Extensible Pattern Matching Via a Lightweight Language Extension

Don Syme
Microsoft Research,
Cambridge, U.K.
dsyme@microsoft.com

Gregory Neverov
Faculty of Information Technology,
Queensland University of Technology,
Brisbane, Australia
gregory.neverov@gmail.com

James Margetson
Microsoft Research,
Cambridge, U.K.
jamarg@microsoft.com

Abstract

Pattern matching of algebraic data types (ADTs) is a standard feature in typed functional programming languages, but it is well known that it interacts poorly with abstraction. While several partial solutions to this problem have been proposed, few have been implemented or used. This paper describes an extension to the .NET language F# called *active patterns*, which supports pattern matching over abstract representations of generic heterogeneous data such as XML and term structures, including where these are represented via object models in other .NET languages. Our design is the first to incorporate both ad hoc pattern matching functions for partial decompositions and "views" for total decompositions, and yet remains a simple and lightweight extension. We give a description of the language extension along with numerous motivating examples. Finally we describe how this feature would interact with other reasonable and related language extensions: existential types quantified at data discrimination tags, GADTs, and monadic generalizations of pattern matching.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Design

Keywords F#, Functional programming, ML, Pattern matching

1. Introduction

Pattern matching in statically-typed functional languages (STFLs) is a powerful feature that facilitates the concise analysis of data via a switch-and-bind control construct. However a well-recognized problem with pattern matching is its inability to operate on abstract data types. This problem prevents pattern matching from being used in scenarios where its effectiveness is highly sought after. For example many strict STFLs include a lazy list data structure but choose to hide the implementation of the data type by exporting it as an abstract type. This precludes library users from pattern matching over the data type, which would be an intuitive thing to do considering the data is a list. For example, consider a module that exports functions to construct and analyse lazy lists:

```
type LazyList<'a>
val nonempty : LazyList<'a> -> bool
val hd       : LazyList<'a> -> 'a
val tl       : LazyList<'a> -> LazyList<'a>
val consl    : 'a -> Lazy<LazyList<'a>> -> LazyList<'a>
val nil      : LazyList<'a>
```

Tasks that were once very simple to code using pattern matching become obtuse using these analysis functions, e.g., consider code that sums elements of a list of integers pairwise using pattern matching[1]:

```
let rec pairSum xs =
  match xs with
  | Cons(x, Cons(y,ys)) ->
      consl (x+y) (lazy (pairSum ys))
  | Cons(x, Nil()) ->
      consl x (lazy nil)
  | Nil() ->
      nil
```

becomes

```
let rec pairSum xs =
  if nonempty xs then
    let x, ys = hd xs, tl xs
      if nonempty ys then
        let y, zs = hd ys, tl ys
          consl (x+y) (lazy (pairSum zs))
      else consl x (lazy nil) )
  else nil
```

Even if `LazyList` were not an abstract type, pattern matching would still be problematic because of the need to force evaluation of the list in the middle of matching. Note that it is nested pattern matches that causes particular problems in this regard.

While this problem has long been recognized (Wadler 1987; Okasaki 1998), it becomes more severe when STFLs are placed in the context of modern object-oriented programming frameworks (e.g., .NET and Java), which is the idea behind languages such as F# (Syme and Margetson 2006), Nemerle (Nemerle 2006) and Scala (Odersky 2006). Object-oriented code heavily employs abstract types to realize encapsulation—a fundamental design principle of object-oriented programming—hence programmers of the languages cited above frequently encounter abstract types but cannot deal with them in a natural manner because of the limitations of pattern matching.

This problem also manifests itself to authors of software libraries. Revealing algebraic data types in a library design fixes the usage model of the type to such a degree that their use in the public

ICFP'07 October 1–3, 2007, Freiburg, Germany.

[1] Throughout this paper we use F# indentation-aware syntax that allows the OCaml `in` keyword to be omitted.

APIs of framework components is hard to encourage. Indeed, it becomes evident that apart from simple cases such as lists, pairs and options, most algebraic data types are *implementations* of types, rather than descriptions of long-term maintainable abstractions. It is also evident that this is one of the reasons why pattern matching and algebraic data types have not been successfully transferred to object-oriented languages such as Java and C# despite proposals in that direction (Odersky and Wadler 1997).

To summarize, pattern matching on concrete types is problematic because—

- it is not extensible;
- it encourages programmers to break abstraction boundaries;
- it leads to libraries that are difficult to maintain and evolve;
- it leads to a discontinuity in programming: programmers initially use pattern matching heavily, and are then forced to abandon the technique in order to regain control over data representations.

The authors have witnessed all of these problems in practice in compiler, theorem prover and library implementations.

This paper considers the problem as it applies to the F# language. F# is a pragmatically-oriented dialect of ML based on the core design of OCaml. It interoperates with other .NET languages and bridges the gap between the functional and object-oriented worlds by providing both ML-style functional programming and type-inferred object-oriented programming. By a minimalistic language extension we are able to make pattern matching a powerful and flexible feature, whether it be used against internal or public, or functional or object-oriented types. While we have worked by modifying F#, our results are applicable to any statically typed functional language. Furthermore we believe our design could be used as a basis for introducing a pattern matching facility in imperative object-oriented languages such as C# and Java.

In this paper we do the following:

- We introduce the concept of an *active pattern* in F#. Active patterns can be used in any pattern expression, can be defined to operate on any type, and can be checked statically for completeness and redundancy of a match. We also give the evaluation semantics of active patterns by way of a pattern interpreter.
- We present numerous examples of active patterns in action which demonstrate their success at functionally decomposing abstract data types.
- We describe how active patterns are implemented in F#.[2]
- We consider how this feature would interact with three other reasonable and related language extensions: existential types quantified at data discrimination tags, GADTs, and monadic generalizations of pattern matching.

The primary specific contribution of this paper is that it presents the first design for extensible pattern matching to incorporate both *partial* and *total* decompositions within the context of a regular, simple and lightweight extension to a language. In addition, this work is different to many previous attempts at extensible pattern matching in functional languages because it:

- considers the feature interactions mentioned above, potentially helping to smooth the path for the adoption of the design in other languages;
- addresses the need for functional languages to interoperate with object-oriented ones, and
- is implemented and available for use in a mature programming language system.

The rest of this paper is structured as follows: in §2 we describe the active pattern mechanism in F#, mostly by example, and in §3 we describe the operational semantics of pattern matching. In §4 we look at further examples, and in §5 we discuss implementation issues. In §6 we look at interactions with other language features, and §7 summarizes and discusses related work.

2. Active Patterns in F#

An active pattern is a pattern defined without reference to a discriminated union type declaration. At a basic level an active pattern is just a regular function, but it is defined using a new syntactic element called a *structured name* which gives it special significance in the language.

2.1 Simple Total Patterns ("Basic Bananas")

Patterns are used to decompose data into a number of sub-cases. To begin our exploration of active patterns we will consider the simplest pattern imaginable: one that decomposes data into only one sub-case.

Suppose we have a type for complex numbers and wish to write an addition function. The `complex` type is not an exported union type but nevertheless we wish to write our `add` function by pattern matching. We can do this using active patterns as follows:[3]

```
open Math.Complex
let (|Rect|) (x:complex) = (x.RealPart, x.ImaginaryPart)

let add a b =
  match a, b with
  | Rect(ar,ai), Rect(br,bi) -> mkRect(ar+br, ai+bi)
```

The first line defines an active pattern called `Rect`. The term `(|Rect|)` is a structured name, in this case a regular name enclosed in "banana" brackets. Structured names may appear anywhere where a regular name is used in a binding position. When a structured name is bound by `let` it has two effects.

