# Statically Checkable Pattern Abstractions

Manuel Fähndrich
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720-1776

John Boyland*
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891

**Abstract**

Pattern abstractions increase the expressiveness of pattern matching, enabling the programmer to describe a broader class of regular forests with patterns. Furthermore, pattern abstractions support code reuse and code factoring, features that facilitate maintenance and evolution of code. Past research on pattern abstractions has generally ignored the aspect of compile-time checks for exhaustiveness and redundancy. In this paper we propose a class of expressive patterns that admits these compile-time checks.

## 1 Introduction

Patterns promote well-structured and readable code by combining matching and binding in a single syntactic "picture." Furthermore, patterns permit important sanity checks on case statements to be performed at compile time. The two commonly used checks verify that a set of patterns is *exhaustive*, that is, any value of the appropriate type is matched by some pattern, and detect whether any pattern is *redundant*, that is, it matches only values already matched by textually preceding patterns.

There are however three major shortcomings with the simple patterns found in languages like ML. First, these patterns are very restricted in terms of the sets of trees (forests) that they can describe. A pattern compares only a syntactically fixed number of initial nodes of each tree. Second, patterns cannot be used with abstract data types. Third, patterns cannot be named and reused. All problems hamper program development, maintenance, and evolution.

A number of proposals have addressed the first problem [8, 2, 6, 12], but without addressing the second one. Wadler's *views* [14] elegantly solve the second problem. However, whether something constitutes a view can in general not be checked by the compiler. Recently a very general solution to all three problems has been proposed by Palao Gostanza *et al* [11]. However, their patterns do not admit compile-time checking.

In this paper we propose a new class of expressive patterns that solve problems one and three. Patterns in this class can be modeled by finite-state tree recognizers. Hence, static checking reduces to the problem of containment between tree recognizers, a property which is decidable [7]. A forthcoming technical report will address problem two on integrating our patterns with abstract datatypes.

The paper is organized as follows. Section 2 motivates our approach with examples and identifies two syntactic restrictions that are sufficient to guarantee that patterns correspond to finite state tree recognizers. Section 3 defines the syntax and semantics of patterns. Section 4 describes a translation of our patterns to a simpler canonical form. Construction of tree recognizers for canonical patterns is straight-forward. Section 5 describes this construction, and how the result can be used to check for exhaustion and redundancy. Section 6 discusses a source translation of canonical patterns to Standard ML patterns. Finally, Section 7 describes related work and Section 8 contains our conclusions.

## 2 Motivation

Naming of constants and functions is a fundamental feature in any programming language. The two main benefits of naming are *factoring*, in which a concept is expressed in a single place, and *encapsulation*, in which a concept is implemented separately from its uses. Functional languages, such as ML, support naming of values but not naming of patterns. In this section, we argue the benefits of *named patterns* for the purpose of factoring and encapsulation, and motivate our extensions to patterns: alternation, recursion, and node creation.

Consider a datatype to represent forests[1]:

```
datatype forest = Node of { label : int,
                            left  : forest,
                            right : forest }
                | Union of forest list
```

*Effort partially sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

To appear in the 1997 ACM SIGPLAN International Conference on Functional Programming, June 9-11 1997, Amsterdam, Netherlands.

[1] For the sake of concreteness we will use ML syntax and terminology throughout the paper. But our results should carry over naturally to other call-by-value functional languages.

where empty forests are represented as an empty union:

```
val empty = Union []
```

ML permits datatype constructors (such as `Union`) to be used in patterns, but does not permit values (such as `empty`) to be used as patterns. Thus the fact that empty forests are represented using empty unions is explicit everywhere pattern matching is used. If the `forest` datatype is later changed by adding an explicit `Empty` constructor, all patterns involving `Union` may need to be changed as well.

On the other hand, suppose one can define a *pattern abstraction* such as:

```
pat Empty = Union [];
```

If this abstraction is used in patterns wherever one wishes to match the empty forest, it is trivial to accommodate the representation change by changing only the pattern definition. Pattern abstractions thus enable factoring.

In functional languages, we can not only name simple values, but also functions. Similarly, we need the power of pattern abstractions with parameters. Consider a datatype modeling types in Oberon 2, which has both fixed size arrays (with a constant integer bound) and open arrays (with a runtime bound):

```
datatype Type = ...
              | FixedArray of int * Type
              | OpenArray of Type;
```

Large parts of a compiler will treat the two array types the same, but may need the respective element types. Rather than having both cases wherever arrays are matched, one can use a single pattern `Array`, which is defined with two *alternatives*:

```
pat Array(elemtype) = FixedArray(_,elemtype)
                    | OpenArray(elemtype)
```

Alternative patterns are already provided in some versions of SML/NJ. We call them simply *or-patterns*. In order to be well-formed, each alternative must bind the same set of pattern variables (`elemtype` in this case).

When a pattern abstraction is used in a pattern, it is applied to actual parameter patterns. We call such uses *pattern applications* or pattern calls, or simply calls. The function `dimensions` given below uses the pattern `Array` to bind the variable `ty` to the element type of any array type, no matter what kind of array it is, fixed or open.

```
fun dimensions (Array(ty)) = dimensions(ty)+1
|   dimensions _ = 0;
```

One can think of a pattern application as being replaced by the body of the named pattern with the actual parameter patterns substituted for the formals. For simple patterns such as `Array`, this intuition is accurate, modulo renaming of bound variables.

The next example illustrates the use of recursive patterns. Consider a datatype for join lists:

```
datatype jlist = None | Single of int
               | Append of jlist * jlist
```

and consider writing a pattern that matches two-element join lists. Before we show a solution to this problem, consider the much simpler problem: how can the set of `jlist`'s representing empty lists be characterized? Of course `None` is empty, but so is the appending of two empty `jlist`'s. This intuition can be translated into the following pattern abstraction:

```
pat Nil = None | Append(Nil,Nil)
```

This pattern abstraction is *recursive* since it includes calls to itself. Recursive patterns can match arbitrarily many nodes in a tree. Using `Nil` we can define:

```
pat One(x) = Single x | Append(One(x),Nil)
           | Append(Nil,One(x))
```

Here we have an example of a recursive pattern abstraction with an argument. It matches any `jlist` with exactly one `Single` node in it, and binds `x` to the element of that node. Using `Nil` and `One`, we can write `Pair` so that it handles any `jlist` with exactly two elements:

```
pat Pair(x,y) = Append(Pair(x,y),Nil)
              | Append(One(x),One(y))
              | Append(Nil,Pair(x,y))
```

Thus a pair is always represented by an `Append` node and the two elements are either in the first subtree, spread between both subtrees, or are both in the second subtree.

