# Plaid: Pattern Language for Abstract Datatypes

Dan R. K. Ports   Austin T. Clements   Irene Y. Zhang

May 14, 2007


## Abstract

*The expressiveness of traditional syntactic pattern matching is severely limited by its lack of abstraction. Because syntax patterns are mired in the built-in types understood by the pattern matching system, they lack the ability to express patterns over abstract data types (ADT's). More advanced pattern matching techniques, such as semantic matching, can overcome this, but at the per-ADT cost of the complex code required to add new pattern combinators to the system.*

*Plaid defines a new pattern language that captures a strict subset of Scheme capable of both regular computation, as well as reverse computation. This allows it to overcome both the limitations of syntactic patterns and the cost of semantic patterns by providing a means by which programmers can write a single specification of the mapping between the abstract and concrete representations of an ADT that simultaneously serves as constructor, predicate, accessor, and pattern combinator for that ADT. This specification is written virtually identically to how a regular constructor would be written.*

*Furthermore, the Plaid pattern language is capable of capturing non-determinism and decisions within pattern matching, thus admitting a very broad interpretation of what can be considered an ADT constructor. This leads to variety of interesting capabilities, such as the ability to view concrete data in multiple abstract ways, the ability to canonicalize multiple concrete representations in one abstract way, and the ability to imagine more convenient representations of existing data.*


## 1 Introduction

Syntactic pattern matching is a classical component of symbolic programming. Unfortunately, its expressiveness is limited by its lack of abstraction. Many languages typically used for symbolic programming, such as Scheme, have non-extensible type systems, meaning that abstract data types exist merely as user-enforced contracts. Since syntax patterns are limited to the *concrete* data representations understood by the pattern matching mechanism (which typically do not extend beyond the built-in types of the language), such systems have no means of expressing patterns over *abstract* data representations. Pattern matching on abstract data types *requires* violating the abstraction barrier and using the concrete representation of the type.

```
(define (make-computer model os)
  (list '*computer* os model))
(define (computer? datum)
  (and (pair? datum)
       (eq? (car datum) '*computer*)))
(define computer-model third)
(define computer-os second)
```

Figure 1: A simple ADT representing a computer

For example, consider the simple ADT presented in Figure 1. In order to use a computer instance in pattern matching, for example to test whether or not a computer is running Linux, a programmer would have to violate the abstraction barrier and write something in terms of the concrete representation, such as

```
(match computer '(*computer* linux _))
```

Semantic pattern matching can potentially alleviate this problem by providing a means of extension by which implementations of abstract data types can provide custom match combinators that expose only abstract representations. Figure 2 demonstrates an implementation of a computer combinator for the semantic pattern system used in *Adventures in Advanced Symbolic Programming*. However, as can be seen from the figure, this vastly complicates the implementation of an abstract data type without fundamentally extending its power.

All of the information about the mapping between the abstract representation and the concrete repre-

```
(define (match:computer? pattern)
  (and (pair? pattern)
       (eq? (car pattern) computer)))
(define match:computer-model second)
(define match:computer-os third)

(define (match:computer model-comb os-comb)
  (define (match data dictionary succeed)
    (and (computer? data)
         (model-comb
          (computer-model data) dictionary
          (lambda (new-dict n)
            (os-comb
             (computer-os data) new-dict
             succeed)))))
  match)

(define (compile-match-computer
         pat use-env loop)
  `(match:computer
    ,(loop (match:computer-model pat))
    ,(loop (match:computer-os pat))))
(eq-put! computer 'pattern-keyword
         compile-match-computer)
```

Figure 2: Implementation of a semantic matching combinator for the computer ADT

sentation is *already* captured by the type's constructor. Thus, any additional code required to inform the pattern matching system of how to handle the abstract representation of a type is redundant with the information already contained in the constructor. In fact, even the typical predicate and accessor functions associated with an abstract data type can be considered a duplication of the knowledge already expressed in the constructor.

In Scheme's evaluation model, abstract data constructors work by eventually reducing to combinations of concrete data constructors. Thus, when used under Scheme's evaluation rules, an abstract data constructor will eventually produce some concrete data structure. However, many of the forms of combination that produce the concrete data structure can be run in *reverse*, using the constructor's own code as a specification of how to *deconstruct* the results of the abstract constructor.