1. The same effect as if it were a regular name: the structured name will be bound to an expression and added to the term environment. In this example the name `(|Rect|)` will be bound to a value of type `complex -> float * float`.
2. The regular name will also be added to the environment of patterns, enabling it to be used in patterns where it is in scope. Previously the only way of adding a new pattern was by defining a new union type.

The type of the pattern is implied from the type of the function. In this case the `Rect` pattern matches `complex` and yields a residual of `float * float`. If a structured name is not bound to a function value then it is not a valid pattern and a type error will result when its tags are used.

In the `add` function, `Rect` is executed as part of the pattern matching process. The `add` function is semantically equivalent to this code without active patterns:

```
let add a b =
  let ar,ai = Rect a
  let br,bi = Rect b
  mkRect(ar+br, ai+bi)
```

Additionally we could define another active pattern that transforms the `complex` type into polar form.

[2] Active patterns using the design described here have been a feature of F# since version 1.9.1. All the examples presented in this paper run on the current release, which is available for download at `research.microsoft.com/fsharp`.

[3] The F# library module `Complex` contains functions `mkRect` and `mkPolar`.

```
let (|Polar|) (x:complex) = (x.Magnitude, x.Phase)

let mul a b =
  match a, b with
  | Polar(m,p), Polar(n,q) -> mkPolar(m+n, p+q)
```

Note the concrete representation of `complex` has *not* syntactically dictated the representation used to consume the type in this program, as it would if we used pattern matching over the concrete structure. The concrete representation of `complex` could change between rectangular and polar form, or between tuple and record types without impacting a consumer's pattern matching code.

The above functions could also have been written

```
let add (Rect(ar,ai)) (Rect(br,bi)) = mkRect(ar+br,ai+bi)
let mul (Polar(m,p)) (Polar(n,q)) = mkPolar(m+n,p+q)
```

with the same result.

2.2 Multiple Case Total Patterns ("Banana Splits")

Decomposing data into one sub-case is not a very general task. We need active patterns to decompose data into one of many sub-cases. We allow this by expanding the format of a structured name. The bananas of a structured name can now enclose multiple names separated by splits, `(|)`.

Suppose we wish to create an active pattern interface to the standard F# lazy list type, which is abstract. We can write this in F# as follows:

```
let (|Cons|Nil|) l =
  if nonempty l then Cons(hd l,tl l)
  else Nil
```

Here the structured name `(|Cons|Nil|)` defines two regular names `Cons` and `Nil`. These regular names are used on the right of the let binding to tag different cases of the pattern. The right-side of the let binding is given an anonymous sum type. Like tuples (i.e., anonymous product types), the F# language also predefines a family of anonymous sum types as follows:[4]

```
type ('a,'b) choice =
  | Choice2_1 of 'a
  | Choice2_2 of 'b

type ('a,'b,'c) choice =
  | Choice3_1 of 'a
  | Choice3_2 of 'b
  | Choice3_3 of 'c

(* etc. *)
```

These types are primarily used to encode the result of multiple case patterns. Hence in this example the active pattern `(|Cons|Nil|)` will have type `'a llist -> ('a * 'a llist, unit) choice`. The compiler translates tags from the structured name to tags of the `choice` type in order. However the programmer never needs to use the `choice` type directly with active patterns except in signatures. The names `Cons` and `Nil` are put in scope as patterns that match `'a llist` and yield `'a * 'a llist` and `unit` residuals respectively. These names are used in patterns to match lazy lists without knowledge of the underlying `choice` type.

The example used in the introduction can now be written in a much more natural way:

```
let rec pairSum xs =
    match xs with
    | Cons(x, Cons(y,ys)) ->
        consl (x+y) (lazy (pairSum ys))
    | Cons(x, Nil()) ->
        consl x (lazy nil)
    | Nil() ->
        nil
```

The pattern names defined by an active pattern have identity. This allows the compiler to perform completeness and redundancy analysis of match blocks, and generate efficient code that does not recompute patterns that have already been applied. In compiled code active patterns are matched over their `choice` type representation. The `pairSum` example will compile to code like the following which does not recompute the active patterns if the first rule fails.

```
let rec pairSum xs =
    match (|Cons|Nil|) xs with
    | Choice2_1 (x, ys') ->
        match (|Cons|Nil|) ys' with
        | Choice2_1 (y,ys)) ->
            consl (x+y) (lazy (pairSum ys))
        | Choice2_2 ()      ->
            consl x (lazy nil)
    | Choice2_2 ()       ->
        nil
```

Already active patterns are powerful enough to provide a robust pattern interface to an existing object-oriented data type—this is important for F# because programmers constantly deal with code in the .NET base class library and from other languages. For example, the .NET base class library provides a `Type` type that represents reified run-time types and is used throughout the frameworks reflection and code generation APIs. The `Type` class is defined as follows:

```
type Type with
  member IsGenericType : bool
  member GetGenericTypeDefinition : unit -> Type
  member GetGenericArguments : unit -> Type[]
  member HasElementType : bool
  member GetElementType : unit -> Type
  member IsByRef : bool // an managed pointer
  member IsPointer : bool // an unmanaged pointer
  member IsGenericParameter : bool
  member GenericParameterPosition : int
```

In essence, this class interface is trying to communicate that a `Type` object can be exactly one of the following:

1. A named typed with a name (represented by another `Type` object) and list of type parameters.
2. A array type with a rank and element type.
3. A pointer type that could be managed or unmanaged.
4. A type parameter[5]

However the API is subtle: for example `GetGenericTypeDefinition` fails if `IsGenericType` returns `false`, when you might expect it to be the identity function in this case. This is a consistent cause of irritating bugs when using this API.

We can define an active pattern that hides this complexity and captures the essential algebraic structure of `Type` objects:

```
let (|Named|Array|Ptr|Param|) (typ : System.Type) =
    if typ.IsGenericType
    then Named(typ.GetGenericTypeDefinition(),
               typ.GetGenericArguments())
    elif typ.IsGenericParameter
    then Param(typ.GenericParameterPosition)
    elif not typ.HasElementType
    then Named(typ, [| |])
```

[4] .NET languages permit overloading by arity of generic type constructor. At the time of writing the F# release only includes choice types up to $n = 7$. It is trivial to have the compiler encode `choice` types of greater n into these.

