ...) svar* (k* ...) . g*) (k* ... (u* ... . g*)))
    ((_ (a . a*) (p . p*) (u* ...) svar* k* . g*)
     (handle-pattern top a p
       (make-pattern-cont a a* p* (u* ... . g*) k*) svar* ()))
    ((_ a p (u* ...) svar* k* . g*)
     (handle-pattern top a p
       (make-pattern-cont a () () (u* ... . g*) k*) svar* ()))))
```

build-clause-part receives the results of **build-goal** and **build-var**. If there are no more pattern parts to process, **build-clause-part** calls its continuation. Otherwise, **build-clause-part** calls **handle-pattern** on the next matching pattern part and argument from the *formals* list.

```
(define-syntax build-cons
  (syntax-rules ()
    ((_ (k* ...) t* p* ... pa pd)
     (k* ... t* p* ... (cons pa pd)))))
```

handle-pattern uses **build-cons** to rebuild pairs that it previously decomposed. **build-cons** simply calls its continuation, adding a *cons* expression in the final pattern to be sent to the miniKanren unifier.

```
(define-syntax build-clause
  (syntax-rules (ex)
    ((_ () (k* ...) ()) (k* ... (succeed)))
    ((_ (pg* ...) (k* ...) ()) (k* ... (pg* ...)))
    ((_ (pg* ...) k* (g* ... (ex . v*)))
     (build-clause ((exist v* pg* ...)) k* (g* ...)))
    ((_ (pg* ...) k* (g* ... g))
     (build-clause (g pg* ...) k* (g* ...)))))
```

Finally, the **build-clause** macro constructs a clause of one or more goals for use in the final **cond**^{e} expression. It processes the flattened list of unifications and **ex** indicators, along with the user-supplied goals, into a finished clause. The first rule in **build-clause** handles the case where no goals are supplied and the pattern produced no unifications. In that case, **build-clause** simply generates a miniKanren *succeed* expression, a goal that always succeeds. The second rule terminates by calling its continuation once all of the goals have been processed. The third rule recognizes an **ex** indicator and generates a new **exist** expression wrapping all of the already processed goals into a new goal. The **exist** expressions must be created in a list, since **build-clause** expects a list of clauses rather than just a single clause. Finally, in the fourth rule, any remaining goals should be unifications or user-supplied goals requiring no further processing and so are simply added to the list of processed goals for the clause.

3.2.4 The *case-id* macro

While the macros described thus far are written for the specific purpose of implementing the pattern matcher, the **case-id** macro is a more general-purpose helper macro. **case-id** determines which list of identifiers contains a match for a supplied identifier. Its syntax is like that of Scheme's standard **case** form, except that an **else** clause is required. The idea is similar to *syn-eq* [4] in that the identifier to be matched is treated as an auxiliary keyword in a generated **let-syntax**-bound macro. However, rather than taking a single list of identifiers to search for a match along with success and failure continuation macros, **case-id** takes a list of clauses, where each clause has a list of identifiers, and a result continuation macro, and requires an **else** clause for when none of the other cases succeed.

```
(define-syntax case-id
  (syntax-rules (else)
    ((_ x ((x** ...) act*) ... (else e-act))
     (letrec-syntax
       ((helper (syntax-rules (x else)
                  ((_ (else a)) a)
                  ((_ (() a) c . c*) (helper c . c*))
                  ((_ ((x . z*) a) c . c*) a)
                  ((_ ((y z* (... ...)) a) c . c*)
                   (helper ((z* (... ...)) a) c . c*)))))
       (helper ((x** ...) act*) ... (else e-act))))))
```

The macro generated by **case-id** determines when identifiers match, which allows us to avoid generating a unification, or when an identifier appears in a list of known identifiers, which indicates that no binding needs to be created for it. The real trick here is that the generated macro exploits the auxiliary keyword support of **syntax-rules** to match an element from the list of identifiers with the identifier to be matched. The auxiliary keyword is the identifier in question.

The **letrec-syntax**-bound macro *helper* searches for a case matching the original identifier *x*, expanding the **else** clause if no match is found. The first rule matches **else** and expands the associated continuation macro. The second rule identifies when the end of a list of identifiers has been encountered, and begins processing the next clause. The third rule matches the originally passed identifier and terminates by expanding the associated continuation macro. Finally, the fourth rule strips off the first identifier from the list, which failed to match in the previous clause, and recurs on the remainder of the list.

4. The variable-binding problem

We have presented a CPS macro implementation of our pattern matcher that delays creation of binding forms, allowing us to generate concise code. Unfortunately, implementing the pattern matcher with CPS macros led us to discover a subtle issue with how **case-id** determines when a variable needs to be created.

4.1 Identifier equality and binding

We can demonstrate the problem by writing a macro that expands into a $\boldsymbol{\lambda}^e$ or **match**^{e} expression. Consider the following macro, which expands into a $\boldsymbol{\lambda}^e$:

```
(define-syntax break-λe
  (syntax-rules ()
    ((_ v) (λe (x y) (((,w . ,v) ,v))))))
```

Here **break-**$\boldsymbol{\lambda}^e$ expects a user-supplied identifier for use in the generated $\boldsymbol{\lambda}^e$ expression. This simple, though admittedly contrived, example demonstrates how a CPS macro implementation of $\boldsymbol{\lambda}^e$ and **match**^{e} that uses **case-id** will not create bindings for identifiers that are free when they are symbolically equal.

To further illustrate the problem, consider some example uses of **break-**$\boldsymbol{\lambda}^e$. First, if we supply *z* as an argument to **break-**$\boldsymbol{\lambda}^e$, it expands as follows:

```
(break-λe z) ⇒
(λe (x y) (((,w . ,z) ,z))) ⇒
(λ (x y)
 (conde
   ((exist (z w)
      (≡ (cons w z) x)
      (≡ z y)))))
```

Here, $\boldsymbol{\lambda}^e$ behaves as expected; it sees both *z* and *w* as new variables that must be bound by the generated **exist** expression.

Instead, a user of **break-λ^e** may decide to use x, which coincidentally happens to be one of the variables bound by the λ^e generated by **break-λ^e**. In the expansion below, x_1 and x_2 are both symbolically x, but represent the x supplied to **break-λ^e** and the generated formal parameter x in the λ^e expression, respectively.

```
(break-λᵉ x₁) ⇒
(λᵉ (x₂ y) (((,w . ,x₁) ,x₁))) ⇒
(λ (x₂ y)
 (condᵉ
   ((exist (w x₁)
      (≡ (cons w x₁) x₂)
      (≡ x₁ y)))))
```

Here, too, identifiers are understood as unique, as we expected, and λ^e creates a binding for x_1.

Finally, the programmer may choose w as the variable to supply to **break-λ^e**. In the example below, w_1 represents the w introduced by the programmer, and w_2 the one introduced by the **break-λ^e** macro. This time, the expansion does not seem to work out so well:

```
(break-λᵉ w₁) ⇒
(λᵉ (x y) (((,w₂ . ,w₁) ,w₁))) ⇒
(λ (x y)
 (condᵉ
   ((exist (w₂)
      (≡ (cons w₂ w₁) x)
      (≡ w₁ y)))))
```

We would have liked bindings to be created for both w_1 and w_2, but since both are free and symbolically equal when **case-id** compares them, they are incorrectly understood as being equal. Although no variable capture occurred and hence hygiene is preserved, the λ^e macro does not work properly in this case, because it leaves unbound a pattern variable that should have been bound.

The issue arises as the confluence of two events. First, as we process the pattern, we delay the creation of bindings until the whole pattern has been processed, leaving free variables free. Second, **case-id** lifts the variable we are testing into an auxiliary keyword in the helper macro to compare it with the list of identifiers. The comparison between x and the identifiers from each list will succeed when both have the same binding or when both are free and they are symbolically equal [5, 7].

In the first example, both w and z are free, but are not symbolically equal. In the second example, x_1 and x_2 are symbolically equal, but one is bound while the other is free, so the comparison fails, as we would expect. It is only in the final case, where both w identifiers are free and symbolically equal, that the problem exhibits itself.

4.2 With great power comes great responsibility

As we have seen, CPS macros provide a powerful mechanism for controlling the order of macro expansion. However, the variable-binding problem limits our ability to use CPS macros to generate bindings selectively based on a running list of identifiers. In order to avoid unintentionally conflating variables, we must bind identifiers as soon as we encounter them, rather than delaying binding until the invocation of a final continuation.

This limitation suggests that CPS macro writers must take particular care to avoid the accidental conflation of free, symbolically-equal identifiers that are introduced from different places in the source. Hygienic macro expansion does not help us here, since the problem is not inappropriate variable capture; rather, it is that variables that should be bound are left unbound. Avoiding accidental conflation of pattern variables therefore becomes the programmer's responsibility.

5. Workarounds

In this section, we present two solutions to the variable-binding issue demonstrated in the previous section. Our first solution uses the **syntax-case** macro system and the *bound-identifier=?* predicate to perform the comparison we actually intend. Second, we present a **syntax-rules**-based solution using eager binding by foregoing certain uses of CPS in favor of a more traditional approach.

5.1 *case-id* with *syntax-case* and *bound-identifier=?*

If we restrict ourselves to CPS macros written using the **syntax-rules** macro system, there is, unfortunately, no easy change we can make that will resolve the variable-binding issue. Fundamentally, **syntax-rules** only provides us with a way to perform what is essentially a *free-identifier=?* check, by generating a macro that has the identifier we wish to match as an auxiliary keyword.

However, the **syntax-case** macro system gives us the ability to compare identifiers according to their *intended use* by employing the *bound-identifier=?* predicate. *bound-identifier=?* takes two identifier arguments and returns #t only if a binding for one identifier would capture the other. Effectively, two identifiers will be *bound-identifier=?* only if they were introduced by the same transformer or within the same macro [7, 2]. In fact, this is the very comparison we would prefer for **case-id**.

We can implement **case-id** straightforwardly with **syntax-case** by using *bound-identifier=?* in a fender, as follows:

```
(define-syntax case-id
  (λ (exp)
    (syntax-case exp (else)
      ((_ x (else e-act)) #'e-act)
      ((_ x ((y x* ...) act) ((x** ...) act*) ... (else e-act))
       (bound-identifier=? #'x #'y)
       #'act)
      ((_ x ((y x* ...) act) ((x** ...) act*) ... (else e-act))
       #'(case-id x
           ((x* ...) act) ((x** ...) act*) ... (else e-act)))
      ((_ x (() act) ((x** ...) act*) ... (else e-act))
       #'(case-id x ((x** ...) act*) ... (else e-act))))))
```

The interface to **case-id** remains the same, and the rest of the pattern matcher implementation need not be changed. In this version of **case-id**, the first clause matches when only the **else** case is left. The second clause extracts an identifier from the list and uses the *bound-identifier=?* check to compare the identifiers. If the comparison succeeds, that case's action is used. The third clause extracts the identifier and throws it away to continue processing the current list, since we have already verified in the previous clause that x and y are not *bound-identifier=?*. The final clause matches when we have exhausted the list of identifiers to be matched for the current case, and so we proceed to the next case from the call to **case-id**.

Using this implementation of **case-id**, when we expand the third **break-λ^e** expression from the previous section, we get

