Combining Total and Ad Hoc Extensible Pattern Matching in a Lightweight Language Extension
MSR Technical Report
MSR-TR-2007-33

Don Syme[1], Gregory Neverov[2], and James Margetson[1]

[1] Microsoft Research,
Cambridge, U.K.
{dsyme,jamarg}@microsoft.com
[2] School of Software Engineering and Data Communications,
Queensland University of Technology,
Brisbane, Australia
g.neverov@student.qut.edu.au

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.

Table of Contents

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 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 analyze 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 pseudo-code that sums elements of a list of integers pairwise using pattern matching:

```
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 matching that causes particular problems in this regard.

While this problem has long been recognized ([Wad87,Oka98]), it becomes chronic when STFLs are placed in the context of modern object-oriented programming frameworks such as .NET and Java (as in the case of F#, SML.NET, Nemerle and to some extent Scala [SM06,BKR06,Con06,Ode06]) because abstract types are heavily employed as the fundamental concept of object-oriented encapsulation. This is a problem not only for the *consumers* of object-oriented frameworks, but also to the *authors* of these frameworks, i.e. library designers. The advent of "framework-oriented" languages such as C# has shown that it is important to ensure that language devices both enable and encourage the library designer to author long-lived, maintainable software components. In this context the traditional approach to revealing algebraic data types through abstraction boundaries fixes the usage model of the type to such a degree that their use in the public APIs of framework components is hard to encourage. Indeed, it quickly becomes evident that apart from simple cases such as lists, pairs and options, 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 OO languages such as Java and C# despite proposals in that direction [OW97].

Because of these problems we decided in 2006 to conduct a series of "design, implement and use" experiments to determine a suitable design for extensible pattern matching over abstract types in the context of F# programming, concluding with a concrete extension to the language. This report documents the chosen design and some of the lessons we learned along the way.[3] While some

[3] Aspects of the design were first presented in July 2006 at the WG2.8 workshop, Boston, then a prototype implemented in the F# release of August 2006 and documented on the F# website and blog. User feedback was received throughout this time. This report describes the revised design of March 2007.

details of the design are F#-specific, we are confident that its essence can be transferred into other settings.

The particular contributions of this report are as follows:

- We describe a simple extension to F# that allows programmers to write specially-named functions that act as *active recognizers* for existing types, including both partial (ad hoc) and total recognizers. Active recognizers can be used as first-class values and the inherently algebraic nature of total recognizers allows the compiler to check for match redundancy and incompleteness.
- We demonstrate the use of the active patterns in F# through numerous examples, many of them novel.
- 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.

One particular observation is that the problem of pattern matching takes on an added significance when decomposing generic, heterogeneous data formats such as the untyped structured representations XML used on object-oriented platforms, or untyped representations of term structures. In these cases a significant part of programming is the imposition of an ad hoc, application-specific schema that carves out subsets of the heterogeneous format. The proposed design seems particularly effective in this context. We give examples in Section 3.

2 Background

In this section we cover background on pattern matching in general and the problems that reveal the need for an extensible pattern matching mechanism.

2.1 Patterns are Everywhere

Patterns occur throughout the design of languages such as F#, OCaml and Standard ML — in the words of Hickey, "patterns are everywhere" [Hic06]. Some of the locations where patterns occur in the F# grammar are shown below

```
let pat = expr            // binding

fun pat -> expr           // anon. function values

let f pat ... pat = expr // named function values

match expr with           // match expressions
| pat -> expr
| pat -> expr
| pat -> expr
```

```
try expr                   // exception handling
with
| pat -> expr
| pat -> expr

{ for pat in expr         // sequence expressions
  for pat in expr
  ...
  -> expr }

{ new type with             // object expressions
     member x.M1(pat,...,pat) = expr
     ...
     member x.MN(pat,...,pat) = expr
}
```

In F# patterns are formed using a number of compositional constructs, shown by form and example below.