[5] There are no binding constructs; type parameters are bound at method and class definitions.

```
    elif typ.IsArray
    then Array(typ.GetElementType(),
               typ.GetArrayRank())
    elif typ.IsByRef
    then Ptr(true,typ.GetElementType())
    elif typ.IsPointer
    then Ptr(false,typ.GetElementType())
    else failwith "MSDN says this can't happen"
```

We can now write code that consumes this type in a functional manner to, say, pretty print a `Type` object:

```
let rec formatType typ =
    match typ with
    | Named (con, [])   ->
        sprintf "%s" con.Name
    | Named (con, args) ->
        sprintf "%s<%s>" con.Name (formatTypes args)
    | Array (arg, rank) ->
        sprintf "Array(%d,%s)" rank (formatType arg)
    | Ptr(true,arg)     ->
        sprintf "%s&"  (formatType arg)
    | Ptr(false,arg)    ->
        sprintf "%s*"  (formatType arg)
    | Param(pos)        ->
        sprintf "!%d" pos

and formatTypes typs =
    String.Join(",",Array.map formatType typs)
```

or to collect the free generic type variables:

```
let rec freeVarsAcc typ acc =
    match typ with
    | Array (arg, rank) -> freeVarsAcc arg acc
    | Ptr (_,arg)       -> freeVarsAcc arg acc
    | Param _           -> (typ :: acc)
    | Named (con, args) ->
        Array.fold_right freeVarsAcc args acc

let freeVars typ = freeVarsAcc typ []
```

The pattern effectively allows us to view `Type` objects as if they had been defined using the following union type:

```
type Type =
    | Named of Type * Type[]
    | Array of int * Type
    | Ptr of bool * Type
    | Param of int
```

2.3 Partial Patterns ("Banana Slices")

Our examples so far have been of active patterns that decompose types into a complete set of sub-cases. However this is not the only useful way to decompose data. Heterogeneous data types such as term structures, XML and strings can be analysed in many different ways, most of which are incomplete and application-dependent. For example, say we have heterogeneous data stored as a string and we want to pattern match over the string to extract structured data. We would like to write code as follows.

```
match str with
| ParseInt i -> IntVal i
| ParseFloat f -> FloatVal f
| ParseDate d -> DateVal d
| ParseColour c -> ColourVal c
| _ -> failwith "unrecognized data"
```

These four "parse" active patterns do not form a complete decomposition of strings. Moreover they overlap because `ParseInt` and `ParseFloat` would both match the string `"0"`. Furthermore these patterns need not be defined together, indeed *should not* be defined together so that new parse patterns can be added in the future. For these reasons these active patterns are *partial* and can be defined like so.

```
let (|ParseInt|_|) s =
  let i = ref 0
  if Int32.TryParse(s, i) then Some !i
  else None

val colors : (string * colour) list

let (|ParseColour|_|) s = try_assoc s colours
```

A partial active pattern is defined using a structured name with a trailing underscore to indicate the incompleteness of a match. A partial pattern either succeeds and yields residual data or it fails. A failure indicates that other patterns in the match block should be tried.

Structured names with an underscore are given an `option` type. Hence, `(|ParseInt|_|)` has type `string -> int option` and `ParseInt` may be used as a pattern that matches `string` and yields `int`.

For completeness our specification includes structured names with multiple cases, e.g, `(|ParseInt|ParseFloat|_|)`. However we have yet to detect any practical benefit in doing so. One advantage of multiple case patterns in the previous section was that it enabled the compiler to perform completeness analysis on a match block. However this is lost here because of the inherent incompleteness of the pattern. The other advantage of multiple case patterns is that it can give more efficient match evaluation. However this is only effective if different cases share a significant part of the active pattern implementation. In practice we have found this seldom occurs—it is too easy to reuse separate existing parse `int` and `float` functions than to write your own code that simultaneously parses an `int` and a `float`. For these reasons, partial active patterns with multiple cases are not implemented in the current release of F#.

Partial patterns have different evaluation semantics to total patterns. Consider the function—

```
let f s =
  match s with
  | ParseInt 0 -> Zero
  | ParseFloat f -> NonZero f
  | _ -> failwith "not a number"
```

If `ParseInt` and `ParseFloat` were part of the same total active pattern then the expression `f "1"` would evaluate to the `failwith` clause. This is because `"1"` would be successfully parsed as an integer but fail matching the integer `1` against `0`. Then the second clause would be skipped because, by virtue of being in a total pattern, `ParseInt` and `ParseFloat` are known to be mutually exclusive. Hence if one succeeds the other must automatically fail, and so the third clause is hit. However this does not happen when the patterns are defined as partial. The success (or failure) of a partial pattern gives no information about how other partial patterns may succeed or fail.

2.4 Parameterized Patterns ("Scrap Your Banana Plate")

When using active patterns—particularly partial ones—it quickly becomes necessary to parameterize them to express queries such as "Match an attribute A on an XML Node" or "Match any term in an abstract syntax tree involving a call to function M". Say we wanted a pattern that matches strings against a regular expression. To do this the active pattern must be parameterized on the regular expression. We can do this like so[6]:

```
let (|ParseRegex|_|) re s =
```

[6] This code uses the standard regular expression library of the .NET framework.

```
  let m = Regex(re).Match(s)
  if m.Success
  then Some [ for x in m.Groups -> x.Value ]
  else None
```

This active pattern has the expected type `string -> string -> (string list) option`—it returns a list of matched groups from the regular expression. The type of `ParseRegex` as a pattern is more complicated. At one level it could be viewed as single-case total pattern that matches `string` and yields `string -> (string list) option`, but that is not particularly useful. So instead initial arguments of an active pattern can be applied at its usage site. So we could write a function that swaps the parts of a hyphenated word like so:

```
let swap s =
  match s with
  | ParseRegex "(\w+)-(\w+)" [l;r] -> r ^ "-" ^ l
  | _ -> s
```

Parameterizing an active pattern results in the loss of its identity, hence the compiler cannot perform redundancy or completeness analysis and a parameterized pattern will be re-evaluated every time it appears in a match block, even if it has syntactically the same arguments, an issue we return to in §5.1.2.

2.5 First-Class Pattern Values ("First-Class Bananas")

Since active patterns are simply functions they are first-class values in the language and hence can be lambda abstracted. This is useful for writing higher-order active patterns—i.e., a pattern parameterized on other patterns. For example, consider an unfold combinator that applies a partial function, `q`, zero or more times (here `q` has type `'t -> ('a * 't) option` and the input `inp` has type `'t`):

```
let qZeroOrMore q inp =
  let rec queryAcc rvs e =
     match q e with
     | Some(v,body) -> queryAcc (v::rvs) body
     | None -> (List.rev rvs,e) in
  queryAcc [] inp
```

Consider a partial active pattern to match Lambda nodes in an expression tree:

```
type expr =
  | Lam of string * expr
  | App of expr * expr
  | Var of string

let (|Lambda|_|) = function Lam(a,b) -> Some(a,b)
                          | _        -> None
```

A total pattern can now be defined using this as a first-class value:

```
let (|Lambdas|) e = qZeroOrMore (|Lambda|_|) e
```

Furthermore, `qZeroOrMore` could even have been written using a variable with a structured name as a parameter:

```
let qZeroOrMore (|Q|_|) inp =
    let rec queryAcc rvs e =
        match e with
        | Q(v,body) -> queryAcc (v::rvs) body
        | _ -> (List.rev rvs,e) in
    queryAcc [] inp
```

This shows that active patterns really are just values with structured names.

2.6 "Both" Patterns ("Have Your Banana and Eat It Too")

Many STFLs such as F#, OCaml and SML '97 include "either" patterns *pat* `|` *pat*, which succeed if either the left or right patterns match (the patterns must bind identical variables at identical types). As has been noted by Rossberg (2007a), the natural dual to "either" patterns is "both" patterns *pat* `&` *pat* that only succeed if both the left and right patterns match. "Both" patterns are not particularly useful in traditional STFLs since most uses can be combined into a single pattern. However, that changes when the set of matching constructs is extensible. For this reason we extend F# matching with "both" patterns. We will see realistic examples of these in §4.2.