Pla*i*d builds a new type of pattern matching system around this notion of reversible computation. We introduce a pattern syntax and pattern matching model that can express patterns in terms of abstract representations. This pattern language is symmetric, allowing a single expression to run either forward or in reverse, so it can be used either to construct or to deconstruct. This pattern language is also expressive enough to capture branching control flow (in the forward direction) and non-determinism (in the reverse direction), allowing it to express not only classical abstract data type constructors, but constructors that alter their behavior depending on their inputs. This allows Pla*i*d patterns to the take advantage of the reversibility of computations like `append`; to view myriad concrete representations in abstract, canonical ways; and to imagine multiple abstract representations for a single concrete representation.

Section 2 describes the pattern language and pattern matching model introduced by Pla*i*d and addresses its implications for pattern matching in the presence of abstract data types as well as non-determinism. Section 3 discusses the pattern matching algorithm required to capture the capabilities of the language. Finally, Section 4 covers our implementation of the pattern system.

# 2 Pattern Language

Pla*i*d integrates a new pattern-matching language into Scheme. It introduces a new *constructor-oriented* pattern syntax to support patterns expressed in terms of abstract data types. The pattern language is embedded in Scheme as two new special forms supporting pattern abstraction and pattern instantiation. Control flow and non-determinism *within* patterns is supported by embedding the pattern instantiation special form into the pattern language itself.

## 2.1 Constructor-Oriented Pattern Syntax

Traditional syntactic patterns are written in a *data-oriented* syntax. This syntax reflects the syntax of reader/printer in the language, but allows for *holes* in the representation in the form of pattern variables. For example, to match a list of two elements whose first element is the value `1` while capturing the value of the second element, one would write a pattern such as

```
(1 (? x))
```

This pattern is itself a list of two elements, whose first element is a `1`. Its only structural difference from the data it is matching is that it contains the pattern variable `(? x)`.

Because the representation of data-oriented patterns precisely mirrors the representation of the data they are matching, the very syntax of such patterns limits them to the concrete types provided by the language, such as (in Scheme), atomic types (numbers, booleans, etc.) and pairs.

Pla*i*d introduces a new syntax for patterns that, instead of resembling the concrete representation of the data being matched, resembles the *code* that would construct the data being matched. In this *constructor-oriented* syntax, the above pattern would be written as

```
(cons 1 (cons x ()))
```

Like in data-oriented patterns, any self-evaluating form is treated as a pattern literal and must match the datum exactly in the sense of equal?. Literal symbols can be matched by means of a **quote** special form in the pattern language[1]. Pattern variables are specified as symbols in the non-first position of a combination. Finally, the special symbol _ indicates a wildcard or "don't-care" pattern.

In order to equal the power of syntactic patterns, the pattern language supports a basis set of *fundamental deconstructors*. Each built-in compound data type has a corresponding fundamental deconstructor (in Scheme, this consists only of pairs, though the system can easily be extended with new fundamental deconstructors). A deconstructor is responsible for checking the type of a datum and recursively pattern matching the components of the datum with the appropriate sub-patterns.

Data-oriented patterns have *implicit* deconstructors. A syntax pattern that contains a pair implicitly means that the datum must be a pair and that the car and cdr of the datum must recursively match the car and cdr of the pattern. Constructor-oriented patterns, on the other hand, make deconstructor names *explicit*. The pattern (cons x ()) is a *combination*, much like a combination in Scheme. The name cons in the first position corresponds to the fundamental deconstructor for pairs (assuming the binding of the name cons hasn't been shadowed). This indicates that the pattern only matches pairs whose car matches the first argument to cons and whose cdr matches the second argument to cons.