Our third pattern extension—node creation—is motivated by the complement to the `Nil` pattern, namely a pattern `Cons` that matches the first element of a `jlist` and also binds a variable to a `jlist` holding the rest of the elements. The difficulty is that there may be no node in the structure being matched that represents the rest of the elements. For example, in the tree

```
Append(Append(Single 1, Single 2), Single 3)
```

there is no subtree holding exactly `2` and `3`. More trivially, the value `Single 1` contains no "empty list" to be bound to the "rest" variable. To solve this dilemma, we introduce a limited form of expressions into patterns, which permit bindings to be augmented with newly created nodes. We can now define `Cons` as follows:

```
pat Cons(x,l) = Single x where l = None
              | Append(Nil,Cons(x,l))
              | Append(Cons(x,l1),l2)
                  where l = Append(l1,l2)
```

The first alternative matches a `Single` node and binds `l` (the "rest" variable) to a newly constructed empty `jlist`. The second alternative handles the case when all of the elements occur in the right subtree, in which case a simple recursive call is used. The third alternative matches `Append` nodes with at least one element in the left subtree. Here `l` must be bound to a `jlist` holding the rest of the elements from the left subtree (available through recursion) and all of the elements of the right subtree. The `jlist` is constructed using `Append`.

### 2.1 Restrictions

The syntax as we have outlined in this section permits overly powerful patterns. For instance, Pedro Palao Gostanza has shown in private communication that the halting problem for Turing machines can be reduced to checking whether a set of (unrestricted) patterns is exhaustive. If we restrict patterns to match *recognizable forests*, compile-time checks

are decidable. *Recognizable forests* are defined to be the forests for which there exist finite-state bottom-up tree recognizers [7]. Below we give examples of two classes of patterns that match *non-recognizable* forests. It turns out that if we syntactically restrict our patterns to avoid these classes, then we can always construct finite tree recognizers for them, which in turn enables the desired compile-time checks.

An example of the first class of patterns to avoid is `Cnt` defined by

```
datatype X = A of X | B of X | C of X | D
pat Cnt(x) = C(x) | A(Cnt(B(x)))
```

The pattern `Cnt(_)` matches the set of trees of the form $\texttt{A}^n\texttt{(C(B}^n\texttt{(\_)))}$, which is not recognizable by a *finite* tree recognizer. The characterizing syntactic property of this pattern is that the recursive call to `Cnt` contains a non-trivial argument pattern `B(x)`. Next consider the pattern `PowerOf2` defined by

```
datatype Nat = Z | S of Nat
pat Even(half) = Z where half = Z
               | S(S(Even(x)))
                   where half = S(x)
pat PowerOf2 = S(Z) | Even(PowerOf2)
```

The pattern `PowerOf2` only matches trees of the form $\texttt{S}^n\texttt{(Z)}$, where $n$ is a power of 2. This set cannot be recognized by a finite tree recognizer. Here the characterizing property is the recursive call to `PowerOf2` as a pattern argument to `Even`.

In general, the two syntactic properties that may cause patterns to match non-recognizable forests are:

1. Non-atomic (non-variable, non-wild card) pattern arguments to recursive calls.
2. Recursion nested within a call to a named pattern.

We can always construct finite tree recognizers for patterns that adhere to these restrictions (Sections 4 and 5). Weaker restrictions are possible but would have made this paper more complex.

## 3 Patterns

In this section, we describe the syntax and semantics of patterns. The static semantics places our extensions in the context of the ML type system.

### 3.1 Syntax

Figure 1 shows the abstract syntax for patterns, and pattern definitions. ML patterns are extended with pattern definitions, or-patterns, pattern calls, node creation, and general as-patterns. We assume a set of variable names $x \in \mathit{Vars}$ and a set of pattern names $f \in \mathit{Funcs}$. Constructors $c \in \mathit{Cons}$ have fixed arity and type $\mathit{typeof}(c)$. Nullary constructors and patterns are written $c()$ and $f()$ respectively, although in examples, we drop the extra (). Node creation is limited to constructor applications and variables. Pattern declarations are sets of mutually recursive pattern definitions.

Syntactic restrictions on patterns are listed below:

1. Patterns must be linear (no variable may be bound twice).

| | | | |
|---|---|---|---|
| (atomic pattern) | $a$ | ::= | `_` $\mid x$ |
| (pattern) | $p$ | ::= | $a$ |
| (constructor) | | $\mid$ | $c\,(p_1, \ldots, p_n)$ |
| (pattern call) | | $\mid$ | $f\,(p_1, \ldots, p_n)$ |
| (as-pattern) | | $\mid$ | $p_1$ `as` $p_2$ |
| (or-pattern) | | $\mid$ | $p_1$ `\|` $p_2$ |
| (where clause) | | $\mid$ | $p$ `where` $x_1$ `=` $s_1$ `and` $x_2$ `=` $s_2$ ... |
| (creation) | $s$ | ::= | $x \mid c\,(s_1, ..., s_n)$ |
| (declaration) | $dec$ | ::= | ... |
| | | $\mid$ | `pat` $f_1$`(`$x_{11}$`,...,`$x_{1n_1}$`)` `=` $p_1$ `and` $f_2$`(`$x_{21}$`,...,`$x_{2n_2}$`)` `=` $p_2$ ⋮ |

Figure 1: The syntax of patterns.

2. Each alternative in an or-pattern must bind the same set of variables.
3. Variable bindings must be used exactly once, i.e., every variable occurring in a pattern is either a formal parameter or used exactly once in a `where`-clause to create a new node. Unused variables must be replaced by `_`.
4. Arguments to recursive calls must be atomic (Avoids the `Cnt` example of Section 2).
5. No recursive calls in pattern arguments (Avoids the `PowerOf2` example).
6. No cycles in the top-level call graph among mutually recursive patterns.