```
(break-λᵉ w) ⇒
(λᵉ (x y) (((,w₂ . ,w₁) ,w₁))) ⇒
(λ (x y)
 (condᵉ
   ((exist (w₂ w₁)
      (≡ (cons w₂ w₁) x)
      (≡ w₁ y)))))
```

with both w_1 and w_2 being bound by the surrounding **exist** expression. This workaround has the advantages of producing very clean miniKanren source and allowing us to keep most of our implementation unchanged, but it does force us to use **syntax-case**.

5.2 Using eager binding with *syntax-rules*

While we can fix the variable-binding issue in our pattern matcher by implementing **case-id** with **syntax-case**, we may prefer to stick with a **syntax-rules**-based implementation. **syntax-rules** offers us the simplicity of pattern matching and rewriting without having to worry about the potentially more complex **syntax-case** macro system or the details of how *bound-identifier=?* works. Here we present a **syntax-rules** solution to the variable-binding issue that works by eagerly binding new identifiers as they are encountered.

Unlike the **syntax-case** solution, which resolved the issue by performing a different kind of comparison in **case-id**, the eager binding approach ensures that our list of seen variables never contains free identifiers. Since we never compare two free identifiers, we no longer need to worry that two symbolically equal identifiers will be conflated, and the **syntax-rules** version of **case-id** can remain unchanged.

This approach is not without complications of its own, since $\boldsymbol{\lambda}^{e}$ and **match**^{e} must expand into **cond**^{e} and **exist**, which impose their own limitations on the expressions in their clauses. The challenge arises because **cond**^{e} expects a set of clauses in which each clause is a list of one or more goals and **exist** expects a list of bindings followed by one or more goals. Since the helpers for $\boldsymbol{\lambda}^{e}$ and **match**^{e} will expand within the context of **cond**^{e} and **exist**, they must expand into valid goals. Part of the difficulty arises from the fact that **cond**^{e} and **exist** perform a monadic transform, which $\boldsymbol{\lambda}^{e}$ and **match**^{e} must be careful not to interfere with.

Unfortunately, these restrictions mean that the eager-binding versions of $\boldsymbol{\lambda}^{e}$ and **match**^{e} cannot generate quite as clean miniKanren code as the original CPS macro implementation. Returning to our *append* example, the fixed version of $\boldsymbol{\lambda}^{e}$ expands to the slightly more verbose:

```
(λ (x y z)
  (cond^e
    ((exist () (≡ () x) (≡ y z)))
    ((exist (a)
       (exist (d)
         (exist ()
           (≡ (cons a d) x)
           (exist (r)
             (exist ()
               (≡ (cons a r) z)
               (append d y r)))))))))
```

Here, the **exist** expressions that bind no variables each enclose more than one goal. Grouping multiple goals inside an **exist** allows them to appear in a position where only one goal is allowed, in much the same way Scheme's **begin** can group multiple expressions into a single expression.

The **break-**$\boldsymbol{\lambda}^{e}$ macro now works correctly in all of the previously shown examples. In particular, (**break-**$\boldsymbol{\lambda}^{e}$ *w*) now expands as follows:

```
(break-λ^e w) ⇒
(λ^e (x y) (((,w2 . ,w1) ,w1))) ⇒
(λ (x y)
  (cond^e
    ((exist (w2)
       (exist (w1)
         (exist ()
           (≡ (cons w2 w1) x)
           (≡ w1 y)))))))
```

We have taken some liberties in this example, since the **exist** macro would need to be in scope in order for it to work properly, but the full expansion of **exist** would needlessly complicate the example.

Even though this version of the pattern matcher uses the original version of **case-id**, it now correctly identifies the two *w* variables as distinct, since the binding for w_2 is created by **exist** before the next section of the pattern is expanded.

In order to implement the eager binding approach, we must alter the part of our pattern matcher that identifies variables to be bound. We accomplish this by refactoring **handle-pattern** into two macros: **do-pattern**, which binds any necessary variables, and a simplified **handle-pattern**, which builds the processed version of the pattern for use in a unification. **handle-pattern** remains a CPS macro and has been simplified in accordance with its reduced mission. As before, the **do-pattern-opt** macro prevents recognizably unnecessary unifications from being generated.

The interface to $\boldsymbol{\lambda}^{e}$ and **match**^{e} does not change, and **handle-clauses** has been rewritten to support expanding into the **cond**^{e} in place.