Explicitly named deconstructors naturally lead to the addition of abstract or *compound* deconstructors to the pattern language. In Scheme, an abstract constructor works by ultimately reducing to some combination of built-in constructors that give the abstract data type is concrete representation. We can do the same thing for deconstructors: provide the mechanism to add a new named deconstructor to the system with a set of formal parameters and whose body is simply a pattern. When this deconstructor is referred to in a pattern, we can follow in the footsteps of the substitution model by simply reducing the pattern to the body of the deconstructor with the argument patterns substituted for the formal arguments. For example,

```
(define-constructor (computer model os)
  (list '*computer* os model))
```

defines a compound deconstructor for the computer ADT, providing a bridge in the pattern language between the abstract and concrete representations of this implementation of a computer. When the pattern matching encounters a pattern of the form

```
(computer _ 'linux)
```

it reduces this to the body of the computer deconstructor, with the argument patterns substituted for the formals.

```
⇝( list '*computer* 'linux _)
```

The deconstructor list itself merely reduces to a pattern consisting of applications of cons

```
⇝(cons '*computer* (cons 'linux (cons _ ())))
```

This final pattern consists solely of fundamental deconstructors, and thus can easily be matched against a concrete datum. The simplified algorithm demonstrated here for handling compound deconstructors is overly eager, but forms the conceptual basis for Pla*i*d's actual matching algorithm, which will be presented in Section 3.

## 2.2 Code–Pattern Duality

Because the pattern language is a strict subset of Scheme, any valid pattern is simultaneously valid Scheme code. Due to the syntax and semantics of the pattern language, the body of a deconstructor *doubles* as a perfectly operational definition of a *constructor*. For example, if one treats the definition of computer given above as a regular procedure definition and runs it "forward" as regular Scheme code, it serves as an effective constructor for the computer ADT. As we saw in the previous section, if we look

[1]Unlike Scheme's **quote**, Pla*i*d's **quote** is currently restricted to symbols, but could be generalized

at the body of the definition as a pattern and run it "backwards", it serves as a deconstructor.

Thus, this single definition, which captures the mapping between the abstract and concrete representations of the ADT, suffices as the one and only specification of this mapping – acting at once as constructor, predicate, and accessor for the ADT. Furthermore, no specific additions to the pattern system beyond this single definition are necessary in order to extend pattern matching to the abstract representation.

**Views** When a collection of abstract constructors share the same concrete representation, it's possible to *view* a given concrete representation in multiple ways through pattern matching. Essentially, the mapping between abstract representation and the concrete representation of an ADT need not be one-to-one; it can be many-to-one. Because the system is unaware of how a concrete datum was originally created, *any* of the constructors that could have been used to create the datum can be used to deconstruct it, thus admitting any number of abstract views to be taken of a single piece of data.

## 2.3 Embedding Patterns in Scheme

Pla*i*d adds two new important special forms (implemented as macros) to the Scheme language – one for pattern abstraction and one for pattern instantiation.

The first, **plambda**, creates new compound deconstructors. Its syntax and semantics are much like those of **lambda**, except that the body must consist of a single, valid pattern. Its value is a procedure that can be used either as a regular Scheme procedure, or as a deconstructor in a pattern. Thus,

```
(define computer
  (plambda (os model)
    (list '*computer* model os)))
```

binds computer to a procedure that, if applied in regular Scheme code, constructs a new compound datum representing a computer; and if applied in a pattern, deconstructs a compound datum as it would a computer.

Just as **define** has a sugared form for defining procedures because the creation and naming of a procedure are so often done together, Pla*i*d has **define−constructor** as sugar for the definition of a constructor/deconstructor. The desugaring mirrors that of **define**, but results in a **plambda** form in place of the usual **lambda** form.

Pla*i*d also introduces a **pcase** expression, which integrates pattern matching directly into Scheme's syntax, control flow, and environment model. The syntax of **pcase** resembles Scheme's **case** expression, but the conditions are patterns. The exact syntax is

```
(pcase expr
  (pattern expr expr ...)
  (pattern expr expr ...)
  ...)
```

**pcase** first evaluates the expression that precedes the clauses. The value of this expression is pattern-matched against the pattern in first clause. If it matches, then any pattern variables in the pattern are bound over the scope of the body of the clause and the body is evaluated. If the match fails, then the next clause is tried, and so on. In this way, **pcase** integrates pattern matching, control flow from the possible failure of pattern matching, and the projection of pattern variables into Scheme's environment.

Pla*i*d diverges from the typical match-function based interface to pattern matching for a number of reasons. Beyond the simple convenience of language-integrated pattern matching, this allows **pcase** to interact with the environment model in important ways. Not only does it allow Pla*i*d to close the gap between pattern variables and Scheme variables by making pattern variables available as regular Scheme variables scoped over the body of the clause, but it also allows proper lexical scoping of deconstructor names.