Structured Name	Kind	Expected Return Type
$(\|A\|)$	Single-case total	a
$(\|A_1\|\ldots\|A_n\|)$	Multi-case total	$(a_1,\ldots,a_n)$ `choice`
$(\|A\|\_\|)$	Single-case partial	a `option`
$(\|A_1\|\ldots\|A_n\|\_\|)$	Multi-case partial	$(a_1,\ldots,a_n)$ `choice option`

Table 1. Kinds of active recognizers and their structured names

2.7 Summary

In this section we have presented the basic design of active patterns in F#. Active patterns are predicated on structured names. Structured names introduce new pattern names into the environment of patterns hence making pattern matching extensible. The abstract syntax of a structured name is $(|id|\ldots|id|\{\_|\}?)$. Table 1 shows the different classes of structured names.

Active patterns are simply functions and as such may take parameters and be parameters themselves. An active pattern function has type

$$\tau_1 \rightarrow \ldots \rightarrow \tau_N \rightarrow \tau_{inp} \rightarrow \tau$$

for some $N \geq 0$, where the N initial arguments are the *parameters* to the pattern and the last argument is the *input* to be matched. The return type must conform to the shape indicated in Table 1 based on the form of the structured name, if not a type error will result on use. The input argument of type τ_{inp} can be any type including abstract types, primitive types, union types and object types. Moreover a type can have any number of active patterns defined over it.

3. Operational Semantics

In this section we give a model operational semantics for pattern match evaluation. We do this in two steps:

1. We give a naive semantics via an interpreter that evaluates patterns rule-by-rule;
2. We informally outline the changes required to give an Okasaki-style semantics (Okasaki 1998) that ensures that invocations of active patterns are cached, i.e., only executed once for a given input.

3.1 A Naive Dynamic Semantics

Since a naive semantics is not difficult, we avoid the traditional approach of using inference rules. Instead we present a simple interpreter for pattern matching, originally as an OCaml/F# program and here presented in programmatic notation using only well-founded recursion, pure lambda calculus constructs and simple data types.[7] We only give the relevant pattern matching portion of the dynamic semantics.

[7] An inference rule presentation is easy to derive from the one we give, should it be deemed necessary. However inference rules are harder to type check, debug and maintain than a simple interpreter.

```
type env
type expr
type exprs = expr list
type state
type tag = string

type pat =
    | PPair of pat * pat                Tuple patterns
    | PTag of tag * pat                 A data pattern
    | PActive of tag * exprs * pat      An active pattern
    | PEither of pat * pat              'or' patterns
    | PBoth of pat * pat                'and' patterns
    | PWild                             _ patterns
    | PId of string                     Variable patterns
    | PConst of int                     Constant patterns

type value =
    | VPair of value * value            Pair values
    | VTag of string * value            Tagged values
    | VConst of int                     Constants

type rule = pat × expr
type rules = rule list
```

Figure 1. Input terms and values for the operational semantics

The input syntax terms are shown in Figure 1. As shown in Figure 2 we assume the existence of a type of environments, a type of expressions, a function *applyExpr* to evaluate/apply expressions, and a function *resolveActiveTag* that resolves an active pattern label to an expression and further information indicating the kind of the pattern (partial or total), the number of tags in the tag set of the pattern and the position of the tag in the tag set.

In Figure 3 we give the definitions of functions *matchPat* and *matchRules* that match a single pattern and a set of rules respectively. We pass an explicit state since evaluating F# expressions may change a global state. The interesting points of the semantics are:

- Active patterns are first resolved to an expression, the expression is applied (perhaps to some additional active parameter arguments), and a further pattern match executed for a pattern built using `Some`, `None`, `Choice1_1`, `Choice2_1`, etc. pattern constructions. That is, the active pattern must resolve to a function expression which returns appropriate `Choice`-tagged data.
- The environment is only extended after pattern matching: identifiers bound by the pattern may not be used in the pattern. This is different to some other proposals for extensible pattern matching (Rossberg 2007b). We think this helps make patterns more readable and understandable.

3.2 Applying the Okasaki Condition

Okasaki has argued convincingly that the only sensible semantics to apply to pattern match execution in a language with side effects is to require that active discrimination functions be run at most once against any given input within the context of a given collection of pattern rules (Okasaki 1998). It is easy to extend our semantics to cover this case.[8]

- Within a single invocation of *matchRules* the state s is augmented with a lookup table keyed by *paths*. Paths are lists of

[8] At the time of writing the F# compiler does not implement the Okasaki semantics, but does run the pattern compilation algorithm we describe in §5. This means it may run active patterns more than once against the same input. It thus effectively assumes that active patterns do not have side effects, or if they do then they are benign.

```
applyExpr :
    env × state × expr × exprs × value → state × value

resolveActiveTag: env × string → expr × bool × int × int

type bind = string * value
type binds = bind list
```

Operator to build a `Choice` tag for the use of an active pattern identifier:
tagName `m n = sprintf "Choice%d_%d" m n`

Operators to combine pattern evaluations conjunctively and disjunctively:

```
f1 ⋀ f2 =
    λ(s,binds).
       s',matchRes = f1 (s,binds)
       match matchRes with
        | None → s',None
        | Some(binds) → f2 (s',binds)

f1 ⋁ f2 =
    λ(s,binds).
        s',matchRes = f1 (s,binds)
        match matchRes with
        | None → f2 (s',binds)
        | Some(binds) → (s',Some(binds))
```

Figure 2. Assumptions and preliminary definitions

```
matchPat : env × pat × value
           → state × binds → state × binds option
matchPat (env,pat,v) (s,binds) =
  match pat,v with
  | PPair(p1,p2), VPair(v1,v2) →
      (matchPat(env,p1,v1) ⋀ matchPat(env,p2,v2)) (s,binds)
  | PBoth(p1,p2)   ,_ →
      (matchPat(env,p1,v) ⋀ matchPat(env,p2,v)) (s,binds)
  | PEither(p1,p2),_ →
      (matchPat(env,p1,v) ⋁ matchPat(env,p2,v)) (s,binds)
  | PTag(s1,p'), VTag(s2,v') when s1 = s2 →
      matchPat (env,p',v') (s,binds)
  | PConst(c1), VConst(c2) when c1 = c2 → (s,Some binds)
  | PWild,_ →  (s,Some binds)
  | PId(nm),v → (s,Some ((nm,v)::binds))
  | PActive(nm,args,p0), _ →
      f,total,numCh,n =  resolveActiveTag(env,nm)
      s',v' = applyExpr(env,s,f,args,v)
      p1  =
        if numCh = 1 then p0
        else PTag(tagName numCh n,p0)
      p2 = if total then p1 else PTag("Some",p1)
      matchPat(env,p2,v') (s',binds)

  | _ → (s,None)

matchRules: env × rules × value
            → state → state × binds option
matchRules(env,rules,v) s =
    match rules with
    | [] → (s,None)
    | (pat,expr) :: rules' →
        s',matchRes = matchPat (env,pat,v) (s,[])
        match matchRes with
        | None -> matchRules (env,rules',v) s'
        | Some(binds) → (s',Some(binds,expr))
```

Figure 3. Pattern Matching: Naive Operational Semantics

identifiers. We assume functions *lookup* and *record* exist to read and write this table.