```
(pat, ..., pat)           -- tuple pattern
[pat; ...; pat]           -- list pattern
[| pat; ...; pat |]       -- array pattern
{ id= pat; ...; id=pat }  -- record pattern
Point(pat, pat)           -- tagged data pattern
pat | pat                 -- either pattern
_                        -- wild pattern
x                        -- variable binding
36                       -- constant pattern
"36"                     -- constant pattern
System.DayOfWeek.Monday   -- constant pattern
:? type                   -- type test pattern
:? type as id             -- type test pattern
null                     -- null test pattern
```

The widespread use of patterns and their compositional nature mean that pattern matching is a major part of the enduring appeal of typed functional languages, and patterns are a fundamental workhorse used throughout F# coding.

2.2 Patterns are Problematic

The primary way to define pattern discriminators in STFLs is through the declaration of a discriminated union type (viz. a "tagged union"). Discriminated union types are declared in F#/OCaml as follows:

```
type expr =
  | Add of expr * expr
  | Mul of expr * expr
  | Var of string
  | Const of float
```

This data type could be used to represent simple arithmetic expressions. The tags of a discrimiated union type (in this case `Add`, `Mul`, etc.) have special significance in the language. Firstly, they are can be used as functions to construct values, e.g.,

```
let e = Add (Var "x") (Const 1.0)
```

Secondly, they can be used as patterns to deconstruct values, e.g.,

```
let eval env e =
  match e with
    | Add (e1,e2) -> eval env e1 + eval env e2
    | Mul (e1,e2) -> eval env e1 * eval env e2
    | Var x -> lookup x env
    | Const n -> n
```

Discriminated union types and pattern matching are linked because only union tags can introduce new patterns to match against. This connection makes pattern matching inflexible.

For example, say that the developer of a library wants to hide the implementation of the `expr` type above by exporting it from a module as an abstract type. An abstract type does not reveal the underlying tags used for a discriminated union type, so users of this module would have no way of doing anything useful with its type. One way for the developer to remedy this is to export a collection of functions that construct and deconstruct values of the abstract type. E.g.,

```
(* abstract type *)
type expr

(* construction functions *)
val makeAdd : expr * expr -> expr
val makeMul : expr * expr -> expr
val makeVar : string -> expr
val makeConst : float -> expr

(* deconstruction functions *)
val tryAdd : expr -> (expr * expr) option
val tryMul : expr -> (expr * expr) option
val tryVar : expr -> string option
val tryConst : expr -> float option
```

The construction functions do not pose any problem as they work just as well as union tags. However the deconstruction functions cannot be used in patterns like their union tag counterparts could. Hence the previously simple and concise `eval` function becomes a series of repeated matches that match against the option type.

```
let eval env e =
  match tryAdd e with
  | Some (e1, e2) -> eval env e1 + eval env e2
  | None ->
  match tryMul e with
  | Some (e1, e2) -> eval env e1 * eval env e2
  | None ->
  match tryVar e with
  | Some x -> lookup x env
  | None ->
  match tryConst e with
  | Some n -> n
  | None -> failwith "should never happen"
```

The problem with deconstruction functions becomes worse when the corresponding pattern matching code would use nested sub-patterns. Consider writing an optimizer for the `expr` language that transforms the data structure based on a collection of rewrite rules, (e.g., the distributive law, $ab + bc = a(b + c)$). Using pattern matching this function is a simple translation of the declarative rules.

```
let optimize x =
  match x with
    | Add (Mul a b) (Mul a' c) when a = a' -> Mul a (Add b c)
    | // more rules...
```

With the deconstruction functions this code becomes an unreadable mess of match blocks which force the programmer to duplicate code at each failure point in a sub-pattern or refactor each rule into its own function.

```
let optimize x =
    match tryAdd x with
    | Some (x1, x2) ->
        match tryMul x1 with
        | Some (a, b) ->
            match tryMul x2 with
            | Some (a', c) when a = a' -> Mul a (Add b c)
            | _ -> // remaining rules...
        | _ -> // remaining rules...
    | _ ->  // remaining rules...
```

To summarize, pattern matching is problematic because—

- it is not extensible;
- it encourages programmers to break abstraction boundaries;
- it cannot evolve: for example, discriminator tags may not be renamed while continuing to support the old "deprecated" tag for backwards compatibility, which is an essential technique heavily used by languages such as C# in order to move library designs forward without catastrophically breaking user code;

– it leads to a discontinuity in programming: programmers start down a path of using pattern matching heavily, and are then forced to abandon the technique in order to regain control over their representations;
– it encourages programmers to use unoptimized term representations (since optimizations would be visible to clients pattern matching against the representation).

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

3 F# Active Patterns By Example

In this section we introduce F# active patterns by example, covering total patterns (which discriminate into one or many cases), partial pattens (which discriminate into one case or failure), and parameterized patterns.

3.1 Simple Total Patterns ("Basic Bananas")

We all know that there are (at least) two equally valid ways of thinking about complex numbers: rectangular and polar coordinates. We use this example to introduce the main technique used in this report: the adoption of some simple notational devices that give rise to special structured names to the functions play the role of *active recognizers* (or just *recognizers* for short). Here are two constructors and two recognizers for the F# type `complex`:

```
let MkRect (x,y) = Complex.mkRect(x,y)
let MkPolar (r,th) = Complex.mkPolar(r,th)

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

The first two lines above define construction functions for these two views, and the last two lines define recognizers for the same. The names of the first two functions are (of course) `MkRect` and `MkPolar`, the names of the second two are `(|Rect|)` and `(|Polar|)`, i.e. structured names that include the "banana" symbols `(| |)`. The four functions can now be used as follows:

```
let mulViaRect c1 c2 =
    match c1,c2 with
    | Rect(ar,ai), Rect(br,bi) -> MkRect(ar*br - ai*bi, ai*br + bi*ar)

let mulViaPolar c1 c2 =
    match c1,c2 with
    | Polar (r1,th1),Polar (r2,th2) -> MkPolar(r1*r2, th1+th2)
```

The use of the tags `Rect` and `Polar` in the `match` constructs of these functions select the recognizers `(|Rect|)` and `(|Polar|)`. Hence structured names introduce new identifiers into the set of valid pattern tags, which previously could only be done by a discriminated union data type declaration.[4]

The types of these recognizers are simply:

```
val (|Rect|) : complex -> float * float
val (|Polar|) : complex -> float * float
```

The selection of recognizers is syntax-directed (based on names and lexical scope) rather than type-directed.

The recognizers are executed as part of the pattern matching process to decompose the inputs `c1` and `c2` into tuples `(ar,ai)`, `(br,bi)`, `(r1,th1)` and `(r2,th2)`, just as if we had written:[5]

```
let mulViaRect c1 c2 =
    let (ar,ai) = destRect c1
    let (br,bi) = destRect c2
    MkRect(ar*br - ai*bi, ai*br + bi*ar)

let mulViaPolar c1 c2 =
    let (r1,th1) = destPolar c1
    let (r2,th2) = destPolar c2
    MkPolar(r1*r2, th1+th2)
```

Recognizers such as `(|Rect|)` are also called "tagged conversions." Patterns can be used in many places in the F# grammar, so the above functions could also have been written:

```
let mulViaRect (Rect(ar,ai)) (Rect(br,bi)) = MkRect(ar*br-ai*bi, ai*br+bi*ar)
let mulViaPolar (Polar(r1,th1)) (Polar(r2,th2)) = MkPolar(r1*r2, th1+th2)
```

Ignoring issues related to floating-point precision, `mulViaRect` and `mulViaPolar` define the same computations via different views. In passing we note that the performance of the functions will depend on the concrete representation used for the input and output complex numbers. However, the input concrete representation has *not* syntactically dictated the representations used in intermediary steps of the client algorithms, as it would if we used pattern matching over the concrete structure.

---

[4] Recognizers can be given the same names as functions, i.e., `(|Polar|)` and `Polar`, but for clarity we have not done that here, and the technique tends not to be used in practice.

[5] Here `destRect` and `destPolar` are assumed to have the same definitions as `(|Rect|)` and `(|Polar|)`.

3.2 Decomposition with Recognizers ("Banana Splits")

Since F# is a dual functional and object-oriented language, there is often a need to engage in the use of class types – either types authored in another .NET language, including the base class libraries, or written in F# itself. Traditional pattern matching cannot be used in these circumstances despite the fact that a discriminated union decomposition of a class hierarchy might be a natural and intuitive way to structure code in a functional setting. One such example is the `System.Type` type from the .NET base class library which represents reified runtime types and is used throughout the Reflection and code generation libraries of .NET. The .NET API being used here is as follows:

```
type System.Type
    with
        member IsGenericType : bool
        member GetGenericTypeDefinition : unit -> Type
        member GetGenericArguments : unit -> Type[]
        member HasElementType : bool
        member GetElementType : unit -> Type
        member IsByRef : bool // nb. ByRef = ‘‘managed pointer’’
        member IsPointer : bool // nb. Pointer = ‘‘unmanaged pointer’’
        member IsGenericParameter : bool
        member GenericParameterPosition : int
```