## 2.4 Control Flow in Patterns

It turns out that **pcase** is fundamentally more than merely a convenient way to express pattern matching in Scheme. By introducing **pcase** into the *pattern language* as well Scheme, the pattern language remains a subset of Scheme, but gains the power of *control flow*.

This allows us to broaden our definition of a constructor beyond simply a direct mapping between an abstract representation and a concrete representation. By admitting **pcase** into the pattern language, constructors are allowed to make decisions about the contents of the concrete representation based on the value of the abstract representation. This leads to branching control flow in constructors and, symmetrically, non-determinism in deconstructors.

**Reversible Computation** In a pattern language without control flow, many interesting constructors

```
(define-constructor (lambda* args body)
  (cons 'lambda (cons args body)))

(define-constructor (define* name expr)
  (pcase expr
    ((lambda* args body)
     (cons 'define
           (cons (cons name args) body)))
    (_ (list 'define name expr))))
```

Figure 3: Another way of looking at Scheme's **define**

are inexpressible. For example, we can now think of append as simply an abstract constructor for lists. We could replace Scheme's standard definition of append with the following

```
(define-constructor (append a b)
  (pcase a
    (() b)
    ((cons this rest)
     (cons this (append rest b)))))
```

Viewed in the forward direction – as regular Scheme code – this implements a somewhat stylized but perfectly operational version of regular append. However, since this definition of append is also valid in the pattern language, it becomes possible to use append in reverse. For example,

```
(pcase '(1 2 3 4)
  ((append x (list 3 4))
   x))
⇒ (1 2)
```

**Canonicalization** Control flow introduces ambiguity into the pattern language. Without control flow, we are limited to mappings from one or more abstract representations to a single concrete representation. Control flow in the pattern language allows us to describe a mapping from multiple concrete representations to one abstract representation, allowing us to *canonicalize* data and to program in terms of unified abstract representations.

For example, Figure 3 shows how one could canonicalize a subset of Scheme's syntax consisting of **lambda**, **define**, and the sugared form of **define**. The "concrete representation" in this example is a Scheme abstract syntax tree, while the "abstract representation" is a canonicalized form consisting of the abstract lambda* and the abstract define*, but *not* an equivalent to the sugared form of **define**.

With these we could, for example, start writing an eval dispatch like the one presented in SICP that relies on patterns for predication, sub-expression selection, as well as desugaring:

```
(define (eval exp env)
  (pcase exp
    ((lambda* args body)
     (make-procedure args body env))
    ((define* var val)
     (define-variable! var (eval val env) env))
    ...))
```

**Imagination** Because **pcase** in the pattern language introduces non-determinism into pattern matching, it may be possible to find multiple valid instantiations of a pattern. The ability to explore multiple instantiations is critical to the matching algorithm, as will be discussed in Section 3.4. **pcase** in Scheme directly exposes this to the user in the form continuation called next that can be invoked to try another satisfying assignment of pattern variables. If there are no more possibilities for the current clause, this continues to the next clause. The ability to try other assignments leads to the possibility of *imagining* multiple abstract ways of looking at one concrete representation.

For example, we can write the following constructor for multiplying two expressions,

```
(define-constructor (** x y)
  (pcase x
    (1 y)
    (_ (pcase y
         (1 x)
         (_ (list '* x y))))))
```

When used in a pattern, the ** deconstructor lets us look at something that is not a multiplication as a multiplication by 1. For example, we can imagine (+ x y) as a multiplication with

```
(pcase '(+ x y)
  ((** a b) (list 'mul a 'by b)))
⇒ (mul 1 by (+ x y))
```

On the other hand, if we try to imagine (* x y) as a multiplication, the pattern matcher will first discover that we can multiply this by 1 to view it as a multiplication. While true, this is not useful, so we can use the next continuation to ask for other ways of imagining it.

```
(pcase '(* x y)
  ((** a b)
   (if (or (eqv? a 1) (eqv? b 1))
       (next)
       (list 'mul a 'by b))))
⇒ (mul x by y)
```

# 3 Matching Algorithm

The new pattern language described above requires a substantially different matching algorithm than a typical pattern matcher because of the complexities of abstraction and control flow in patterns. Here, we present an algorithm that allows patterns in the Pla*i*d pattern language to be matched.