- At each initial call to *matchPat* the path is empty. In the recursive calls to *matchPat* the path is extended in different cases as follows:
  - for **PPair** it is extended by L/R on the left/right respectively;
  - for **PTag** it is extended by the data tag;
  - for **PActive** patterns with no arguments it is extended by the pattern tag;
  - for **PActive** patterns with arguments it is extended by a freshly generated identifier, for reasons covered in §5.1.2;
  - for other cases the path is not extended.

The lookup table is consulted at **PActive** patterns by replacing the application of *applyExpr* with:

```
| PActive (nm,args,p'), _ →
  s,v' =
      match lookup s path with
      | None →
          s',v' = applyExpr(env,s,f,args,v)
          s'' = record s path v'
          s'',v'
      | Some v' → s,v'
  ...
```

These changes are together sufficient to ensure that we only invoke an unparameterized active pattern on the "same" input at most once while matching a value against a set of rules, where paths are used to determine if two inputs are the same.

3.3 Static semantics

We do not give a corresponding static semantics, as it follows the normal type-checking rules for patterns. However we note the following:

- As expected, an additional case for active patterns is required and it follows the form of the corresponding case for the dynamic semantics with a constraint that the return type of the active pattern has the shape expected as specified in Table 1.
- Different total and partial patterns may be used in the same pattern to match against the same inputs. However, this may impair the compiler's ability to analyse redundancy and incompleteness.
- The static semantics stay simple only if we do not attempt to specify redundancy and incompleteness checking. These do not normally form part of the specification of pattern matching in ML-family languages and are instead seen as compiler-specific features added to enhance programmer productivity. [9]

4. Further Examples of Active Patterns

In this section we look at three additional examples of the use of active patterns.

4.1 Join Lists

Join lists are a classic example of the use of view-like mechanisms in functional languages. They are also an example of recursive pattern definitions. Here is the standard polymorphic join list example in F# code:

[9] Recently Maranget (2007) has proved the correctness of some pattern matching algorithms with respect to these properties, and we believe these techniques may be helpful for active patterns as well.

```
type 'a jlist =
    | Empty
    | Single of 'a
    | Join of 'a jlist * 'a jlist

let rec (|Cons|Nil|) = function
    | Single x                -> Cons(x, Empty)
    | Join(Cons(x,xs), ys)    -> Cons(x, Join(xs, ys))
    | Join(Nil(), Cons(y,ys)) -> Cons(y, Join(ys, Empty))
    | Empty
    | Join(Nil(), Nil()) -> Nil()

let jhead js =
    match js with
    | Cons(x,_) -> x
    | Nil       -> failwith "empty list"

let rec jmap f xs =
    match xs with
    | Cons(y,ys) -> Join(Single(f y), jmap f ys)
    | Nil() -> Empty
```

The definition of the `(|Cons|Nil|)` total pattern is syntactically very close to the corresponding *view* definition as proposed in (Wadler 1987). This is pleasing: the pattern being defined can be used within its own definition, and type inference works effectively for these definitions.

4.2 XML Matching

XML is perhaps the most important structured heterogeneous data type in use today. In this section we present an initial version of defining compositional patterns for XML fragments. We focus on patterns that traverse the immediate structure of XML trees, rather than query operators. The talented programmer is free to define suitable new patterns, perhaps based on advanced query tools that may be implemented by existing XML libraries such as XLinq (Meijer and Beckman 2006).

Our example uses the `System.Xml` API of the .NET libraries. For our purposes this simply makes XML available as an untyped expression tree accessed via the following types and dot-notation members:

```
type XmlNode with
  member Item : string -> XmlNode
  member Name : string
  member Attributes: XmlAttributeCollection
  member ChildNodes: XmlNode[]

type XmlAttributeCollection with
  member GetNamedItem: string -> string

type XmlAttribute with
  member Value: string
```

In this example our aim is to map XML representing a scene graph of 3D shapes into an algebraic datatype, e.g. consider the following input:

```
<Scene>
     <Intersect>
        <Sphere r='2' x='1' y='0' z='0'/>
        <Intersect>
           <Sphere r='2' x='4' y='0' z='0'/>
           <Sphere r='2' x='-3' y='0' z='0'/>
        </Intersect>
        <Sphere r='2' x='-2' y='1' z='0'/>
     </Intersect>
</Scene>"
```

A suitable F# type to represent this data in a strongly-typed fashion is:

```
type scene =
```

```
    | Sphere of float * float * float * float
    | Intersect of scene list
```

We first define some general-purpose and simple active patterns that we can reuse for many XML samples. The partial pattern `(|Elem|_|)` checks an element has a given name:[10]

```
let (|Elem|_|) name (inp: #XmlNode) =
    if inp.Name = name then Some(inp)
    else None
```

We next define patterns `(|Attributes|)` to extract the attributes from a node, and `(|Attr|)` to look for an attribute of a particular name and extracts is value:

```
let (|Attributes|)  (inp: #XmlNode) = inp.Attributes

let (|Attr|_|) attr (inp: XmlAttributeCollection) =
    match inp.GetNamedItem(attr) with
    | null -> None
    | node -> Some(node.Value)
```

Our final general-purpose pattern converts a string to a float:

```
let (|Float|_|) s =
    try Some(Float.of_string s) with _ -> None
```

We can now write a derived pattern to match a collection of attributes that represent a vector, e.g. `x='-3' y='0' z='0'`. Note this pattern cannot fail except by raising an exception:

```
let (|Vector|_|) inp =
   match inp with
   | (Attr "x" (Float x) &
      Attr "y" (Float y) &
      Attr "z" (Float z)) -> Some(x,y,z)
   | _ -> None
```

We can now write recursive functions to map XML nodes named `Sphere` or `Intersect` into the datatype:[11]

```
let rec (|ShapeElem|_|) inp =
    match inp with
    | Elem "Sphere"
        (Attributes (Attr "r" (Float r)  &
                     Vector (x,y,z)))
        -> Some (Sphere (r,x,y,z))
    | Elem "Intersect" (ShapeElems(objs))
        -> Some (Intersect objs)
    | _ -> None

and (|ShapeElems|) inp =
    [ for (ShapeElem y) in inp.ChildNodes -> y ]
```

Finally we can wrap this up in a parse function that checks the top node is a `Scene` node and extracts the shapes from its child nodes:

```
let parse inp =
    match inp with
    | Elem "Scene" (ShapeElems elems) -> elems
    | _ -> failwith "not a scene graph"

let inp = "... the XML above ..."
let doc = new XmlDocument()
let res = doc.LoadXml(inp)
```

We have now successfully mapped an untyped XML document into the following strongly typed data:

```
res : scene
    = Intersect
        [ Sphere((2.0,1.0,0.0,0.0);
          Sphere(2.0,4.0,0.0,0.0);
          Sphere(2.0,-3.0,0.0,0.0) ];
        Sphere(2.0,-2.0,1.0.0.0) ]
```

4.3 Quotations

F# allows a form of meta-programming where F# code can be reified as values at run-time and manipulated (Syme 2006b). Quasi-quotation provides a convenient means of constructing code values; however there is no convenient solution for deconstructing code values. Traditional pattern matching cannot be used because code is represented by an abstract type. Even if it could, it is useful to have multiple different decompositions to view code at the right level of abstraction for the analysis being performed, e.g., in terms of low-level lambda abstractions or in terms of high-level control structures.

