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

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

Indiana University, Bloomington, IN 47405

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

Abstract

In this paper we present two implementations of the pattern-matching macros λ^e and **match**^{e} for αKanren. The first implementation generates clean code, but our use of CPS-macros in its implementation leads to a problem in determining when a new binding needs to be introduced for a pattern variable. This problem stems from our delayed creation of bindings, where the comparison of identifiers is done first and then binders created in a later step. This may lead to an issue when macros generating λ^e or **match**^{e} expressions may appear to break hygiene because the CPS-macros incorrectly identify two identifiers as being the same. The second implementation addresses these concerns by using more traditional macros, generating bindings as new variables are encountered.

1. Introduction

We present two implementations of the λ^e and **match**^{e} macros. The first implementation is written using macros in continuation-passing style (CPS) [**?**]. While this implementation generates nice clean αKanren code, we encounter an issue related to comparing identifiers that may lead to programs that appear to break hygiene, suggesting that some care must be taken when writing macros in CPS style. In particular our implementation delayed the creation of bindings until the final step, which led to problems where two identifiers in a pattern that share the same symbolic representation, but were created in different sections of the program may be identified as being equal, since both remain free during the macro expansion process. The second implementation fixes the issue with the first implementation by using a more traditional macro style, introducing bindings for new variables as it proceeds. No knowledge of αKanren is required for our discussion of the CPS-macro issue we encountered, which is described in section 4.

The Prolog family of logic programming languages has long used patterns to define logic rules. The first appearance of a variable in these patterns leads to a new logic variable being created in the global environment used by Prolog. αKanren, a nominal logic programming language [**?**] similar to αProlog, on the other hand, provides no facility for pattern matching and requires logic variables and noms be explicitly introduced with **exist** and **fresh**, respectively. While this does not limit αKanren's expressiveness, programs written in αKanren are often longer when compared with αProlog equivalents.

The λ^e and **match**^{e} macros described in this paper allow αKanren[1] to take advantage of pattern matching with automatic variable creation, without changing the semantics of the language. The clauses of λ^e and **match**^{e} represent a disjunction and are expanded into a **cond**^{e} expression, αKanren's disjunction operator. Each pattern is then expanded into a set of variable bindings and unifications.

In the next section we provide a brief refresher for αKanren, followed by a description of λ^e and **match**^{e}, along with illustrative examples. The following two sections present the two implementations of λ^e and **match**^{e}. Finally future work and conclusion sections discuss next steps for the λ^e and **match**^{e} macros and wrap up the paper.

2. An αKanren Refresher[2]

αKanren is an embedding of nominal logic programming in Scheme. It extends the Scheme language with a term constructor $\bowtie$ (pronounced "tie") and five operators: $\equiv$, #, **exist**[3], **fresh**, and **cond**^{e}.

$\equiv$ unifies two terms using nominal unification. **exist** and **fresh**, which are syntactically similar to Scheme's **lambda** and whose bodies are conjoined, are used to introduce new lexical variables; those introduced by **exist** bind logic (or unification) variables, while those introduced by **fresh** bind *noms* (also called "names" or "atoms" in nominal logic). A nom unifies only with a logic variable or with itself. noms are often used to represent variable names. # is a freshness constraint: (# *a t*) asserts that the nom *a* does *not* occur free in *t*. $\bowtie$ is a term constructor: ($\bowtie$ *a t*) creates a term in which all free occurrences of the nom *a* in *t* are considered bound. Thus (# *a* ($\bowtie$ *a t*)) always succeeds.

cond^e, which is syntactically similar to **cond**, expresses a disjunction of clauses. Each clause may contain arbitrarily many conjoined goals. Unlike **cond**, every clause of a **cond**^{e} will return a stream of results, as long as the conjoined goals in the clause all succeed.

run provides an interface between Scheme and αKanren; it allows the user to limit the number of answers returned, and to specify a logic variable whose value should be *reified* to obtain answers. Reification is the process of replacing distinct logic variables in a

2009 Workshop on Scheme and Functional Programming

[1] As well as miniKanren [**?**, **?**]

[2] A similar introduction to αKanren originally appeared in the αLeanTAP paper [**?**].