## 3.1 Matching Requires Unification

Though a traditional pattern matcher appears to suffice for matching Pla*i*d patterns, in fact *unification* is necessary. Unification is the generalization of pattern matching in which both arguments may contain patterns. Indeed, since unification is symmetric, both arguments can be referred to as *patterns* rather than a pattern and a datum.

To see why unification is required, consider again the example of append:

```
(define-constructor (append a b)
  (pcase a
    (() b)
    ((cons this rest)
     (cons this (append rest b)))))
```

The second clause requires both that a can be expressed as a cons of this and rest, and that b can be expressed as a cons of this and a append call, or in other words that this is the first element of both a and the datum. Because a **pcase** expression can place multiple requirements on a variable, such as this in this example, unification is necessary to ensure that a satisfactory assignment is found, if one exists.

## 3.2 Basic Unification

The basis for our unification algorithm is the one from SICP. It takes two patterns as an input, along with a *unifier environment*. The unifier environment contains a set of bindings from variable names to their values. (In Section 3.4, we extend the unifier environment to also include a failure continuation.) The unifier returns a new environment containing all the bindings from the initial environment, plus any new bindings that needed to be added to make the two patterns equivalent. Note that a pattern variable may be bound to an expression containing other variables, in order to express a dependency between variables.

The most basic set of unification rules is shown in Table 1, for a pattern language consisting entirely of literals, wildcards, and variables (*i.e.* no deconstructors or choices). Two literals unify if they are identical, and wildcards unify with anything. When either one of the patterns is a variable, the environment is extended, binding that variable to the other pattern, if possible. If the variable already has a value, the old value is unified with the proposed new value, which may cause unification to fail, or which may cause other variables to become bound.

| **Condition** | **Rule** |
|---|---|
| Both literals? | Succeed if *same* literal |
| Either wildcard? | Succeed |
| Either variable? | Add binding if consistent |
| One literal? | Fail |

Table 1: Basic unification rules

## 3.3 Unifying Deconstructors

The pattern language from the previous section is not particularly interesting, since it does not include support for deconstructors, even fundamental ones like cons. Unification of fundamental deconstructors is straightforward. Two fundamental deconstructors unify if they are of the same type (*e.g.* they are both conses), and if each argument unifies with its corresponding argument on the other side.

Unification of compound deconstructors, the analogous operation to application in an evaluator, proceeds as follows. The body of the deconstructor is unified against the datum, with argument names $\alpha$-renamed in order to avoid conflicts with other uses of the same compound deconstructor. In the environment that results from this unification, each of the $\alpha$-renamed argument names is unified against the pattern from the original application of the deconstructor. An example is shown in Figure 4.

An alternate approach to compound deconstructor unification uses $\beta$-substitution instead of $\alpha$-renaming to handle arguments. Rather than unifying the body of the deconstructor with the datum and then unifying the argument names with their values, we could instead substitute the appropriate argument value wherever the corresponding name appears in the deconstructor body, and unify the resulting expression against with the datum. Though the two alternatives are clearly semantically equivalent, we opted for the $\alpha$-renaming-and-unification strategy because $\beta$-substitution could be more computationally intensive if a complex expression needs to be substituted into an expression in multiple places.

```
(define (make-computer model os)
  (cons '*computer*
        (cons os (cons model ()))))
```

1. ```
   (unify (make-computer x 'linux)
          (cons '*computer*
                (cons 'linux (cons 'pc ()))))
   ```

2. ```
   (unify (cons '*computer*
                (cons os (cons model ())))
          (cons '*computer*
                (cons 'linux (cons 'pc ()))))
   ```

   $\Longrightarrow$ os $\leftarrow$ linux; model $\leftarrow$ pc

3. ```
   (unify model x)
   ```

   $\Longrightarrow$ x $\leftarrow$ pc

4. ```
   (unify os 'linux)
   ```

Figure 4: Example of compound deconstructor unification. A compound deconstructor is unified against a datum consisting of cons fundamental deconstructors (1). To perform this unification, first the deconstructor body is unified against the datum, yielding bindings for the formal arguments (2). Next, each of the formal arguments is unified against its corresponding parameter from the call to the deconstructor (3 & 4). $\alpha$-renaming of parameter names is not shown.