Matching on quotations was a major consideration for the design of active patterns, initially sparked by quotation matching in ForteFL (Grundy et al. 2006), and code patterns in MetaML (Taha and Sheard 1997). For example, quotation literals, written `<@@ ... @@>`, can be passed as parameters to active patterns which use the literals to help drive the matching process:

```
open Quotations
open Quotations.Raw

// interp : Quotations.Raw.Expr -> float
let rec interp inp =
  match inp with
  | TopDefnApp <@@ sin @@> [x] -> sin (interp x)
  | TopDefnApp <@@ cos @@> [x] -> cos (interp x)
  | Double(x) -> x
  | _ -> failwith "unrecognized"

printf "res1 = %g" (interp <@@ sin(cos(1.0)) @@>)
```

In this example, the active pattern `TopDefnApp` from the F# library matches quotation terms that represent applications of a specific function indicated by its parameter, in this case the F# functions `sin` and `cos`.

5. Implementation

In this section we look at two aspects related to the implementation of the mechanism described in this paper: pattern match compilation and the representation of return results.

5.1 Pattern Match Compilation

For pattern match compilation F# uses the generalized pattern compilation algorithm of Scott and Ramsey (2000) with a left-to-right heuristic. Modifying this algorithm to implement a valid interpretation of active patterns was fairly straight-forward.

The algorithm of Scott and Ramsey (2000) works as follows. At each step, a heuristic chooses a *point of investigation* for a collection of *frontiers*. Frontiers represent partially investigated pattern match rules. A point of investigation corresponds to a single decision point (e.g. a switch on an integer tag). Each point of investigation is represented by a sequence of integers called a path, and, in the absence of active patterns, represents a path to a subterm of the input term. Given the point of investigation, the frontiers are divided into those edges that are *relevant*, i.e. where information from the investigation may result in the success/failure of the rule, and those that are *tips*, i.e. irrelevant. A decision tree node is then constructed that has subtrees corresponding to *projecting* the success/failure of the investigation through the relevant edges. A default case is added for the tips. The process is then repeated until all frontiers are exhausted. Match incompleteness warnings can be given if a final "dummy" rule is ever exercised.

[10] As in OCaml the notation `#`ty means "a type variable constrained to be any subtype of ty".

[11] The definition of `ShapeElems` uses F# list comprehension notation.

5.1.1 Modification 1: Choosing the Edge Set

In the absence of active patterns, the algorithm of Scott and Ramsey (2000) ensures that all irrelevant frontiers have a trivial (i.e. wildcard or variable) patterns at the point of investigation. With active patterns this assumption is no longer valid, because it might take several different investigations to run several different active patterns against a given input. We thus modified the algorithm as follows:

- When partitioning edges, choose a *prefix* of relevant edges based on the point of investigation, where all the edges are related to the same pattern. If the pattern has no *identity*, i.e., is a parameterized active pattern, then only the first relevant edge is chosen.

5.1.2 Modification 2: Pattern Identity and Path Identifiers

A second modification to the algorithm of Scott and Ramsey (2000) is necessary to ensure a distinction between "sub-terms" and "paths". Paths describe potential points of investigation in the pattern structure. In the presence of active patterns, paths must record which active patterns have led us to a particular nested pattern. Consider the following:

```
let (|Bit|) n =
    let mask = 1ul <<< n in
    fun inp -> ((inp &&& mask) <> 0ul)

match 0b0001000100ul with
| Bit 3 true  -> printfn "No!"
| Bit 2 false -> printfn "No No!"
| Bit 2 true & Bit 3 false  -> printfn "Yes indeed!"
| _ -> failwith ""
```

If the `Bit 3` pattern succeeds, but its `true` sub-pattern fails, then no information is gained about the success or failure of the `false` sub-pattern of `Bit 2 false`. This is because the parameter to the pattern is different in each case, or, more specifically, because we don't consider parameterized patterns to have any kind of identity. In a naive extension of the original algorithm these would be given identical path locations, which would be incorrect.

For this reason, we extended the notion of path so that different instances of parameterized patterns encountered through pattern match compilation are allocated fresh, unique integers and these integers are used within paths.

5.1.3 Modification 3: Rule Chunking

The extensive use of active patterns (particularly partial patterns) can quickly lead to significant (even exponential) blow up in the size of generated decision trees (Okasaki 1998). This is partly due to the fact that failing sub-patterns can lead to duplications of the large frontier sets that are used to investigate multiple rules simultaneously.

For this reason, we additionally modified the algorithm of Scott and Ramsey (2000) to abandon the use of large frontier sets whenever partial patterns are used. That is, when compiling N rules, we have a choice as to whether we compile all rules simultaneously, or one-by-one, or in chunks. We choose a prefix of rules up to the first that uses any kind of partial pattern. This may result in active patterns being called more times than may be expected, but reduces code size substantially on some real-world examples.

5.2 Performance and the Representation of Return Results

Performance is not the primary focus of this paper, for the following reasons:

- We believe that even a naive implementation of the constructs described here would increase expressive power sufficiently to justify their inclusion in a language.
- The inclusion of multi-way *total* patterns in the design inherently gives us a foundation for significantly better performance than the proposed extensions for ad hoc matching. Multi-way total patterns allow multiple rules to be explored with a single discrimination, as with regular matching on discriminated unions.
- Important cases such as "conversion patterns" (i.e., patterns such as `(|Complex|)`) do not occur any overhead: they are just function calls that can be inlined and optimized as usual.

In addition, we know of several techniques that should, in theory, substantially improve the performance of patterns but which we have not yet implemented. In particular, one performance consideration is the representation used for return results of patterns. The current F# implementation uses:

- `null` for a failing partial pattern (i.e. `None` is represented as `null`);
- a boxed value for a succeeding partial pattern (i.e. `Some(1)` results in a boxed integer);
- a simple unboxed value for single-tag total patterns like `Complex`;
- a boxed tagged value such as `Choice3_1(1)` for multi-tag total patterns .

Tuples in return values also currently require an extra allocation. This means the current implementation does perform allocations on many pattern calls.

However, an easy technique that will eliminate nearly all allocations is available to us: .NET supports type-safe *structs*, i.e. types whose representation is not a heap-allocated GC pointer but rather an inline collection of values, generally immutable and copied as needed. While the F# compiler doesn't yet use structs for options, choices and tuples, it is clear that these are excellent candidates to do so. This may also bring other performance benefits to F# code.[12] However such a change must be thoroughly performance tested as it has ramifications well beyond the scope of this paper.

6. Feature Interactions and Future Work

In this section we look at how active patterns interact with some related language features. These features do not currently exist in F#, though some are likely to be added in due course. However, this paper aims to make a contribution relevant to languages other than F#, and hence we consider it essential to think through potential feature interactions in OCaml, Haskell and other statically typed functional languages.

6.1 Types for Recognizers

The types we have given for patterns use an encoding of anonymous unlabeled sum-types tagged by the name `choice`:

```
val (|Cons|Nil|): 'a llist -> ('a * 'a llist,unit) choice
```

However, unlabeled sum types are not a particularly useful extension to functional languages. It is evident that OCaml-style polymorphic variant types would be useful here:

```
val (|Cons|Nil|) :
    : 'a llist -> [ `Cons of ('a * 'a llist) | `Nil ]
```

This raises the question: could an active pattern mechanism be built entirely in terms of the tag information in a labeled sum type? This appears difficult without some kind of syntactic extension, but is an open question and is an interesting, especially for the OCaml community.