[3] The name **exist** is chosen to avoid conflict with R^{6}RS Scheme's [**?**] exists.

term with unique names. The first such variable to be found is represented by the symbol $\_0$, the second by $\_1$, and so on. For example:

```
(run^5 (q)
  (exist (x y z)
    (cond^e                                    ((3 2 2)
      ((≡ x 3) (≡ y 2) (≡ z y))        ⇒       (_0 _0 _0)
      ((≡ x y) (≡ y z))                        (_0 _1 _0))
      ((≡ x z)))
    (≡ `(,x ,y ,z)^4 q)))
```

This **run** expression has three answers, each corresponding to one line of the **cond**^{e}. In the first answer, all three variables have been instantiated to ground values. In the second, the three variables have been unified with one another, so they have the same reified value. In the third, x and z share the same reified value, which is distinct from that of y.

Nominal unification equates α-equivalent binders:

```
(run^1 (q) (fresh (a b) (≡ (⋈ a a) (⋈ b b)))) ⇒ (_0)
```

Although the noms a and b are distinct and would therefore fail to unify, this **run** expression succeeds. Like the terms $\lambda a.a$ and $\lambda b.b$, the terms $(\bowtie a\ a)$ and $(\bowtie b\ b)$ bind in the same way and are thus α-equivalent.

αKanren is based on αProlog [**?**], which implements the nominal unification of Urban, Pitts, and Gabbay [**?**], and miniKanren, an earlier logic programming language [**?**, **?**]. For a more complete description of αKanren, see Byrd and Friedman [**?**].

3. Using λ^e and match^e

The λ^e and **match**^{e} macros provide pattern matching with automatic variable creation functionality. The central difference between λ^e and **match**^{e} is that λ^e generates a λ expression with an embedded **cond**^{e}, while **match**^{e} **let**-binds its expression to a temporary, and then matches the temporary with the pattern.

3.1 Writing the *append* Relation

Appending two lists is a common operation in logical and functional programming languages and Prolog has a very concise definition for this operation.

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

Using λ^e the αKanren version can be expressed almost as succinctly. In particular the two clauses correspond exactly to the two Prolog rules used to define append.

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

As in the Prolog version of this code, λ^e unifies x with () and y with z in the first clause and generates new logic variables a, d, and r along with unifications for `(,a . ,d)` with x, and `(,a . ,r)` with z. The actual expansion is left until the next two sections where we discuss the two implementations.

In append^o, __ is used to indicate 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. Instead of __ here, we can also use ,y. λ^e recognizes a variable being matched against itself and avoids adding the unnecessary unification.

[4] Here, backquote and comma are used to build a list of logic variables: the expression `(,x ,y ,z)` is equivalent to `[X, Y, Z]` in Prolog. Similarly, the expression `(,x . ,y)` constructs a pair, and is equivalent to `[X|Y]` in Prolog.

3.2 λ^e and match^e Description

The basic format for λ^e is:

```
(λ^e formals
  (pattern-1 goal-1 ...)
  (pattern-2 goal-2 ...)
  ...)
```

where the *formals* maybe any valid λ formals expression, including those for variable length argument lists.

Each clause of the λ^e expression will be expanded into a clause of a **cond**^{e} expression, αKanren's disjunction operator. In each clause the pattern will be matched against the formals supplied in λ^e and expanded into a set of variable bindings and unifications, including any user specified goals within the scope of the variables created by the pattern. It is important to note that variables not named in the formals list or the pattern, but used in user specified goals must still be explicitly introduced by the user.

The pattern matcher recognizes the following forms:

() The null list.

__ Similar to Scheme's _, this represents a position where an expression is expected, but we do not care what the expression actually is.

,*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 λ^e a new logic variable will be created.

,@*a* A nom a. Similar to the logic variable case, it will be created if it does not exist.

'e Preserves the expression, e. This is provided as an escape for the other special forms where the exact contents should be preserved. For instance, if we wish to match __ we could use '__ in our pattern to remove the special meaning of __.

sym Where *sym* is any Scheme symbol, will be preserved in the unification patterns as a symbol.

(*a* . *d*) Arbitrarily nested pairs and lists are also allowed.