### 3.4 Nondeterministic Unification

Unification becomes fundamentally different once the ability to make choices is introduced into the pattern language. For example, the expression

```
(pcase x
  (1 'a)
  (_ 'b))
```

could be equal to either 'a or 'b if x = 1. At the time the **pcase** is encountered, it is not possible to determine which value it should take, since a later unification might force that choice to be reconsidered. For example, we might unify a variable y with the **pcase** expression above, choosing 'a for the value of y, and later learn that that choice was incorrect by unifying y with 'b. This requires control flow backtracking, since the value of y might have been used to make decisions in the meantime.

To address this challenge, we expand the definition of a unifier environment slightly: in addition to a set of variable bindings, it may now include a failure continuation that can be called in the event that a dependent unification fails in the future. Calling this failure continuation expresses the notion that no consistent unification is possible using the current variable bindings, and requests another possible variable assignment. Control is backtracked to the previous **pcase** pattern, and the next clause in sequence is tried. Of course, it is possible for unification of the **pcase** to fail, in which case *its* failure continuation is called, perhaps causing another control flow backtracking. The root failure continuation simply causes the most recent call to the unifier to return *#f*. It is used when no consistent unification environment can be found, such as when no choices have been made during unification (*e.g.* during the unification of an expression that contains no **pcase** clauses) or if every **pcase** clause has led to a unification failure.

As a result of this backtracking, unification becomes a search problem. It can be viewed as searching the space of possible **pcase** choices for an assignment that allows unification to proceed. As a result, search techniques such as dependency-directed backtracking can be applied.

## 4 Implementation

Pla*i*d is implemented atop Scheme as an elegant set of macros and procedures; it does not require any changes to the underlying Scheme evaluator. The major components of the implementation are a representation for patterns and mechanism for constructing them, described in Section 4.1; constructs for defining new constructors and deconstructors, described in Section 4.2; and a unifier that performs the actual pattern matching, described in Section 4.3.

### 4.1 Abstract Pattern Structure

Patterns are represented internally by a set of abstract data types representing the different pattern structures: literals, variables, wildcards, deconstructors, and **pcase**s. The unifier operates solely on these data types.

The abstract pattern structure can be generated in two ways. First, a *pattern compiler* operates on syntactic expressions at macro-expansion time. For example, it converts (cons 1 (cons a ())) into the abstract structure shown in Figure 5. This compiler is used, for example, to build the appropriate patterns from the arguments to a **pcase** expression. Second, the *constructorifier* operates on data at runtime. It

```
(pattern:deconstructor
 (lambda ()
   (constructor->deconstructor cons))
 (list
  (pattern:literal 1)
  (pattern:deconstructor
   (lambda ()
     (constructor->deconstructor cons))
   (list (pattern:variable a)
         (pattern:literal ())))))
```

Figure 5: Example of abstract pattern structure

converts data — pairs, symbols, numbers, etc. — to the *same* abstract representation. For example, the list (1 a) also translates into the abstract structure of Figure 5.

Using the same representation for both patterns and data reflects the duality between constructors and deconstructors described in Section 2.1. A datum is represented in exactly the same way as the expression that produces it. This symmetry makes the unifier elegant, since it operates on two arguments that have the same form, rather than a pattern and datum with different structure.

## 4.2 plambda and define-constructor

Pla*i*d integrates a new constructor, **plambda**, into Scheme as a macro. Like **lambda** in Scheme, the **plambda** macro returns a Scheme procedure, but in addition, **plambda** binds the newly created procedure to a compound deconstructor using an eq-hash table. The compound deconstructor includes a list of formals and the pattern-compiled version of the procedure body. A procedure created by **plambda** is exactly the same as a procedure returned by a **lambda** when used in normal Scheme code, however the procedure is also valid pattern syntax. When the procedure occurs in a pattern, the corresponding deconstructor is retrieved and used for matching the pattern.