Restrictions 1 and 2 are standard. Restriction 3 makes the technical material in the rest of the paper simpler, and restrictions 4 and 5 guarantee that we can form finite tree recognizers for each pattern. Restriction 6 disallows non-terminating patterns, such as

```
pat Bottom(x) = Bottom(x)
```

Non-terminating patterns cause non-termination in the implementation as well as in the translation given in the next section. They furthermore break the correspondence between the operational meaning of a patterns and its tree recognizer.

### 3.2 Static Semantics

We give a set of typing rules that extend ML's type system [10] for patterns and declarations. The type rules use type environments $TE$ which are finite maps from variables $\mathit{Vars}$ to types $\tau$, and pattern environments $F$, which are finite maps from pattern names $\mathit{Funcs}$ to types $\tau$. We write the union of two environments with disjoint domains as $TE_1 + TE_2$. Figure 2 contains three kinds of judgments:

- $F \vdash_{\mathrm{p}} p : (TE, \tau)$ states that in pattern environment $F$, $p$ matches values of type $\tau$ and binds each variable $x$ in the domain of $TE$ to a value of type $TE(x)$.
- $TE \vdash_{\mathrm{s}} s : (\tau, U)$ states that in environment $TE$, the expression $s$ has type $\tau$ and uses variables $U$.

- $F \vdash_{\mathrm{d}} d : F'$ states that declaration $d$ extends environment $F$ to $F'$.

Most rules are straight-forward, hence we only describe the [WHERE] rule. Intuitively, a pattern $p$ `where` $x_i = s_i$ matches whatever $p$ matches, and introduces extra bindings for $x_i$, defined by $s_i$. The constructs $s_i$ must use variable bindings produced by $p$. We therefore type $s_i$ in the type environment $TE$ produced by $p$. Because we want each binding to be used only once for simplicity, we remove the bindings $U_i$ used by $s_i$ from $TE$ in the resulting environment (written $TE \backslash_{U_i}$).

### 3.3 Dynamic Semantics

In Section 6, we define the semantics with a source-level translation to Standard ML. Here we informally contrast two possible match-semantics, *local match* vs. *global match*. Local and global match differ in the way pattern parameters are handled. We illustrate the distinction by means of an example. Consider a pattern call

```
Elem(3)
```

to some pattern abstraction `Elem`. With local match semantics, `Elem(3)` matches a tree $t$, if `Elem(x)` matches $t$, and 3 matches $x$. Global match on the other hand expands the pattern definition of `Elem`, by substituting the actual argument pattern 3 for the formal. The resulting pattern is then used to match $t$. The two approaches yield different results if or-patterns are involved at some level. If the definition of `Elem` is as below, then `Elem(x)` can match the tree $t =$ `Append(Single(2),Single(3))` in two different ways.

```
pat Elem(x) = Single(x)
            | Append(Elem(x),_)
            | Append(_,Elem(x))
```

If the second alternative is used, then `x` is bound to 2, if the third is used, `x` is bound to 3. Since we desire a deterministic semantics, we follow the tradition of ML by choosing the "first match" with alternatives being considered left to right.

Now consider again the local match vs. global match distinction. Using local match, `Elem(3)` does not match $t$, since `Elem(x)` commits to the second branch without considering the pattern argument 3. Using global match, `Elem(3)` matches $t$, since we expand the alternatives with the argument pattern. In our example we obtain

```
  Single(3)
| Append(Elem(3),_)
| Append(_,Elem(3))
```

and it is now clear that the second branch cannot match (after one more expansion). Global match only commits to a branch if it matches the input tree w.r.t. the given argument patterns.

Local matching enables a simple implementation without backtracking. This choice of semantics is used in Palao Gostanza *et al*'s active destructors [11]. However, as we describe in Sections 4 and 6, the more powerful global match semantics can also be efficiently implemented. Hence, since global match is more expressive and may be more intuitive, we chose global match semantics for our patterns.

$$F \vdash_{\mathrm{p}} \_ : ([], \tau) \qquad \text{[WILD]}$$

$$F \vdash_{\mathrm{p}} x : ([x \mapsto \tau], \tau) \qquad \text{[PVAR]}$$

$$\frac{\begin{array}{l} F \vdash_{\mathrm{p}} p_i : (TE_i, \tau_i) \\ \mathrm{dom}(TE_i) \cap \mathrm{dom}(TE_j) = \emptyset \quad \text{forall } i \neq j \in 1 \ldots n \\ TE = TE_1 + \cdots + TE_n \\ \mathit{typeof}(c) = \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau \end{array}}{F \vdash_{\mathrm{p}} c\,(p_1, \ldots, p_n) : (TE, \tau)} \qquad \text{[PCON]}$$

$$\frac{\begin{array}{l} F \vdash_{\mathrm{p}} p_i : (TE_i, \tau_i) \\ \mathrm{dom}(TE_i) \cap \mathrm{dom}(TE_j) = \emptyset \quad \text{forall } i \neq j \in 1 \ldots n \\ TE = TE_1 + \cdots + TE_n \\ F(f) = \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau \end{array}}{F \vdash_{\mathrm{p}} f\,(p_1, \ldots, p_n) : (TE, \tau)} \qquad \text{[PAPP]}$$

$$\frac{\begin{array}{l} F \vdash_{\mathrm{p}} p_1 : (TE_1, \tau) \\ F \vdash_{\mathrm{p}} p_2 : (TE_2, \tau) \\ \mathrm{dom}(TE_1) \cap \mathrm{dom}(TE_2) = \emptyset \end{array}}{F \vdash_{\mathrm{p}} p_1 \texttt{ as } p_2 : (TE_1 + TE_2, \tau)} \qquad \text{[AS]}$$

$$\frac{\begin{array}{l} F \vdash_{\mathrm{p}} p_1 : (TE, \tau) \\ F \vdash_{\mathrm{p}} p_2 : (TE, \tau) \end{array}}{F \vdash_{\mathrm{p}} p_1 \mid p_2 : (TE, \tau)} \qquad \text{[OR]}$$

$$\frac{\begin{array}{l} F \vdash_{\mathrm{p}} p : (TE, \tau) \\ TE \vdash_{\mathrm{s}} s_i : (\tau_i, U_i) \\ x_i \notin \mathrm{dom}(TE) \\ U_i \subseteq \mathrm{dom}(TE) \\ x_i \neq x_j, U_i \cap U_j = \emptyset \quad \text{forall } i \neq j \\ TE' = TE \backslash_{U_1} \ldots \backslash_{U_n} \end{array}}{F \vdash_{\mathrm{p}} p \texttt{ where } x_1 \texttt{ = } s_1 \texttt{ ... and } x_n \texttt{ = } s_n : (TE'[x_i \mapsto \tau_i], \tau)} \qquad \text{[WHERE]}$$