(⋈ *n e*) A ⋈ constructor, binding the nom n in the expression e.

The ⋈ form is included to provide a more natural way for programmers to include αKanren's binding construct in the patterns to be matched.

Similar to αProlog, patterns in λ^e and **match**^{e} use different syntax to indicate the difference between a logic variable and a nom. In αProlog, this is done using capitalization of variables, inherently segmenting the name space between logic variables and noms. In λ^e and **match**^{e} the **unquote** and **unquote-splicing** keywords are used to differentiate, but the variable names themselves both share the same name variable name space within Scheme. This means, for instance, that we cannot create both a logic variable x and a nom x in the same lexical scope.

match^e has the following format:

```
(match^e expr
  (pattern-1 goal-1 ...)
  (pattern-2 goal-2 ...)
  ...)
```

where *expr* is any Scheme expression and the patterns for each clause use the same pattern format as λ^e. Using **match**^{e} the pattern is matched against the whole expression, rather then the individual formal parameters of λ^e.

3.3 A More Involved Example

append^o provides a nice introduction to λ^e, but a more involved example will help further illustrate the use of λ^e and **match**^{e}. Here we present typ^o, a type-inferencer for the simply-typed λ-calculus,

from [**?**] to illustrate the use of noms and tie. The type environment is represented using a simple association list. The function to lookup items in the environment can be easily implemented in αKanren as follows:

```
(define lookup^o
  (λ^e (x tx g)
    ((__ __ ((,x . ,tx) . ,d)))
    ((__ __ ((,x̂ . ,tx̂) . ,d))
     (# x x̂)
     (lookup^o x tx d))))
```

We can then implement the type inferencer, typ^o as follows:

```
(define typ^o
  (λ^e (g e te)
    ((__ (var ,x) __) (lookup^o x te g))
    ((__ (app ,rator ,rand) __)
     (exist (trator trand)
       (≡ '(→ ,trand ,te) ,trator)
       (typ^o g rator trator)
       (typ^o g rand trand)))
    ((__ (lam (⋈ ,@b ,ê)) (→ ,trand ,tê))
     (exist (ĝ)
       (# b g)
       (≡ '((,b . ,trand) . ,g) ĝ)
       (typ^o ĝ ê tê)))))
```

Here, *trator* and *trand* in the second clause, and $\hat{g}$ must be introduced by the programmer, since they are not introduced in the program source, but are needed by the goals that specify the recurrence in typ^o.

The first typ^o example shows that $\lambda c.\lambda d.c$ has type $(\alpha \to (\beta \to \alpha))$.

```
(run* (q)
 (fresh (c d)
   (typ^o '() '(lam ,(⋈ c '(lam ,(⋈ d '(var ,c))))) q)))
⇒

((→ _0 (→ _1 _0)))
```

The second example shows that self-application does not type check, since the nominal unifier uses the occurs check [**?**].

```
(run* (q)
 (fresh (c)
   (typ^o '() '(lam ,(⋈ c '(app (var ,c) (var ,c)))) q)))
⇒

()
```

The final example is the most interesting, since it searches for terms that inhabit the type $(\mathsf{int} \to \mathsf{int})$.

```
(run^2 (q) (typ^o '() q '(→ int int))) ⇒

((lam (tie a_0 (var a_0)))
 (lam (tie a_0 (app (lam (tie a_1 (var a_1))) (var a_0)))))
```

The first two terms found are $\lambda b.b$ and $\lambda b.(\lambda a.a)b$. αKanren reifies noms as a_0, a_1, etc. In the two terms above a_0 and a_1 are reified noms.