Because the constructor-to-deconstructor mapping is stored in the hash table using the procedure as its identifier, the procedure itself serves as the identifier of the deconstructor. At compile-time, a deconstructor record is created containing a thunk that looks up the deconstructor associated with the name at runtime. This means that deconstructor names inherit the scoping properties of normal Scheme variables, since the deconstructor name is resolved by Scheme according to the environment model, and the constructor-to-deconstructor mapping is automatic.

```
(define (unify p1 p2 env)
  (cond ((both? pattern:literal?)
         (if (eq? (pattern:literal/value p1)
                  (pattern:literal/value p2))
             env
             ((unifier-environment/fail env))))
        ((either? pattern:wildcard?) env)
        ((either? pattern:variable?)
         => (w/l (lambda (var other)
                   (extend-if-possible
                    var other env))))
        ((either? pattern:literal?)
         ((unifier-environment/fail env)))
        (else (error "Don't know how to unify"
                     p1 p2))))
```

Figure 6: Core unifier code, implementing basic unification (without deconstructors or pcases)

A **define−constructor** macro is introduced to provide syntactic sugar for the common operation of creating a constructor and binding it to a name. In much the same way as **define** can create an implicit **lambda** and bind it to a name, **define−constructor** can do the same with a **plambda**.

## 4.3 Unification

In order to perform pattern matching, Pla*i*d employs a unifier as described in Section 3. The unifier operates on the abstract pattern structures described in Section 4.1 as well as *unifier-environment* structures that contain a set of variable bindings and a failure continuation. The unifier takes two patterns and an initial environment, and returns either a new environment with the necessary bindings to make the patterns unify, or *#f* if no unification is possible.

A simplified version of the core unifier code is shown in Figure 6. The unifier is implemented in a data-directed style, testing the types of its input patterns and applying an appropriate rule.[2] Unification of fundamental and compound deconstructors proceeds as described in Section 3.3.

### 4.3.1 Nondeterminism and Backtracking

To introduce nondeterminism into the unifier to support **pcase** patterns, we use **call−with−current−continuation** in much the same way one might use it to implement McCarthy's

[2]The data-directed style of the unifier suggests replacing the monolithic **cond** statement with a dispatch table to improve additivity. Though it is a useful strategy in general, we rejected this approach here because the order of rule evaluation is critical and we wanted to make this explicit.

amb operator. Upon encountering a **pcase** pattern, the unifier captures its current continuation using **call−with−current−continuation**, installs it in the unifier environment, and proceeds to unify the consequent against the datum and the condition against the variable. If either of these unifications, or another unification in the future fails, the failure continuation is invoked and control flow returns to the **pcase** unification. The unifier then tries the next clause, or recursively invokes its failure continuation if no clauses remain.

If necessary, a root failure continuation is installed as the first step to a call to unify. Specifically, the unifier checks whether the input environment already includes a failure continuation. If it does not (represented by the symbol cant−fail), the unifier creates a new failure continuation that, when invoked, causes the current call to unify to return *#f*. It then proceeds with unification, using this new failure continuation. After unification is completed, the unifier checks whether one of its unification steps created a new failure continuation (*e.g.* if a **pcase** was encountered within). If it has not, then it replaces the root failure continuation with the symbol cant−fail before returning the environment to the user.

To see why this is necessary, consider the example of Figure 7. In this example, a compound deconstructor containing a **pcase** is unified against a datum. The **pcase** unifier tries the first clause, but when the unification fails, the failure continuation is invoked, backtracking to the **pcase**, which tries the next clause and succeeds. Observe that it is sometimes necessary to keep the failure continuation in the environment returned to the caller, where it might be used for another unification, or exposed to the user as the next construct of a Scheme **pcase**. For example, if instead of using ( list '∗computer∗ x 'osx) as our datum, we had used ( list '∗computer∗ x y), the unification of the first clause would have succeeded, binding x to 'pc and y to 'freebsd. But if we later unified y with 'osx, we would have to backtrack to the **pcase** and try again, since a unifying assignment is possible with the other clause. However, the root failure continuation should never be returned to the user (hence the need to replace it with cant−fail). If it were returned, a failure in a future call to unify would cause the *previous* call to return *#f*.

#### 4.3.2 Scoping

The unifier uses $\alpha$-renaming to properly implement the correct scoping behavior of compound deconstructors and **pcase** statements. Before the body of a compound deconstructor is unified, in the algorithm of Section 3.3, the names of the formal parameters are renamed to new uninterned symbols. This means that name conflicts will not occur, either between the name of one of the parameters and a variable used elsewhere, or between parameters in multiple applications of the same deconstructor.