$$TE[x \mapsto \tau] \vdash_{\mathrm{s}} x : (\tau, \{x\}) \qquad \text{[SVAR]}$$

$$\frac{\begin{array}{l} TE \vdash_{\mathrm{s}} s_i : (\tau_i, U_i) \\ U_i \cap U_j = \emptyset \quad \text{forall } i \neq j \in 1 \ldots n \\ U = U_1 \cup \cdots \cup U_n \\ \mathit{typeof}(c) = \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau \end{array}}{TE \vdash_{\mathrm{s}} c\,(s_1, \ldots, s_n) : (\tau, U)} \qquad \text{[SCON]}$$

$$\frac{\begin{array}{l} \tau_i = \tau_{i1} \rightarrow \cdots \rightarrow \tau_{in_i} \rightarrow \tau_{i0} \\ F' = F[f_i \mapsto \tau_i] \\ F' \vdash_{\mathrm{p}} p_i : (TE_i, \tau_{i0}) \\ \mathrm{dom}(TE_i) = \{x_{i1}, \ldots, x_{in_i}\} \\ TE_i(x_{ik}) = \tau_{ik} \quad k = 1, \ldots, n_i \end{array}}{F \vdash_{\mathrm{d}} \ldots f_i\texttt{(}x_{i1}\texttt{,...,}x_{in_i}\texttt{) = } p_i \texttt{ and ...} \quad : F'} \qquad \text{[PAT]}$$

Figure 2: Type rules for patterns.

## 4 Canonical Patterns

This section describes the crux of the paper, a translation from the patterns we defined in Section 3 to a simpler canonical form. A pattern is *canonical* if the pattern arguments in any pattern call are variables. In terms of the grammar in Figure 1, the only change is in the production for pattern calls, which becomes

$$\text{(pattern call)} \qquad | \; f\ (x_1, \ldots, x_n)$$

Non-atomic pattern arguments account for nearly all the complexity (and the expressiveness) of the patterns. As we will see in Sections 5 and 6, construction of tree automata and implementation of canonical patterns is almost trivial. Moreover, the distinction between local and global match semantics discussed in Section 3.3 becomes irrelevant for canonical patterns. All the potential backtracking, and matching against node creations is handled by the translation to canonical patterns.

To reduce a pattern to canonical form, we essentially perform the expansion described w.r.t. the global match semantics, i.e. substitution of argument patterns for formals in the body of pattern definitions. Due to recursion, the expansion may however be infinite. The following observation gives some intuition for why a finite partial expansion is sufficient:

> In the infinite expansion of a given pattern, every pattern abstraction is only called with finitely many distinct argument patterns.

This fact follows immediately from our syntactic restriction that argument patterns to recursive calls must be atomic. Thus in the expansion, argument patterns to recursive calls are either `_`, or sub-patterns of the original pattern arguments. The possibility of sub-patterns comes from `where`-patterns, since they bind new variables to sub-patterns of arguments.

Another way to look at the expansion is as a specialization of pattern abstractions to all contexts they appear in. Again, the syntactic restrictions guarantee that there are only finitely many specializations.

The canonicalization works as follows: for each pattern call with non-atomic argument patterns, we create a new pattern definition that takes as arguments the free variables of the original argument list. The body of the new definition is obtained by substitution of the arguments into the body of the pattern definition being called. Finally, the original call is replaced with a call to the new pattern definition with arguments being the free variables of the original pattern arguments.

As an example, the pattern call `Cons(3,Cons(y,_))` is replaced by a call to a specialized version of `Cons`, namely `Cons3ConsY_(y)`. Assuming the definition of `Cons` given earlier, Figure 3 shows the specializations involved.

Besides recursion, specialization is complicated by `where`-patterns. To handle node creation, we essentially partially evaluate the pattern matching against created nodes. This evaluation is possible due to the absence of recursion in argument patterns and cycles in top-level call graphs (restrictions 5 and 6 in Section 3.1). Due to the matching performed during translation, `fail` patterns (which never match) may be introduced. `Fail` patterns can always be removed from canonical patterns, but it is simpler to deal with them during the automata construction and the source translation to ML.

```
pat Cons3ConsY_(y) = Append(Nil(),Cons3ConsY_(y))
                   | Append(Cons3ConsY_(y),_)
                   | Append(Cons3Nil(),ConsY_(y))
and Cons3Nil() = Single(3)
               | Append(Nil(),Cons3Nil())
               | Append(Cons3Nil(),Nil())
and ConsY_(y) = Single(y)
              | Append(Nil(),ConsY_(y))
              | Append(ConsY_(y),_)
```

Figure 3: Canonicalization of `Cons(3,Cons(y,_))`.

A cache of ongoing substitutions is used to limit the number of specializations to a finite number. The details of the translation are given in Appendix A. While the worst case expansion of patterns during the translation is exponential, we believe the algorithm will prove tractable for normal cases. For example, patterns using chains of `Cons` patterns yield cubic-size canonical patterns.

## 5 From Patterns to Tree Automata

In this section, we show how to construct non-deterministic bottom-up tree automata for patterns. The problems of exhaustion and redundancy then reduce to deciding inclusion between regular tree languages.

### 5.1 Definitions

A non-deterministic bottom-up tree automaton is a triple $\Delta = (A, \delta, F)$ where $A$ is a set of states, $\delta$ is a set of transitions of the forms $c(a_1, .., a_n) \longrightarrow a$, $(a_1, a_2) \xrightarrow{\wedge} a$ or $a_1 \xrightarrow{\epsilon} a_2$, where $c$ is an $n$-ary constructor from *Cons*, and $a_1, \ldots, a_n, a$ are states from $A$. The set $F \subseteq A$ is the set of accepting states.