[12] The designers of Nemerle (Nemerle 2006) have reported corresponding performance improvements for tuples in private correspondence.

6.2 Tag-Bound Existentials and GADTs

Existentials are a natural extension to pattern matching in languages with enriched datatypes (Läufer and Odersky 1992) or subtyping and runtime types. For example, the following is the likely syntax for a proposed extension to F# where type variables can be existentially quantified at pattern matches involving type tests:[13]

```
match obj with
| <'a>    :? List<'a> as l ->  ...
| <'a>    :? 'a[] as arr -> ...
| <'k,'v> :? Dictionary<'key,'value> -> ...
```

This extension is not yet implemented in F#, but is implementable, by using some of the reflection machinery of the .NET Common Language Runtime, and there are many known examples where it would be useful.

But what of active patterns? For example, it would be reasonable to expect to be able to write recognizers that abstract one or more of these patterns:

```
match obj with
| <'a>  AnyListOrArray(l : 'a list) ->  ...
| ...
```

However what is the type of `AnyListOrArrray`? One natural encoding is to permit anonymous existentials as part of the return type of patterns:

```
val (|AnyListOrArray|_|) : obj -?> (∃'a. 'a list)

let (|AnyListOrArray|_|) (obj) : (∃'a. 'a list) =
    match obj with
    | <'a> :? List<'a> as l ->  Some(l)
    | <'a> :? 'a[] as arr -> Some(Array.to_list arr)
    | _  -> None
```

Here we have assumed an extension to the type algebra of the form $\exists\alpha.\ \tau$, and τ_1 `-?>` τ_2 is used as a shorthand for τ_1 `->` τ_2 `option`. We have also assumed an implicit "pack" operation on each branch of the result of the implementation of the active pattern.

Generalized Algebraic Data Types (GADTs) generalize existentials by allowing data construction tags to existentially quantify constraints as well as variables (Xi et al. 2003). Here a natural encoding is again to enrich the type system to ensure that simple function types are rich enough to encompass these constraints. For example, consider the following possible signature for a partial active pattern to match "lambda" nodes in a strongly typed abstract term structure, one of the canonical examples of GADTs:

```
type Expr<'a>  // an abstract type
val (|Lambda|_|)
    : Expr<'a>
      -?> (∃'b 'c. ('a = 'b -> 'c) => Var<'b> * Expr<'c>)
```

Here we have assumed an extension to the type algebra of the form $\exists\alpha.\ C$ `=>` τ, where C expresses equational type constraints, which are sufficient to capture those that correspond to GADT declarations.

While the above approach to existentials and GADTs is plausible, it is also an intrusive addition to a STFL, especially (but not only) with regard to type inference. For this reason it may instead be reasonable to explore non-type-based extensions that only permit the use of existentials as part of the return type of patterns. This is indeed in the spirit of GADTs themselves which draw much of their expressive power by being a limited locale for existential quantification. The logical conclusion of this design is that patterns have a more special status in the language than they currently do in the design we have described.

[13] The F# pattern "`:?` ty `as` id" is a type-test pattern, and if it succeeds it binds id to the input value at the stronger type. In the current F# design no patterns may bind type variables. In the proposed extension the existentials would be witnessed by solving the type tests w.r.t. the runtime type of the input object.

Match expressions:

$$[[\texttt{matchm} < M > e\ \texttt{with}\ rules]] = \texttt{let}\ t{=}e\ \texttt{in}\ [[\texttt{rules}]]_{M,t}$$

Rules:

$$[[p\texttt{->}e\texttt{|}rules]]_{(M,t)} = plus\ [[p]]_{M,t,e}\ [[rules]]_{M,t}$$

$$[[\emptyset]]_{(M,t)} = zero$$

Patterns:

$$[[C\ p]]_{M;t;e} = bind\ (C\ t)\ (\texttt{fun}\ t' -> [[p]]_{M;t';e})$$

$$[[x]]_{M;t;e} = \texttt{let}\ x{=}t\ \texttt{in}\ return\ e$$

where each t is a fresh variable.

Figure 4. Monadic desugaring of simple patterns

6.3 Monadic and Transactional Pattern Matching

So far we have observed that partial patterns are functions of type `'a -> 'b option`. The choice of the `option` is arbitrary and many other types could be used. In particular, it is possible to generalize the return type of a pattern matching function to anything that implements Haskell's `MonadPlus` type class (Tullsen 2000).

In future work we expect to extend F# with support for constrained higher-kinded type parameters. In this case the return type of structured names (`|`A`|_|`) could feasibly be generalized to a `'M when 'M :> MonadPlus`. [14]

Regular pattern matching cannot be immediately adapted to become monadic matching: for example, a monadic matching construct should not necessarily *run* the monadic value produced. Instead we consider a *monadic match* expression of the form `matchm<`ty`>`, where the monad being used is explicitly specified. A `matchm` expression can then be translated into a regular monadic expression using rules such as those in Figure 4. The translation assumes the monad of interest stays fixed throughout the pattern, i.e., that nested patterns match in the same monad as the outer pattern.

Useful instances of `MonadPlus` include lazy lists for backtracking evaluation and the software transactional memory (STM) monad for transactional evaluation (Harris et al. 2005). Different choices of matching monad produce remarkably different semantics for the match block.

For lazy lists the `plus` operation is concatenation. This means that the results of multiple rules are aggregated, which is quite different to the first-rule-succeeds interpretation of simple (option monad) pattern matching.

More interestingly, for STM monads the *zero* operation causes a transaction to re-execute (and potentially block) and the *plus* operation rolls back the effects of the first transaction if it fails and then executes the second. Hence a transaction monad can be used to control the use of side-effects in a pattern by rolling back effects when a pattern fails. Thus at most one rule succeeds. For example, we can use an active pattern to read from two concurrent `MVar` values in a transaction:

```
val (|ReadMVar|_|) : 'a MVar -> 'a STM
let f mv1 mv2 =
  atomically
```

[14] The higher-kinded type parameter generalizes the occurrence of `option`. The constraint `'M :> MonadPlus` dictates that `'M` is an instance of `MonadPlus`. We assume the `MonadPlus` type defines the standard members *return*, *bind*, *zero* and *plus*.

```
(matchm<STM> mv1, mv2 with
 | ReadMVar x, ReadMVar 0 -> x
 | _, ReadMVar y -> y)
```

Reading from an `MVar` is a destructive, side-effecting operation. Using the STM monad, if the first match case fails the effect of reading `mv1` and `mv2` is rolled-back before the second match case is evaluated. The monadic interpretation of this code would be semantically equivalent to this Haskell code:

```
f mv1 mv2 = atomically $
  do { x <- readMVar mv1; y <- readMVar mv2;
       guard (y==0); return x } `mplus`
  do { y <- readMVar mv2; return y }
```

7. Assessment and Related Work

This paper has presented the first design for extensible pattern matching to incorporate both partial and total decompositions within the context of a regular, simple and lightweight extension. We have given a description of the language extension along with numerous motivating examples. Finally we have looked at how this feature interacts with other reasonable and related language extensions.

Since this work first began in mid-2006 there has been a mini-explosion in discussions, designs and prototypes of view-like mechanisms in programming languages (Syme 2006a; Emir and Odersky 2007; Rossberg 2007b; Peyton Jones 2007; Jambon 2007). We believe our design achieves the best overall functionality for a simple extension to the core of a statically-typed functional programming language.