Similarly, **pcase** scoping requires that, when unifying the consequent of a clause, variables occurring outside the pcase can be shadowed by new variables created inside the condition of the clause. For example, in the expression

```
(pcase x
  ((cons x y) x))
```

the x appearing in the only clause is distinct from the x that serves as the key for the **pcase**. This is also implemented using $\alpha$-renaming. Before unification of a **pcase**, the free variables of the condition are identified (taking into account that it may contain a **pcase** itself), and $\alpha$-renamed within both the condition and consequent of that clause.

## 5 Future Work

There are a number of aspects of the Pla*i*d pattern matching language, algorithm, and implementation we would look to examine further in future. While the pattern syntax is a subset of Scheme, it diverges from the simple and absolute uniformity of Scheme syntax. In particular, we would like to thoroughly understand and, if possible, reconcile the currently incompatible semantics of the first position of a pattern combination, which must be a name that corresponds to a deconstructor in the environment, and the remaining positions, which must be patterns. We would also like to explore how numbers can be better handled within the pattern language. Currently, the pattern system is limited to treating numbers as opaque literals with no notion of the relationships between different numbers. For example, currently arithmetic operators and comparators do not exist in the pattern language.

A number of improvements to the implementation are possible, particularly through more thorough static analysis of patterns. This may allow us to eliminate names in compiled patterns altogether, perhaps replacing them with vector offsets to eliminate the need to construct and search through binding dictionaries. This may also eliminate the need to $\alpha$-rename compound deconstructor bodies.

Finally, there are a number of plausible modifications of the unification algorithm itself. For example, because of the limitations of the pattern language, it may be possible to reliably detect non-terminating conditions during pattern matching (such as a compound deconstructor calling itself). Also, despite the elegance of symmetric unification, it may simplify the system to switch to *asymmetric* unification, in which a pattern is matched against a datum instead of another pattern. This places numerous restrictions on the set of cases the unifier needs to consider, and may eliminate the need for the conversions between data and patterns that are currently necessary when interfacing with the unifier.

# 6 Conclusion

Traditional syntactic pattern matching is incompatible with abstraction, greatly limiting its power. Pla*i*d shows that this incompatibility needs not be the case by introducing a new pattern language that provides support for abstract data types. The key insight that makes abstraction possible is that by defining the pattern language to be a restricted subset of Scheme, a single definition suffices as both a constructor and a deconstructor — it provides both an implementation of a procedure and a mechanism for matching the data produced by that procedure.

Our pattern language incorporates this notion of abstraction based on reversible computation, and a reversible control flow construct, **pcase**. The language is sufficiently expressive to represent many common and useful procedures, often with minimal modifications to their typical implementations. Combining these abstractions with a pattern matching system allows us to easily implement techniques for viewing data in different ways, such as canonicalization and imagination. As a result, systems such as semantic pattern matchers, which traditionally lend themselves to complex pattern matching semantics, become trivial because Pla*i*d allows the programmer to declaratively specify a desired view of the data rather than an implementation of how to convert the data to that view.

```
(define-constructor (my-computer model)
  (list '*computer*
        model
        (pcase model
            ('pc 'freebsd)
            ('mac 'osx))))

1. (unify (list '*computer* x 'osx)
          (my-computer a))

2. (unify (list '*computer* x 'osx)
          (list '*computer*
                model
                (pcase model
                    ('pc 'freebsd)
                    ('mac 'osx))))

3. (unify x model)

   ⟹ x ← model

4. (unify 'osx (pcase model
                 ('pc 'freebsd)
                 ('mac 'osx)))

   ⟹ fail ←#[pcase failure continuation]

5. (unify 'osx 'freebsd)

   ⟹ failure! Invoke continuation and return to (4)

6. (unify 'osx (pcase model
                 ('mac 'osx)))

   ⟹ fail ←#[root failure continuation]

7. (unify 'osx 'osx)

8. (unify model 'mac)

   ⟹ model ← 'mac
```

Figure 7: Example of **pcase** unification. A compound deconstructor containing a **pcase** is unified with a datum consisting of cons fundamental deconstructors (1). When the **pcase** is encountered, a new failure continuation is set (4), and the first clause's consequent is unified with the datum (5). This fails, so control flow backtracks to the **pcase**, which tries the next clause; because the next clause is the final clause, the root failure continuation is kept (6). The **pcase** clause's consequent is unified with the datum (7) and its condition against the key variable (8), which succeeds.