An *epsilon* transition $a_1 \xrightarrow{\epsilon} a_2$ states that the automaton can enter state $a_2$ if it can enter state $a_1$. An *and* transition $(a_1, a_2) \xrightarrow{\wedge} a$ states that the automaton can enter state $a$ if it can enter both $a_1$ and $a_2$. In order to capture the semantics of these transitions, we define the *closure* of a set of states $A$ w.r.t. transitions $\delta$, (denoted $\text{closure}_\delta(A)$) as the smallest set satisfying

$$\begin{aligned} a \in \text{closure}_\delta(A) \quad \Leftrightarrow \quad & a \in A \vee \\ & a' \in \text{closure}_\delta(A) \wedge a' \xrightarrow{\epsilon} a \in \delta \vee \\ & a', a'' \in \text{closure}_\delta(A) \wedge (a', a'') \xrightarrow{\wedge} a \in \delta \end{aligned}$$

Given a tree $t$ over *Cons*, a *run* of $\Delta$ is an assignment of subsets of $A$ to each subtree in $t$ (written $\Delta(t)$) defined in a bottom-up fashion: $\Delta(c(t_1, \ldots, t_n)) = \text{closure}_\delta(\{a \mid a_i \in \Delta(t_i), c(a_1, \ldots, a_n) \longrightarrow a \in \delta\})$. A run is successful if one of the states at the root of the tree $t$ is in $F$ ($\Delta(t) \cap F \neq \emptyset$). $\Delta$ accepts all trees for which there is a successful run.

An automaton $\Delta = (A, \delta, F)$ is *deterministic* if $\delta$ contains no $\epsilon$ or $\wedge$ transitions and moreover, when it contains two transitions for the same constructor $c(a_1, .., a_n) \longrightarrow a \in \delta$ and $c(a_1, .., a_n) \longrightarrow a' \in \delta$ then $a = a'$. In a run of a deterministic automaton, all the sets $A(t)$ have at most one element. We include $\epsilon$ and $\wedge$ transitions in our automata to make the construction simpler. Such transitions can always be eliminated. Appendix B shows how a nondeterministic

automaton can be transformed into a deterministic automaton.

### 5.2 Construction

Given a canonical pattern $p$, we show how to construct a tree-automaton $\Delta_p$ recognizing the same forest as $p$. Since canonical pattern definitions are only called with pattern variables, parameters are only used for binding and do not affect pattern matching. As a result, we can ignore parameters during the automata construction.

Let $a_p$ be a (unique) state for every sub-pattern $p$ in the program, and $a_f$ be a (unique) state for every pattern definition $f$. Furthermore, let $a_\tau$ be a (unique) state for every type $\tau$ used in the program. Let $A^*$ be the complete set of all these states. In the following discussion, we assume each $\tau$ is a monomorphic datatype. We believe our results can be extended to polymorphic types.

We define a transition set $\delta^*$ among the states $A^*$. First, for each datatype $\tau$ declared as follows:

$$\texttt{datatype } \tau \texttt{ = } c_1 \texttt{ of } \tau_{11}\texttt{*}\ldots\texttt{*}\tau_{1m_1} \texttt{ | } \ldots \texttt{ | } c_n \texttt{ of } \tau_{n1}\texttt{*}\ldots\texttt{*}\tau_{nm_n}$$

we add edges

$$\{c_1(a_{\tau_{11}},\ldots,a_{\tau_{1m_1}}) \longrightarrow a_\tau, \ldots, c_n(a_{\tau_{n1}},\ldots,a_{\tau_{nm_n}}) \longrightarrow a_\tau\}$$

Next for every pattern definition $\texttt{pat } f(\ldots) = p$, we add the edge $a_p \xrightarrow{\epsilon} a_f$.

Then for each kind of pattern $p$ of inferred type $\tau$, we add additional edges:

| | |
|---|---|
| _ | $a_\tau \xrightarrow{\epsilon} a_p$ |
| x | $a_\tau \xrightarrow{\epsilon} a_p$ |
| fail | |
| $c(p_1,\ldots,p_n)$ | $c(a_{p_1},\ldots,a_{p_n}) \longrightarrow a_p$ |
| $f(x_1,\ldots,x_n)$ | $a_f \xrightarrow{\epsilon} a_p$ |
| $p_1$ as $p_2$ | $(a_{p_1}, a_{p_2}) \xrightarrow{\wedge} a_p$ |
| $p_1$ \| $p_2$ | $a_{p_1} \xrightarrow{\epsilon} a_p, a_{p_2} \xrightarrow{\epsilon} a_p$ |
| $p'$ where ... | $a_{p'} \xrightarrow{\epsilon} a_p$ |

The automaton for any pattern $p$, $\Delta_p$ is $(A^*, \delta^*, \{a_p\})$. Of course, many of the states and edges will be irrelevant to a run which must eventually contain $a_p$. One can define $A^*_p$ to be those states from which $a_p$ is reachable, and $\delta^*_p$ to be $\delta^*$ restricted to this set and then define $\Delta_p$ as $(A^*_p, \delta^*_p, \{a_p\})$.

### 5.3 Checking Exhaustion and Redundancy

We can decide exhaustiveness of a set of patterns $(p_i)$ of type $\tau$ by forming the union of the automata $\Delta_{p_i}$, and verifying that $(A^*_\tau, \delta^*_\tau, \{a_\tau\}) \subseteq \bigcup_i \Delta_{p_i}$. For redundancy, one can test whether $\Delta_p \subseteq \bigcup_i \Delta_{p_i}$ holds, in which case $p$ is redundant w.r.t. $(p_i)$. With the transformation to deterministic bottom-up tree automata in Appendix B, these relations are all decidable [7]. The complexity of the decision procedure is worst-case exponential in the size of the canonical patterns (due to the subset construction for deterministic automata).

## 6 Implementation

Each pattern abstraction $f$ is implemented as a function $f$ that takes a tree and either returns bindings for the parameters, or raises a reserved exception `Fail`. For a pattern definition of the form

$$\texttt{pat } f(x_1,\ldots,x_n) \texttt{ = } p$$

we generate a function definition of the form

$$\texttt{fun } f\texttt{(node) = } \mathcal{T}_p\,[(p,\texttt{node})]\,(x_1,\ldots,x_n)$$

As an example, Figure 4 shows how the pattern definition `Cons3ConsY` from Section 4 is translated.

```
fun Cons3ConsY_(node) =
   (case node
    of Append(v1,v2) =>
        (let () = Nil(v1)
         in let (y) = Cons3ConsY_(v2)
            in (y))
    |  _ => raise Fail)
   handle Fail =>
      (case node
       of Append(v1,v2) =>
           (let (y) = Cons3ConsY_(v1)
            in (y))
        |  _ => raise Fail)
      handle Fail =>
         (case node
          of Append(v1,v2) =>
              (let () = Cons3Nil(v1)
               in let (y) = ConsY_(v2)
                  in (y))
           |  _ => raise Fail)
```

Figure 4: Translation of `Cons3ConsY_` from a pattern to a function.