While the typ^o and lookup^o relations are fairly concise, in both cases the matches are done primarily against a single argument to the λ^e expression rather then all of the arguments. Instead of using λ^e then, we might choose to use **match**^{e}.

```
(define typ^o
  (λ (g e te)
    (match^e e
      ((var ,x) (lookup x te g))
      ((app ,rator ,rand)
       (exist (trator trand)
         (≡ '(→ ,trand ,te) ,trator)
         (typ^o g rator trator)
         (typ^o g rand trand)))
      ((lam (⋈ ,@b ,ê))
       (exist (ĝ trand tê)
         (≡ '(→ ,trand ,tê) te)
         (# b g)
         (≡ '((,b . ,trand) . ,g) ĝ)
         (typ^o ĝ ê tê))))))

(define lookup^o
  (λ (x tx g)
    (match^e g
      ((,a . ,d) (≡ '(,x . ,tx) a))
      (((,x̂ . ,tx̂) . ,d)
       (# x x̂)
       (lookup^o x tx d)))))
```

Both implementations of λ^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 ⋈ and __ auxiliary keywords are exported with the library.

4. Implementing λ^e and match^e

In implementing a macro for the λ^e and **match**^{e} pattern language we had a few goals in mind. The primary goal is providing the convenience of the pattern language, but we would also like to generate code that will perform at least as well as if the generated code had been written by a human. As a rule of thumb this means that we would like to bring new logic variables and noms into scope only when we know they will be used and we would like to create as few empty **exist** or **fresh** constructs as possible.

In order to understand a little better why these are our guiding performance rules, we need to know a little bit more about the implementation of αKanren. While logic variables and noms are represented by the programmer as normal variables within Scheme, behind the scenes a substitution is used to keep track what logic variables are bound to. This information is then used by the unifier in αKanren to determine when matches succeed and fail. The unifier may also extend this substitution as logic variables are unified with other objects, including noms, ground terms, and other logic variables, in the system. While this substitution must exist throughout an αKanren program, the programmer is not responsible for explicitly naming and managing the substitution, instead the various forms in αKanren, such as **cond**^{e}, **exist**, and **fresh** are responsible for creating a monadic transform on the code contained within that threads the substitution through the goals contained within these forms.

In our case **exist** and **fresh** are of particular import because these are the forms we will be generating for each logic variable and nom created. **exist** and **fresh** are both binding forms in αKanrenand effectively generate a let that binds the variables named in the **exist** or **fresh** to a new logic variable or nom, respectively. While each logic variable created incurs only a small cost, we would prefer to create only those logic variables we know will be needed. In order to provide a more concrete example of this, consider the following **exist** expression.

```
(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 since ($\equiv$ x y) will fail because ($\equiv$ '(,x . ,y) '(a . b)) binds x to 'a and y to 'b. We could write this with tighter lexical scope for our variables as:

(**exist** (x y) ($\equiv$ '(,x . ,y) '(a . b)) ($\equiv$ x y) (**exist** (z) ($\equiv$ z 'c)))

This rule helps **exist** clauses to fail more quickly which cuts off αKanren's search for solutions which can be important in long searches.

The other rule — is this whole description worth while?

4.1 λ^e and match^e

4.2 Determining When to Add Bindings

One of the key issues in implementing λ^e and **match**^{e} is determining when a variable reference encountered in the pattern is a new variable requiring a binding not an existing variable. Since **syntax-rules** does not directly provide a method for testing the equality of two identifiers, a macro like *syn-eq* [**?**] provides a way for a list of identifiers to be maintained while the macro is being expanded and used to direct computation based on the appearance of an identifier in the list of already encountered variable identifiers.

In order to use **syntax-rules** to perform this comparison a macro-generating macro is used, where the identifier being searched for is made into an auxiliary keyword in the generated macro.

Here is the implementation of *syn-eq* (renamed as **mem-id**):

(**define-syntax mem-id** (**syntax-rules** () ((_ *id* ($x*$ ...) *conseq altern*) (**letrec-syntax** ((*helper* (**syntax-rules** (**id**) ((_ () *c a*) *a*) ((_ (**id** . $z*$) *c a*) *c*) ((_ (*y* . $z*$) *c a*) (*helper* $z*$ *c a*))))) (*helper* ($x*$ ...) *conseq altern*)))))

Here the **letrec-syntax** bound macro *helper* recurs until either the keyword, **id**, originally passed in as the first argument to **mem-id** is encountered, or until the end of the list, expanding into the original *conseq* or *altern* code depending on which case succeeds. For λ^e and **match**^{e} the *conseq* will not create the variable, while the *altern* will, since it will only happen when **id** is not found in the list.

4.3 Controlling Expansion Order

In general CPS-macros [**?**] seem to provide a good fit for a pattern matching macro where parts of the pattern must be reconstructed for the use during unification and bindings for variables must be generated outside these unifications.

Returning to the append^o example above, the expanded **cond**^{e} expression might be:

```
(define appendo
  (λ (x y z)
   (conde
      ((≡ () x) (≡ y z))
      ((exist (a d)
         (≡ (cons a d) x)
         (exist (r)
           (≡ (cons a r) z)
           (appendo d y r)))))))
```

In order to expand into this **cond**^{e} expression, each clause of the λ^e has its pattern processed, looking for variables that must be introduced and building a list of unifications, keeping track of where variables must be introduced. Since the pattern is processed from left to right and the variables encountered in the left most part of the clause should be scoped around the goals generated to the right, we cannot expand into **exist** and **fresh** expressions in place. Instead a place holder is put into the generated list of unifications for the variable creation and the continuation macro will take care of building the **exist** and **fresh** expressions.

To give an example of what this looks like, here is the second clause of append^o at this intermediate stage:

(**ex** *a d*) ($\equiv$ (*cons a d*) *x*) (**ex** *r*) ($\equiv$ (*cons a r*) *z*)

Here the (**ex** *a d*) and (**ex** *r*) are place holders for **exist** expressions. The scoping for the **exist** will include all of the items to the right of it in the list, so the list is processed from right to left, with any user supplied goals serving as the left most part of the expression. The final expression will be:

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

Note that the **exist** expressions are placed such that each generated variable has the smallest possible scope. This is important because we would like each clause in the **cond**^{e} expression to fail as quickly as possible. Here if the ($\equiv$ (*cons a d*) *x*) unification fails, *r* will never be created. For more details on this implementation we have included the source for the CPS-macro implemnetation in appendix A.

4.4 Identifier Equality and Binding

While this technique of delaying creation of the binding forms allows us to generate concise code, it has a subtle issue in how **mem-id** determines when a variable needs to be created. The problem can be detected when we write a macro that expands into a λ^e or **match**^{e}.

Consider the following macro that expands into a λ^e expression.

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

Here **break-**λ^e expects a variable name to be supplied to the macro that will be used in the generated λ^e. This simple, though admittedly contrived example, demonstrates how a CPS-macro implementation of λ^e and **match**^{e} that uses the **mem-id** trick, produces code in conflict with our intuitions about Scheme's hygienic macros.

To demonstrate the problem and hone in on what is going on, three examples are presented here.

First, if *d* is supplied as to **break-**λ^e it expands as follows:

```
(break-λ^e d) ⇒
(λ^e (x y)
   (((,a . ,d) ,d))) ⇒
(λ (x y)
 (cond^e
    ((exist (a d)
       (≡ (cons a d) x)
       (≡ x y)))))
```

As expected, λ^e sees both *a* and *d* as new variables that need to have a binding created in the **exist** expression.

Instead, a user of **break-**λ^e may decide to use *x*, which just happens to be one of the variables bound by 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, respectively.

```
(break-λ^e x_1) ⇒
(λ^e (x_2 y)
   (((,a . ,x_1) ,x_1))) ⇒
(λ (x_2 y)
 (cond^e
    ((exist (a x_1)
       (≡ (cons a x_1) x_2)
       (≡ x_1 y)))))
```

Here too, hygiene is preserved and a binding x_1 is created. Finally a programmer may choose *a* as the variable to supply to **break-**λ^e. Here the expansion does not seem to work out so well:

```
(break-λ^e a) ⇒
(λ^e (x y)
   (((,a . ,a) ,a))) ⇒
(λ (x y)
 (cond^e
    ((exist (a)
       (≡ (cons a a) x_1)
       (≡ a y)))))
```

Although we may have expected the hygienic macro system to create a binding for two distinct *a* variables as it did with *x* in the second example, instead only one binding is created. In the final resulting expression, the *a* supplied to **break-**λ^e will remain unbound, resulting in an unexpected result.

The issue arises as the confluence of two events. First, as we processed the pattern binding creation is delayed until the whole pattern has been processed. Second, **mem-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 **id** and each identifier from the list will succeed when both have the same binding or when both are free and they are symbolically equal [**?**, **?**].

In the first example, both *a* and *d* 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 not, so the comparison fails as we would hope. It is only in the final case both *a* identifiers are free and symbolically equal that the problem exhibits itself.

4.5 Caution Must be used with CPS-macros

The conclusion to draw from this experience is that while CPS-macros provide a mechanism for controlling the order a macro expands in, if it is used in conjunction with generating bindings selectively based on a running list of identifiers, these identifiers must be bound as they are encountered rather then delaying this to a final continuation.

The larger implications of this, is that CPS-macros may not be as powerful a tool as originally thought. In particular, it shows that macro writers using this technique must take particular care to ensure that any new variables bound through this method are written in such a way that they avoid accidental capture and ensure that the expanded code will not contain unbound variables, such as our first implementation of λ^e and **match**^{e} encountered.

Currently, we know of no way to work around this using only **syntax-rules** macros without imposing the responsibility for ensuring proper variable handling on the user.

5. Fixing the λ^e and match^e Implementation

The CPS-macro implementation of λ^e and **match**^{e} encountered a problem without a readily available solution, so the second implementation for the most part restricts itself to more straightforward macro writing techniques. This approach is not without complications of its own, since λ^e and **match**^{e} must expand into **cond**^{e}, **exist**, and **fresh**, which impose their own rules about the expressions in their body. The challenge arises because **cond**^{e} expects a set of clauses, where each clause is a list of one or more goals and **exist** and **fresh** expect a list of binders followed by one or more goals. Since the helpers for the λ^e and **match**^{e} will expand within the context of **cond**^{e}, **exist**, or **fresh**, they must expand into valid goals.

Part of this arises from the fact that the current implementation of **cond**^{e}, **exist**, and **fresh** perform a monadic transform, that λ^e and **match**^{e} must be careful not to interfere with.

Unfortunately these restrictions mean that λ^e and **match**^{e} cannot generate quite as clean αKanren code as the original CPS-macro implementation. Looking again at our append^o example, using this correct version of the macro, we generate 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^o d y r)))))))))
```

Here the **exist** clauses that bind no variables are needed to preserve the property that only a single goal appear in each position within the body of the **cond**^{e} clauses as a sort of **begin** expression.

The **break-**λ^e macro now works correctly in all cases, though. In particular the (**break-**λ^e *a*) example now expands into[5]:

```
(break-λ^e a_1) ⇒
(λ^e (x y)
   (((,a_2 . ,a_1) ,a_1))) ⇒
(λ (x y)
  (cond^e
    ((exist (a_2)
       (exist (a_1)
         (exist ()
           (≡ (cons a_2 a_1) x)
           (≡ a_1 y)))))))
```

Even though this version of λ^e and **match**^{e} uses **mem-id** it now correctly identifies the two *a* variables as unique, since the binding is created by **exist** before the next section of the pattern is expanded.

[5] In this example we have taken some liberties, 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.

```
(define-syntax λ^e
  (syntax-rules ()
    ((_ args c c* ...)
     (λ args (match-to-cond^e args (c c* ...))))))

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

λ^e and **match**^{e} provide the interface to the underlying set of macros that eventually generates the finished **cond**^{e} expression. In the case of λ^e a λ expression with the same *args* will be generated around the **cond**^{e} expression, while **match**^{e} will generate a **let** binding for the expression to be matched.

```
(define-syntax match-to-cond^e
  (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*)) ...))))
```

match-to-cond^e 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 (unquote unquote-splicing __)
    ((_ have () (() . g*)) (exist-helper () . g*))
    ((_ have (a . a*) ((p . p*) . g*))
     (do-pattern-opt have a p a* p* . g*))
    ((_ have a (p . g*))
     (do-pattern-opt have 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** and **do-pattern-opt** to generate the variable bindings and unifications for each clause, finally expanding into the list of user supplied goals.