```
(define-syntax handle-clauses
  (syntax-rules ()
    ((_ (a* ...) (c c* ...))
     (cond^e ((do-clause (a* ...) (a* ...) c))
             ((do-clause (a* ...) (a* ...) c*)) ...))
    ((_ (a* ... . r) (c c* ...))
     (cond^e ((do-clause (a* ... r) (a* ... . r) c))
             ((do-clause (a* ... r) (a* ... . r) c*)) ...))
    ((_ a (c c* ...))
     (cond^e ((do-clause (a) a c))
             ((do-clause (a) a c*)) ...))))
```

handle-clauses is responsible for generating the **cond**^{e} expression, passing first a list of named variables, then the original argument list, and finally the clause to be processed to **do-clause**.

```
(define-syntax do-clause
  (syntax-rules ()
    ((_ svar* () (() . g*)) (exist-helper () . g*))
    ((_ svar* (a . a*) ((p . p*) . g*))
     (do-pattern-opt svar* a p a* p* . g*))
    ((_ svar* a (p . g*))
     (do-pattern-opt svar* a p () () . g*))))
```

The **do-clause** macro processes each formal from the argument list with the corresponding part of the pattern, relying on **do-pattern-opt** to generate the variable bindings and unifications for each clause. Finally, it expands into the list of user-supplied goals.

```
(define-syntax do-pattern-opt
  (syntax-rules (unquote __)
    ((_ svar* a (unquote p) a* p* . g*)
     (case-id p
       ((a) (do-clause svar* a* (p* . g*)))
       (else (do-pattern svar* a ,p () ,p a* p* . g*))))
    ((_ svar* a __ a* p* . g*) (do-clause svar* a* (p* . g*)))
    ((_ svar* a p a* p* . g*)
     (do-pattern svar* a p () p a* p* . g*))))
```

While we could generate unifications for each part of the pattern, we would prefer to recognize unnecessary unifications and not generate them, as in the original implementation. **do-pattern-opt** ensures that unifications are not generated when a __ is encountered at the top level or when a logic variable is being matched with itself. In all other cases, **do-pattern-opt** calls **do-pattern**, which generates the necessary **exist** bindings or unifications.

```
(define-syntax do-pattern
  (syntax-rules (quote unquote)
    ((_ svar* a () () op () ())
     (do-clause svar* ()
       (() (handle-pattern op (handle-pattern-cont a) ()))))
    ((_ svar* a () () op a* p* . g*)
     (exist ()
       (handle-pattern op (handle-pattern-cont a) ())
       (do-clause svar* a* (p* . g*))))
    ((_ svar* a (unquote p) r op a* p* . g*)
     (case-id p
       (svar* (do-pattern svar* a r () op a* p* . g*))
       (else (exist (p)
               (do-pattern (p . svar*) a r () op a* p* . g*)))))
    ((_ svar* a (quote p) r op a* p* . g*)
     (do-pattern svar* a r () op a* p* . g*))
    ((_ svar* a (pa . pd) () op a* p* . g*)
     (do-pattern svar* a pa pd op a* p* . g*))
    ((_ svar* a (pa . pd) r op a* p* . g*)
     (do-pattern svar* a pa (pd . r) op a* p* . g*))
    ((_ svar* a p r op a* p* . g*)
     (do-pattern svar* a r () op a* p* . g*))))
```

In **do-pattern**, the **unquote** rule uses **case-id** to determine if the logic variable *p* has been encountered. If not, a binding is generated and *p* is added to the list of known variables. The other rules are responsible for traversing the full pattern. Some optimization is also performed by **do-pattern**: it avoids generating unnecessary *succeed* goals by recognizing when it has reached the end of the pattern and there are no user-supplied goals, in which case it treats the final pattern unification as if it were a user-supplied goal.

Once bindings for all new variables have been created, the original pattern is passed off to **handle-pattern**, and the rest of the pattern and formal parameters are passed back to the **do-clause** macro to continue processing.