Figure 5 gives the translation of patterns to Standard ML. The translation function $\mathcal{T}_p$ for patterns takes a list of pattern-variable pairs and code to be generated if the match succeeds. A pair $(p, v)$ represents a match of $v$ against $p$. At runtime, $v$ will be bound to the tree to be matched against $p$. In the translation for or-patterns we catch the `Fail` exception and try the next alternative.
To implement an expression of the form

$$\begin{array}{l}\texttt{case } e \texttt{ of}\\ \quad p_1 \texttt{ => } e_1\\ \quad \ldots\\ \texttt{ | } p_n \texttt{ => } e_n\end{array}$$

we rewrite it to

$$\begin{array}{l}\texttt{let } v \texttt{ = } e \texttt{ in}\\ \quad (\mathcal{T}_p\,[(p_1, v)]\,e_1)\\ \quad \texttt{handle Fail => } (\mathcal{T}_p\,[(p_2, v)]\,e_2)\\ \quad \ldots\\ \quad \texttt{handle Fail => } (\mathcal{T}_p\,[(p_n, v)]\,e_n)\\ \quad \texttt{handle Fail => raise Match}\end{array}$$

The simple translation given here does not make use of the many sophisticated techniques for improving the efficiency of pattern-matching, such as jump tables, or Sestoft's technique for using information known from previous matches [13]. Adapting these techniques to handle recursive pattern definitions is one interesting area for further research.

$$
\begin{array}{rcl}
\mathcal{T}_p & : & (p \times \mathit{Vars})^* \longrightarrow \mathit{Exp} \longrightarrow \mathit{Exp} \\
\mathcal{T}_p\,[]\,e & = & e \\
\mathcal{T}_p\,[(\_, v) : r]\,e & = & \mathcal{T}_p\,r\,e \\
\mathcal{T}_p\,[(x, v) : r]\,e & = & \mathcal{T}_p\,r\,[\![\texttt{let } x \texttt{ = } v \texttt{ in } e]\!] \\
\mathcal{T}_p\,[(\texttt{fail}, v) : r]\,e & = & [\![\texttt{raise Fail}]\!] \\
\mathcal{T}_p\,[(c(p_1, \ldots, p_n), v) : r]\,e & = & [\![\texttt{case } v \\
 & & \quad \texttt{of } c(v_1, \ldots, v_n) \texttt{ => } \mathcal{T}_p\,[(p_1, v_1) : \ldots : (p_n, v_n) : r]\,e \\
 & & \quad \texttt{| \_ => raise Fail}]\!] \\
 & & (v_i \text{ fresh}) \\
\mathcal{T}_p\,[(f(x_1, \ldots, x_n), v) : r]\,e & = & [\![\texttt{let } (x_1, \ldots, x_n) \texttt{ = } f(v) \\
 & & \quad \texttt{in } (\mathcal{T}_p\,r\,e)]\!] \\
 & & (v_i \text{ fresh}) \\
\mathcal{T}_p\,[(p_1 \texttt{ as } p_2, v) : r]\,e & = & \mathcal{T}_p\,[(p_1, v) : (p_2, v) : r]\,e \\
\mathcal{T}_p\,[(p_1 \texttt{ | } p_2, v) : r]\,e & = & [\![(\mathcal{T}_p\,[(p_1, v) : r]\,e) \texttt{ handle Fail => } (\mathcal{T}_p\,[(p_2, v) : r]\,e)]\!] \\
\mathcal{T}_p\,[(p \texttt{ where } x_1 \texttt{ = } s_1 \texttt{ and } \ldots \texttt{ and } x_n \texttt{ = } s_n, v) : r]\,e & = & \mathcal{T}_p\,[(p, v) : r]\,[\![\texttt{let } x_1 \texttt{ = } s_1 \texttt{ and } \ldots \texttt{ and } x_n \texttt{ = } s_n \texttt{ in } e]\!]
\end{array}
$$

Figure 5: Translation of canonicalized patterns to Standard ML.

## 7 Related Work

A class of work extends Hoffman and O'Donnell's [9] simple matchers with complex operations such as the subtree operator of Trafola [8], the vertical iterator of Dora [2, 6], and Queinnec and Geffroy's recursive tree operator [12]. All these extensions match recognizable forests, but do not address pattern abstraction per se.

Aitken and Reppy's *abstract value constructors* (AVC) [1] form a subclass of our named patterns without alternation, recursion or node creation. But unlike our patterns, some AVCs can also be used as values. The authors do not address static checking of AVCs.

Wadler's *views* [14] define alternative, free data types that are isomorphic to the underlying representation. The isomorphism is described using `in` and `out` functions with general computation. Views naturally admit exhaustiveness and redundancy checks, but whether something constitutes a view can in general not be checked by the compiler. In contrast to our patterns, views can also be used as values. Burton and Cameron [4] drop views as values and obtain a system similar to ours. However, their implementation requires translation via the `in` function from the underlying data type to the view before matching can be performed. Such a translation is not necessary in our framework.

Palao Gostanza *et al* [11] propose pattern abstractions that completely separate patterns from data types, Boyland [3] implements named patterns with recursion using first-match semantics, and Erwig has a proposal for *active patterns* [5] in which patterns are parameterized by values as well as patterns. All of these mechanisms permit arbitrary computation in patterns and thus cannot be statically analyzed.

## 8 Conclusions

We have described an extension to ML: pattern abstractions with recursion, alternation, and node creation. Patterns using these abstractions can be checked at compile time for exhaustiveness and redundancy using finite-state tree recognizers. Implementation of the patterns is based on a simplifying substitution that partially evaluates the patterns against nodes created in patterns. The result of the substitution can be translated in a straight-forward manner to a set of ML patterns and ML functions.

The size of the translation is worst-case exponential in the size of the original pattern definitions, and the deterministic automata needed for the compile time checking may be doubly-exponential. While the theoretical complexity looks discouraging, we believe—based on experience with a prototype implementation—that these worst-case bounds are only met for contrived examples.