Peyton Jones et al. have started a lengthy and useful design note on a possible extensible pattern-matching design for Haskell (Peyton Jones 2007). In this discussion they highlight five features that a view-like mechanism may have in Haskell: the *value input feature*, *implicit maybes*, *transparent ordinary patterns*, *nesting* and *integration with type classes*, the last of which can be seen as a Haskell equivalent of *views as first-class values*. In the context of F#, the design described in this paper effectively has all five of these features, which correspond as follows:

Peyton Jones Classification	*Our Terminology*
Value input feature	Parameterized patterns
Implicit maybes	Partial patterns
Transparent ordinary patterns	Total patterns
Nesting	Nesting of active patterns
Integration with type classes	Patterns as first-class values

To our knowledge no other proposed design in this area achieves this combination of features with a single, simple and consistent extension to the language.

Many of the existing proposals for extensible pattern matching in other languages focus only on partial matching and leave total matching unaddressed (Erwig 1997; Emir and Odersky 2007).

Two recent designs for languages close in spirit to F# are Rossberg's views and ad hoc patterns for HamletS (Rossberg 2007b), and Emir and Odersky's "unapply" or "extractor" methods for Scala (Emir and Odersky 2007). Ignoring differences between object-oriented and functional syntax, the Scala proposal essentially matches the F# design for partial pattern matching, though the potential to combine the mechanism with the rich object constraint and composition system of Scala opens interesting possibilities.

Rossberg's work introduces views as a new type-like definition construct, as in Wadler's initial proposal for views, and partial patterns via a separate extension to pattern matching. In some ways the proposal is richer (e.g., views are named and view aliases are supported), in other ways it appears less satisfactory (e.g., partial patterns and views are distinct mechanisms).

Peyton Jones (2007) and Emir and Odersky (2007) give a good review of related work in this area. Many previous proposals to tackle the problem of pattern matching and abstraction have concentrated primarily on the supporting the definition of either views (Wadler 1987; Burton and Cameron 1993), Okasaki's proposal for Standard ML (Okasaki 1998) or partial patterns (Gostanza et al. 1996; Erwig 1997; Erwig and Jones 2000).

Le Fessant and Maranget (2001); Maranget (2007) have proved the correctness of algorithms for optimizing pattern matching and for pattern incompleteness and redundancy checking : it is very interesting to consider how to extend their techniques to active patterns. Fähndrich and Boyland (1997) have looked at permitting only "statically checked" definitions of patterns in terms of existing patterns (as opposed to defining recognizers by arbitrary functions). Using extensible patterns as first-class values was first proposed by Tullsen (2000), where he also observed the monadic generalization we consider in 6.3, though not its potential application to transactions. Sophisticated forms of XML-specific language constructs and matching have been studied by a range of authors: Hosoya and Pierce (2001); Benzaken et al. (2003).

Acknowledgments

We owe thanks to Simon Peyton Jones, Martin Odersky, Andreas Rossberg, Phil Wadler and Burak Emir for discussions on this topic. We also thank Cedric Fournet, Claudio Russo, Georges Gonthier and Ralf Herbrich for helping with informal assessments of the design and its implementation.

References

V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of 2003 ACM SIGPLAN International Conference on Functional Programming. ACM Press, 2003.*, 2003.

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

Burak Emir and Martin Odersky. Matching Objects with Patterns. In *ECOOP '07*, 2007. To appear.

Martin Erwig. Active patterns. In *Implementation of Functional Languages*. Springer, 1997.

Martin Erwig and Simon Peyton Jones. Pattern Guards and Transformational Patterns. In *Haskell Workshop*, 2000.

Manuel Fähndrich and John Boyland. Statically checkable pattern abstractions. In *International Conference on Functional Programming*. ACM, 1997.

Pedro Palao Gostanza, Ricardo Pena, and Manuel Nunez. A new look at pattern matching in abstract data types. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 110–121, New York, NY, USA, 1996. ACM Press.

Jim Grundy, Tom Melham, and John O'Leary. A reflective functional language for hardware design and theorem proving. *Journal of Functional Programming*, 16(2):157–196, 2006.

Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming*. ACM, 2005.

Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. *ACM SIGPLAN Notices*, 36(3):67–80, 2001.

Martin Jambon. Micmatch. `martin.jambon.free.fr/micmatch.html`, 2007.

Konstantin Läufer and Martin Odersky. An extension of ML with first-class abstract types. In *ACM SIGPLAN Workshop on ML and its Applications, San Francisco, California*, pages 78–91, June 1992.

Fabrice Le Fessant and Luc Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*. ACM Press, 2001.

Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):647–656, 2007.

Erik Meijer and Brian Beckman. XLinq: XML Programming Refactored. `research.microsoft.com/~emeijer`, 2006.

Nemerle. Nemerle website. `nemerle.org`, 2006.

Martin Odersky. Scala website. `scala.epfl.ch`, 2006.

Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Principles of Programming Languages*. ACM, 1997.

Chris Okasaki. Views for Standard ML. In *SIGPLAN Workshop on ML, Baltimore, Maryland, USA*, pages 14–23, September 1998.

Simon Peyton Jones. View patterns: lightweight views for Haskell (wiki entry). `hackage.haskell.org/trac/ghc/wiki/ViewPatterns`, 2007.

Andreas Rossberg. Generalizing layered patterns to conjunctive patterns. `successor-ml.org`, 2007a. Search for "Generalizing Layered Patterns".

Andreas Rossberg. Hamlet S: To Become or Not To Become Successor ML. `www.ps.uni-sb.de/hamlet/hamlet-succ-1.3.0S4.pdf`, 2007b. Appendix B.17 and B.19.

Kevin Scott and Norman Ramsey. When Do Match-compilation Heuristics Matter? Technical Report CS-2000-13, University of Virginia, 2000.

Don Syme. Active patterns in F#. `blogs.msdn.com/dsyme`, 2006a.

Don Syme. Leveraging .NET meta-programming components from F#: Integrated queries and interoperable heterogeneous execution. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 2006b.

Don Syme and James Margetson. F# website. `research.microsoft.com/fsharp`, 2006.

Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation*. ACM, 1997.

Mark Tullsen. First class patterns. In *Practical Aspects of Declarative Languages*. Springer, 2000.

Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*. ACM, 1987.

Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 224–235, New York, NY, USA, 2003. ACM Press.

A. An Example Without Active Patterns

Below is the example from §2.2 without active patterns:

```
open System
let rec formatType (typ : Type) =
  if typ.IsGenericParameter  then
    sprintf "!%d" typ.GenericParameterPosition
  elif typ.IsGenericType ||
     not typ.HasElementType then
    let args = if typ.IsGenericType
               then typ.GetGenericArguments()
               else [| |]
    let con = typ.GetGenericTypeDefinition()
    if args.Length = 0
    then sprintf "%s" con.Name
    else sprintf "%s<%s>" con.Name (formatTypes args)
  elif typ.IsArray then
    sprintf "Array(%d,%s)"
        (typ.GetArrayRank())
        (formatType (typ.GetElementType()))
  elif typ.IsByRef then
    sprintf "%s&"
        (formatType (typ.GetElementType()))
  elif typ.IsPointer then
    sprintf "%s*"
        (formatType (typ.GetElementType()))
  else failwith "MSDN says this can't happen"

and formatTypes typs =
  String.Join(",",Array.map formatType typs)
```