```
(define-syntax handle-pattern
  (syntax-rules (quote unquote __)
    ((_ () (k* ...) t* p* ...) (k* ... t* p* ... '()))
    ((_ __ (k* ...) t* p* ...) (k* ... (t . t*) p* ... t))
    ((_ (unquote p) (k* ...) t* p* ...) (k* ... t* p* ... p))
    ((_ (quote p) (k* ...) t* p* ...) (k* ... t* p* ... 'p))
    ((_ (pa . pd) k t* p* ...)
     (handle-pattern pa
       (handle-pattern pd (build-cons k)) t* p* ...))
    ((_ p (k* ...) t* p* ...) (k* ... t* p* ... 'p))))
```

The revised **handle-pattern** macro no longer needs to know about the argument being processed, nor does it need to know whether it is called at the top level of a pattern or within a pattern, so the first two arguments have been removed, simplifying the macro quite a bit. However, **handle-pattern** still needs a continuation, since it proceeds recursively through the pattern.

handle-pattern also adds new bindings for the temporary variables needed by the __ matches. This is safe because we will always need these temporary variables and because we no longer use **case-id** to guide our decisions about which variables need to be bound. The sixth rule of **handle-pattern**, previously the seventh rule, still uses **build-cons** in order to reconstruct a matched pair.

In addition to the updated **handle-pattern**, we need a new continuation for it to call, **handle-pattern-cont**.

```
(define-syntax handle-pattern-cont
  (syntax-rules ()
    ((_ v t* p) (exist-helper t* (≡ p v)))))
```

handle-pattern-cont simply generates a unification, wrapping it with an **exist** for any temporaries that need to be bound. Rather than a standard **exist** expression, we use the following **exist-helper** in order to avoid generating unnecessary **exist** expressions when possible:

```
(define-syntax exist-helper
  (syntax-rules ()
    ((_ ()) succeed)
    ((_ () g) g)
    ((_ t* g* ...) (exist t* g* ...))))
```

exist-helper generates the *succeed* goal when supplied an empty bindings list and no goals, or the provided goal when supplied an empty bindings list and a single goal. Otherwise, it generates a normal **exist** expression.

6. Conclusion

CPS macros provide a powerful mechanism for controlling the order of macro expansion, but care must be taken when using this technique with conditional expansion. In particular, we must use caution when using **syntax-rules** with the auxiliary keyword trick to perform variable comparisons, or we may end up treating two free identifiers that are symbolically equal as the same, even if they will not be equal when they are bound. However, we can work around this limitation either by using **syntax-case** for performing the comparisons with *bound-identifier=?*, or by using eager binding to ensure that no two free variables will ever be compared. We hope that these techniques will prove useful for macro implementors who find themselves faced with a similar issue in using CPS macros. An interesting area of further investigation in this regard would be to look at ways to bring the ability to perform *bound-identifier=?* comparisons to **syntax-rules**. Already some implementations of **syntax-rules**, such as the one provided with Chez Scheme [2], provide a fender syntax similar to that of **syntax-case** which allows the use of such techniques, although this has not yet found its way into the standard.

Acknowledgments

The authors wish to express their thanks to the anonymous reviewers, whose thoughtful comments and suggestions have improved this paper.

References

[1] W. E. Byrd and D. P. Friedman. From variadic functions to variadic relations. In *Proceedings of the 2006 Scheme and Functional Programming Workshop, University of Chicago Technical Report TR-2006-06, 2006*, pages 105–117, 2006.

[2] R. K. Dybvig. *Chez Scheme Version 7 User's Guide*. Cadence Research Systems, 2005.

[3] D. P. Friedman, W. E. Byrd, and O. Kiselyov. *The Reasoned Schemer*. The MIT Press, 2005.

[4] E. Hilsdale and D. P. Friedman. Writing macros in continuation-passing style. In *Scheme and Functional Programming 2000*, page 53, 2000.

[5] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, Sept. 1998.

[6] O. Kiselyov. Macros that compose: Systematic macro programming. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 202–217, London, UK, 2002. Springer-Verlag.

[7] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten (eds.). Revised^6 report on the algorithmic language Scheme, September 2007.