### Acknowledgments

We thank Alex Aiken, Chris Okasaki, Pedro Palao Gostanza, David Gay, Zhendong Su, and the anonymous referees for their helpful and insightful comments on earlier drafts of the paper.


### References


[1] William E. Aitken and John H. Reppy. Abstract value constructors. In *ACM SIGPLAN Workshop on ML and its Applications*, pages 1–11. 1992.

[2] John Boyland, Charles Farnum, and Susan L. Graham. Attributed transformational code generation for dynamic compilers. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques. Workshops in Computer Science*, pages 227–254. Springer-Verlag, Berlin, 1992.

[3] John Tang Boyland. *Descriptional Composition of Compiler Components.* PhD thesis, University of California, Berkeley, 1996. Available as technical report UCB//CSD-96-916.

[4] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, April 1993.

[5] Martin Erwig. Active patterns. In *8th International Workshop on the Implementation of Function Languages*. 1996.

[6] Charles Farnum. *Pattern-based languages for prototyping compiler optimizers*. PhD thesis, Computer Science Division—EECS, University of California, Berkeley, December 1990. Available as technical report UCB//CSD-90-608.

[7] Ferenc Gécseg and Magnus Steinby. *Tree Automata*. Akadémiai Kiadó, Budapest, 1984.

[8] Reinhold Heckmann. A functional language for the specification of complex tree transformations. In Harald Ganzinger, editor, *European Symposium on Programming (ESOP '88)*, volume 300 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, Berlin, 1988.

[9] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[10] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.

[11] Pedro Palao Gostanza, Ricardo Peña, and Manuel Núñez. A new look at pattern matching in abstract data types. In *Proceedings of the ACM SIGPLAN Internation Conference on Functional Programming (ICFP '96)*, *ACM SIGPLAN Notices*, 31(6):110–121, 1996.

[12] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to symbolic pattern matching with intelligent backtrack. In *Workshop for Static Analysis*. October 1992.

[13] P. Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Gluck, and P. Thiemann, editors, *Partial Evaluation. International Seminar. Selected Papers*, pages 446–464. Springer-Verlag, Berlin, February 1996.

[14] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 307–313. ACM Press, New York, January 1987.

## A Reduction to Canonical Patterns

This appendix describes the translation of patterns to canonical form in detail. We translate a mutually recursive group of pattern definitions together. We assume that patterns occurring in case expressions are first factored out into pattern definitions so that patterns outside pattern definitions are canonical. Hence, we only have to deal with pattern definitions. In the course of the translation process, we generate new pattern definitions, these are added to the set of mutually recursive definitions currently being analyzed.

Figure 7 defines four functions that perform the translation at compile-time:

$\mathcal{S}_p$ This function takes a cache of specializations, an environment and a pattern. The environment binds parameter names to patterns. The function returns a canonical pattern, one in which all calls to pattern definitions have only pattern variables for arguments. Translation starts by calling this function with an empty cache and an identity environment for free pattern variables.

$\mathcal{S}_w$ This function takes a node expression (from a `where` clause) and a pattern. It returns a set of pairs: each an environment and a set of bindings of variables to node expressions.

$\mathcal{S}_s$ This function takes a set of bindings of variables to node expressions, and a single node expression. It returns a substitution of the latter using the given bindings.

$\mathcal{S}_f$ This function takes the cache of specializations, and a pattern call. It returns a pattern call with only the free variables as parameters.

We also make use of an auxiliary function:

**free** This function (not shown) returns the free variables of a pattern or used variables of a node expression.

We use some operations on environments and patterns. These operations are defined in Figure 6:

$+$ This operation joins two environments together. If they each provide a binding for the same pattern variable, a conjunction of the two bindings is produced. (Here we require unrestricted `as` patterns.)

$\oplus$ This operation joins pairs of an environment and a set of node bindings. It is extended to operate on sets of such pairs. We use it to combine results of compile-time pattern matching against node creations.

$\vee$ This operation joins patterns together in alternation. If the set of patterns to join is empty, it returns the special pattern `fail` that doesn't match anything.

## B Nondeterministic to Deterministic Automata

Given a nondeterministic tree automaton, as defined in Section 5, we show that an equivalent deterministic automaton can be created. If we did not include 'and' transitions in our nondeterministic automata, we could have referred to Gécseg and Steinby's standard definition of tree automata [7] for the proof.

Let $\Delta = (A, \delta, F)$ be a (nondeterministic) tree automaton. We construct a deterministic automaton $\Delta' = (A', \delta', F')$ as follows:

$$\begin{aligned} A' &= \mathcal{P}(A) \\ F' &= \{a' \in A' \mid a' \cap F \neq \emptyset\} \end{aligned}$$

with a transition in $\delta'$ defined for every $c \in \mathit{Cons}$ with arity $n$ and every $n$-tuple $(a'_1, \ldots, a'_n) \in A'^n$:

$$c(a'_1, \ldots, a'_n) \longrightarrow \mathrm{closure}_\delta(\{a \mid a_i \in a'_i, c(a_1, \ldots, a_n) \longrightarrow a \in \delta\})$$

By construction, we have therefore that $\Delta'$ is deterministic and moreover, runs of the two automata are closely related:

$$\forall_t \quad \Delta'(t) = \{\Delta(t)\}$$

Furthermore, $\Delta$ accepts $t$ if and only if $\Delta(t) \cap F \neq \emptyset$ if and only if $\Delta(t) \in F'$. Thus we see the two automata accept the same tree language.

$$
\begin{array}{rcl}
(e + e')(x) & = & e(x) \ \texttt{as} \ e'(x) \quad e(x), e'(x) \text{ defined} \\
(e + e')(x) & = & e(x) \quad e(x) \text{ defined} \\
(e + e')(x) & = & e'(x) \quad e'(x) \text{ defined}
\end{array}
$$

$$
\begin{array}{rcl}
(e, w) \oplus (e', w') & = & (e + e', w \cup w') \\
\{(e_i, w_i) \mid 0 < i \leq n\} \oplus \{(e'_j, w'_j) \mid 0 < j \leq m\} & = & \{(e_i, w_i) \oplus (e'_j, w'_j) \mid 0 < i \leq n, 0 < j \leq m\} \\
\bigoplus \{\} & = & \{([], \{})\}
\end{array}
$$

$$
\begin{array}{rcl}
\bigvee \{\} & = & \texttt{fail} \\
\bigvee \{p\} & = & p \\
\bigvee \{p_0, p_1, \ldots, p_n\} & = & p_0 \ \mid \ \bigvee \{p_1, \ldots, p_n\}
\end{array}
$$