```
(define-syntax do-pattern-opt
  (syntax-rules (unquote unquote-splicing __)
    ((_ have a (unquote p) a* p* . g*)
     (mem-id p (a)
       (do-clause have a* (p* . g*))
       (do-pattern have a ,p () ,p a* p* . g*)))
    ((_ have a (unquote-splicing p) a* p* . g*)
     (mem-id p (a)
       (do-clause have a* (p* . g*))
       (do-pattern have a ,@p () ,@p a* p* . g*)))
    ((_ have a __ a* p* . g*) (do-clause have a* (p* . g*)))
    ((_ have a p a* p* . g*)
     (do-pattern have a p () p a* p* . g*))))
```

While λ^e and **match**^{e} could generate unifications for each part of the pattern, better code can be generated by recognizing unnecessary unifications and not generating them. **do-pattern-opt** ensures that no unification is generated when the logic variable or nom in the pattern part matches the formal parameter or when a __ indicates a "do not care" argument. In all other cases **do-pattern** will be used to generate variable bindings and the unification.

```
(define-syntax do-pattern
  (syntax-rules (quote unquote unquote-splicing)
    ((_ have a () () op () ())
     (do-clause have () (() (pat-to-goal #t op () () a))))
    ((_ have a () () op a* p* . g*)
     (exist ()
       (pat-to-goal #t op () () a)
       (do-clause have a* (p* . g*))))
    ((_ have a (unquote p) r op a* p* . g*)
     (mem-id p have
       (do-pattern have a r () op a* p* . g*)
       (exist (p) (do-pattern (p . have) a r () op a* p* . g*))))
    ((_ have a (unquote-splicing p) r op a* p* . g*)
     (mem-id p have
       (do-pattern have a r () op a* p* . g*)
       (fresh (p) (do-pattern (p . have) a r () op a* p* . g*))))
    ((_ have a (quote p) r op a* p* . g*)
     (do-pattern have a r () op a* p* . g*))
    ((_ have a (pa . pd) () op a* p* . g*)
     (do-pattern have a pa pd op a* p* . g*))
    ((_ have a (pa . pd) r op a* p* . g*)
     (do-pattern have a pa (pd . r) op a* p* . g*))
    ((_ have a p r op a* p* . g*)
     (do-pattern have a r () op a* p* . g*))))
```

The main job of λ^e and **match**^{e} is to generate variable bindings and unifications for the pattern. **do-pattern** is responsible for generating these bindings. **unquote** and **unquote-splicing** use **mem-id** to determine if the logic variable or nom has been encountered, If not a biding is generated and it is added to the list of known variables. The other clauses are responsible for traversing the full pattern. Some optimization is also peformed 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, treating the final pattern unification as a user supplied goal.

Once bindings for all new variables exist, the original pattern is passed off to **pat-to-goal** and the rest of the pattern and formal parameters are passed back to the **do-clause** to continue processing.

```
(define-syntax pat-to-goal
  (syntax-rules (quote unquote unquote-splicing ⋈ __
                 pair-l pair-r ⋈-l ⋈-r)
    ((_ #f e () t* v) (exist-helper t* (≡ e v)))

    ((_ #t __ z t* v) (pat-to-goal #f t z (t . t*) v))
    ((_ #t (unquote p) z t* v) (pat-to-goal #f p z t* v))
    ((_ #t (unquote-splicing p) z t* v)
     (pat-to-goal #f p z t* v))
    ((_ #t (quote p) z t* v) (pat-to-goal #f 'p z t* v))

    ((_ #t (⋈ pa pd) z t* v)
     (pat-to-goal #t pa (⋈-l z pd) t* v))
    ((_ #f e (⋈-l z pd) t* v)
     (pat-to-goal #t pd (⋈-r z e) t* v))
    ((_ #f e (⋈-r z pa) t* v)
     (pat-to-goal #f (⋈ pa e) z t* v))

    ((_ #t (pa . pd) z t* v)
     (pat-to-goal #t pa (pair-l z pd) t* v))
    ((_ #f e (pair-l z pd) t* v)
     (pat-to-goal #t pd (pair-r z e) t* v))
    ((_ #f e (pair-r z pa) t* v)
     (pat-to-goal #f (cons pa e) z t* v))