Figure 6: Definitions of $+$, $\oplus$, and $\vee$

$$
\begin{array}{rcl}
\mathit{Env} & = & \mathit{Vars} \mapsto p \\
\mathit{Bindings} & = & \mathcal{P}(\mathit{Vars} \times s) \\
\mathit{Cache} & = & (\mathit{Funcs} \times (p \times \ldots \times p)) \mapsto \mathit{Funcs}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}_p & : & \mathit{Cache} \to \mathit{Env} \to p \to p \\
\mathcal{S}_p\, C\, e\, \_ & = & \_ \\
\mathcal{S}_p\, C\, e\, x & = & e(x) \\
\mathcal{S}_p\, C\, e\, \texttt{fail} & = & \texttt{fail} \\
\mathcal{S}_p\, C\, e\, c(p_1, \ldots, p_n) & = & c(\mathcal{S}_p\, C\, e\, p_1, \ldots, \mathcal{S}_p\, C\, e\, p_n) \\
\mathcal{S}_p\, C\, e\, f(p_1, \ldots, p_n) & = & \mathcal{S}_f\, C\, f(\mathcal{S}_p\, C\, e\, p_1, \ldots, \mathcal{S}_p\, C\, e\, p_n) \\
\mathcal{S}_p\, C\, e\, (p_1 \mid p_2) & = & \mathcal{S}_p\, C\, e\, p_1 \mid \mathcal{S}_p\, C\, e\, p_2 \\
\mathcal{S}_p\, C\, e\, (p_1 \texttt{ as } p_2) & = & \mathcal{S}_p\, C\, e\, p_1 \texttt{ as } \mathcal{S}_p\, C\, e\, p_2 \\
\mathcal{S}_p\, C\, e\, (p \texttt{ where } x_1 \texttt{ = } s_1 \texttt{ and } \ldots \texttt{ and } x_n \texttt{ = } s_n) & = & \bigvee \{(\mathcal{S}_p\, C\, (e \setminus_{\{x_1,\ldots,x_n\}} + e')\, p) \texttt{ where } w' \mid (e', w') \in \bigoplus_i \mathcal{S}_w\, s_i\, e(x_i)\}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}_w & : & s \to p \to \mathcal{P}(\mathit{Env} \times \mathit{Bindings}) \\
\mathcal{S}_w\, x\, p & = & \{([x \mapsto p], \{\})\} \\
\mathcal{S}_w\, s\, \_ & = & \{([x \mapsto \_ \mid x \in \mathrm{free}(s)], \{\})\} \\
\mathcal{S}_w\, s\, y & = & \{([x \mapsto x \mid x \in \mathrm{free}(s)], \{y \texttt{ = } s\})\} \\
\mathcal{S}_w\, s\, \texttt{fail} & = & \{\} \\
\mathcal{S}_w\, c(s_1, \ldots, s_n)\, c(p_1, \ldots, p_n) & = & \bigoplus_i \mathcal{S}_w\, s_i\, p_i \\
\mathcal{S}_w\, c(s_1, \ldots, s_n)\, c'(p_1, \ldots, p_{n'}) & = & \{\} \quad (c \neq c') \\
\mathcal{S}_w\, s\, f(p_1, \ldots, p_n) & = & \mathcal{S}_w\, s\, (\mathcal{S}_p[]\, [x_i \mapsto p_i]\, p) \\
 & & \text{where } f \text{ declared as } \texttt{pat } f(x_1, \ldots, x_n) \texttt{ = } p \\
\mathcal{S}_w\, s\, (p_1 \mid p_2) & = & \mathcal{S}_w\, s\, p_1 \cup \mathcal{S}_w\, s\, p_2 \\
\mathcal{S}_w\, s\, (p_1 \texttt{ as } p_2) & = & \mathcal{S}_w\, s\, p_1 \oplus \mathcal{S}_w\, s\, p_2 \\
\mathcal{S}_w\, s\, (p \texttt{ where } x_1 \texttt{ = } s_1 \texttt{ and } \ldots \texttt{ and } x_n \texttt{ = } s_n) & = & \{(e, w \setminus_U \cup \{x_i \texttt{ = } \mathcal{S}_s\, w\, s_i \mid 0 < i \leq n\}) \mid (e, w) \in \mathcal{S}_w\, s\, p\} \\
 & & \text{where } U = \bigcup_i \mathrm{free}(s_i)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}_s & : & \mathit{Bindings} \to s \to s \\
\mathcal{S}_s\, \{\ldots, x \texttt{ = } s, \ldots\}\, x & = & s \\
\mathcal{S}_s\, \{x_i \texttt{ = } s_i\}\, x & = & x \quad \forall_i x_i \neq x \\
\mathcal{S}_s\, w\, c(s_1, \ldots, s_n) & = & c(\mathcal{S}_s\, w\, s_1, \ldots, \mathcal{S}_s\, w\, s_n)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{S}_f & : & \mathit{Cache} \to p \to p \\
\mathcal{S}_f\, C\, f(\vec{x}) & = & f(\vec{x}) \quad x_i \text{ are pattern variables} \\
\mathcal{S}_f\, C\, f(\vec{p}) & = & C(f, \vec{p})\, (\mathrm{free}(\vec{p})) \quad (\text{if } C(f, \vec{p}) \text{ exists}) \\
\mathcal{S}_f\, C\, f(\vec{p}) & = & g\, (\mathrm{free}(\vec{p})), \quad (g \text{ fresh}) \\
 & & \text{add declaration } \texttt{pat } g(\vec{y}) \texttt{ = } \mathcal{S}_p\, C'\, [x_i \mapsto p_i]\, p \\
 & & \text{where} \\
 & & \quad C' = C[(f, \vec{p}) \mapsto g] \\
 & & \quad \vec{y} = \mathrm{free}(\vec{p}) \\
 & & \quad f \text{ declared as } \texttt{pat } f(\vec{x}) \texttt{ = } p
\end{array}
$$

Figure 7: Translation to canonical patterns.