    ((_ #t p z t* v) (pat-to-goal #f 'p z t* v))))
```

Each pattern part and its corresponding formal parameter are assembled into a unification by **pat-to-goal**. The **pat-to-goal** macro is also responsible for generating temporary logic variables for places where _ is used within a pattern. These temporaries will never be unified with other variables, but need to be created to ensure the shape of the pattern is being preserved. Since **pat-to-goal** must create variable bindings and the final goal, we rely on something very similar to the CPS-macros used in the first implementation. Here though, the continuation is represented using a zipper [**?**] to indicate how the tree has been traversed and if a $\bowtie$ or *cons* is the appropriate constructor for a given part of the pattern.

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

exist-helper is a helper macro, used to generate the *succeed* goal when supplied an empty argument list and no goals, the provided goal, when supplied an empty argument list and a single goal, or a normal **exist** expression when variables need to be bound or more then one goal has been supplied.

6. Future Work

6.1 Further Investigations into CPS-macros and Binders

Although we were able to rewrite our CPS-macro into a more traditional style macro, our approach was largely to use what we had learned to rewrite from scratch. The traditional method also required us to have some knowledge of how the underlying **cond**^{e}, **exist**, and **fresh** macros work in order to ensure that we did not produce invalid code with our macro. It would be preferable to have an approach for transforming CPS-macros into traditional style macros and vice-versa, so that a macro programmer who encounters similar problems is not left needing to restart from scratch.

It is also possible that there is an idiom we could use with CPS-macros that would allow us to get the best of both worlds, allowing us to return to a CPS-macro implementation of λ^e and **match**^{e} without the problem related to identifiers being left free during the process as our current implementation does.

6.2 Improving the λ^e and match^e Macros

While the intent of λ^e and **match**^{e} is to provide a convenience for programmers using αKanren, they also provide information about the variables used in a pattern match that was previously unavailable to $\equiv$.

Since the pattern matching macro knows when new logic variables and fresh noms are being introduced, it may be possible to use this information to create a specialized version of $\equiv$ that would use this information, preserving the soundness of αKanren's nominal unification, while providing better performance. In particular in the case of new logic variables it may be possible to avoid looking up these variables in the substitution, avoiding a relatively expensive operation.

In addition to knowing which variables were created in the pattern, we may also be able to create specialized versions of $\equiv$ during the expansion process that make use of the structure of the pattern in order to perform the unification more quickly avoiding looking up new logic variables during the unification process.

7. Conclusion

λ^e and **match**^{e} bring a simple, but powerful pattern matching abstraction to αKanren, that should help programmers express relations more concisely and lead to easier to use code. While pattern matching macros are nothing new in the Scheme community, our pair of CPS-macros with variable binding creation uncovered a macro design issue the recommends a certain amount of caution be exercised when CPS-macros are used to selectively bind new variables. If not it is possible to write macros that break our intuitions about how hygienic macros work leading to the possibility of generating invalid code or potentially even variable capture in the worst case, all without leaving the relative safety of **syntax-rules** macros. Finally, it demonstrated that in certain situations more traditional macro writing techniques seem to be the only way to accomplish certain tasks.

8. Acknowledgements

We gratefully acknowledge Lindsey Kuper, who initially pointed out the bug with how our CPS-macro implementation was treating identifiers.

A. First λ^e and match^e Implementation

The CPS-macro version of λ^e and **match**^{e} follows. Note that in this version of the code, **mem-id** has been generalized to provide the *case-id* macro, which works similar to **case**.

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

(define-syntax make-match-cont
  (syntax-rules ()
    ((_ args a a* p p* g ((pr* . g*) ...) . pc)
     (handle-pattern top a p
       (make-clause-cont a a* p* ()
         (build-clause g
           (match-to-cond^e args ((pr* . g*) ...) . pc)))
       ((a . a*) () ())))))

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

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

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

(define-syntax make-clause-cont
  (syntax-rules ()
    ((_ a a* p* l k (seen evar fvar) . p)
     (build-goal a
       (build-var evar fvar
         (build-clause-part a* p* l seen k)) . p))))

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

(define-syntax build-pattern
  (syntax-rules ()
    ((_ f (k* ...) lv p* ... pa pd) (k* ... lv p* ... (f pa pd)))))

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

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