Note that the algebra includes structured generic types and free generic type parameters, though no binding constructs.[6] We can now define a recognizer that captures the essential algebraic structure of `System.Type` values:

```
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, [| |])
    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"
```

Here the name `(|Named|Array|Ptr|Param|)` is a structured name for a single function. The name includes multiple tags surrounded by `(| |)` banana marks and separated by `|` marks (a "banana split"). The use of multiple tags indicates that the function performs simultaneous case-decomposition and data-decomposition. The declaration brings into scope four recognizer tags that can be used within pattern matching, e.g. to pretty-print a `System.Type`:

[6] Generic type parameters are bound at method and class definitions.

```
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 in a `System.Type`:

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

let freeVars typ = freeVarsAcc typ []
```

Note that the tags defined in a structured name (e.g., `Named`) are used on the right-hand-side of its `let`-bound definition to tag the different result cases of the recognizer. Outside of the `let` the tags may only be used in patterns.

The recognizer effectively allows us to view `System.Type` values as if they had been defined using the following algebra:

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

Total recognizers have "identity" in the sense that the F# compiler performs redundancy and completeness analysis for patterns involving uses of total recognizers. We return to this issue in Section 6.1.

As a second example we consider an recognizer for F# lazy lists.

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

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
```

Types The F# types of an recognizer reveal the encoding of result types as an anonymous sum type. For example:

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

Here the type constructor `Choice` represents an F# encoding of anonymous sum types. Like `Tuple`, `Choice` is a special type constructor known to the F# compiler that is compiled to cascading instances of a fixed number of concrete `Choice`N types defined in the F# library:[7]

```
type Choice<'a,'b> =
  | Choice2_1 of 'a
  | Choice2_2 of 'b

type Choice<'a,'b,'c> =
  | Choice3_1 of 'a
  | Choice3_2 of 'b
  | Choice3_3 of 'c

...
type Choice<'a,'b,'c,'d,'e,'f,'g>  =
  ...
```

We return to the question of types for active patterns in Section 7.

3.3 Partial Recognizers ("Banana Slices")

Our examples so far have been of *total* decompositions of types. However, in practice, many types are far too heterogeneous and their decompositions too irregular to make total decompositions the only useful analysis tool. In particular, heterogeneous types such as term structures, XML and strings can be decomposed in many ways, most of which will be irregular and application-dependent.

For this reasons it is essential that any extensible pattern matching technique incorporate the ability to define and use *ad hoc* pattern matching techniques, just as functions allow us to define *ad hoc* construction techniques. Indeed, many of the previous and current proposals for extensible pattern matching in other languages focus only on *ad hoc* matching and leave total matching unaddressed [Erw97,EOW06].

It is useful to begin with contrived but simple examples where we treat the integers as a heterogeneous type. Partial recognizers are defined using identifiers with the form `(|Label|_|)`, e.g.,

[7] At the time of writing the concrete library types must be used directly.

```
let (|MulThree|_|) inp =
    if inp % 3 = 0 then Some(inp/3) else None

let (|MulSeven|_|) inp =
    if inp % 7 = 0 then Some(inp/7) else None
```

Partial recognizers return a value of type *ty* `option` for some *ty*. They can be used as follows:

```
match 28 with
| MulThree(residue) -> printf "3 * %d = 28! Unlikely" residue
| MulSeven(residue) -> printf "7 * %d = 28!" residue
| _ -> printf "no match!"
```

This produces the result `7 * 4 = 28!`.

We next look at some simple manipulations over term structures of the kind often found in theorem proving libraries. Consider the following algebraic data type:

```
type Expr =
  | VarExpr    of string
  | LambdaExpr of ExprVar * Expr
  | AppExpr    of Expr * Expr
```

Partial recognizers can now be defined that correspond to the application of named variables to 1, 2, 3 or N arguments:

```
let (|App1|_|) = function AppExpr(VarExpr(k),x)     -> Some(k,x)        | _ -> None
let (|App2|_|) = function AppExpr(App1(k,x1),x2)    -> Some(k,x1,x2)    | _ -> None
let (|App3|_|) = function AppExpr(App2(k,x1,x2),x3) -> Some(k,x1,x2,x3) | _ -> None
let (|AppN|_|) =
    let rec queryAcc e acc =
        match e with
        | AppExpr(f,x) -> queryAcc f (x::acc)
        | VarExpr (k)  -> Some(k,acc)
        | _ -> None in
    fun e -> queryAcc e []
```

Instances of these recognizers can now be defined that recognize particular constructs:

```
let (|Cond|_|)  = function App3("cond",e1,e2,e3)         -> Some(e1,e2,e3) | _ -> None
let (|Tuple|_|) = function AppN("tuple",e)               -> Some(ty,e)     | _ -> None
let (|Let|_|)   = function App2("let",e,LambdaExpr(v,b)) -> Some(v,e,b)    | _ -> None
let (|Equality|_|)  = function App2("=",e1,e2) -> Some(e1,e2)    | _ -> None
```

Ad hoc additional recognizers can easily be defined for particular term structures:

```
/// Recognize the encoded form of a && b
let (|LazyAnd|_|) x =
    match x with
    | Cond(x,y,Bool(false)) -> Some(x,y)
    | _ -> None

/// Recognize the encoded form of a || b
let (|LazyOr|_|) x =
    match x with
    | Cond(x,Bool(true),y) -> Some(x,y)
    | _ -> None

/// Recognize two beta-reducible forms
let (|BetaReducible|_|) x =
    match x with
    | Let(v,e,b)          -> Some((v,e),b)
    | App(Lambda(v,b),e)  -> Some((v,e),b)
    | _ -> None
```

We return to further examples of ad hoc matching in the Section 5.2.

3.4 Parameterized Recognizers ("Scrap Your Banana Plate")

When using recognizers it quickly becomes apparent that it is very useful to be able to parameterize recognizers, e.g. to express queries such as "Split a string at character N", "Match an attribute A on an XML Node" or "Match any term involving a call to function M". For illustrative purposes we continue the somewhat contrived example from the previous section, where we had

```
let (|MulThree|_|) inp =
    if inp % 3 = 0 then Some(inp/3) else None

let (|MulSeven|_|) inp =
    if inp % 7 = 0 then Some(inp/7) else None
```

These can be replaced by a single parameterized recognizer:

```
let (|MulN|_|) n inp =
    if inp % n = 0 then Some(inp/n) else None
```

The only remaining question is syntax for parameters at pattern usage points. For the current F# release we have chosen the following syntax, which is the most concise imaginable, though has also been criticized because the distinction between expression arguments and pattern arguments is subtle:[8]

[8] For the moment the F# syntax is considered "provisional" and the compiler will give a warning to this effect. Peyton-Jones has suggested the syntax `(`*`expr -> pat`*`)` for Haskell [Joc07].

```
match 28 with
| MulN 3 residue -> printf "3 * %d = 28! Unlikely" residue
| MulN 7 residue -> printf "7 * %d = 28!" residue
| _ -> printf "no match!"
```

Regardless, syntactic issues should not obscure the fact that parameterized patterns are a useful device, fitting neatly into the existing STFL techniques for abstraction and code reuse. We will see many examples of parameterized ad hoc matching in the Section 5.2.

Total recognizers may also be parameterized, e.g. the following recognizer views an unsigned 32-bit integer through a rotation of n bits.

```
let (|Rotated|) n (x:uint32) = (x >>> n) ||| (x <<< 32 - n)

begin match 0x04030201u with
| Rotated 8 0x01040302u -> printf "yes!"
| _ -> printf "no! no! no!"
end
```

Passing parameters to recognizers results in the loss of "identity" for the recognizer, and the F# compiler will not perform redundancy or completeness analysis for patterns involving uses of parameterized recognizers (see Section 6.1). While we are unlikely to revise this choice for F# it is possible to consider performing this analysis up to some equality relation for expression parameters, perhaps as an optional compiler feature.

3.5 Recognizers as First-Class Values ("First-Class Bananas")

The next technical point of the F# active pattern design is a simple one: recognizers are first class values. 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
```

Given the term structure defined in Section 3.3, it is reasonable to define the partial recognizer:

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

A total recognizer 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 recognizers really are just "values with structured names."

3.6 "Both" Patterns

We present one final extension of the standard STFL model of pattern matching. 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 [Ros07a], the natural dual to "either" patterns are "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 the use of "both" patterns in the Section 5.2. For now, here is a simple example showing how to check the value of two particular bits:

```
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 ""
```

4 Operational Semantics

In this section we give a model operational semantics for pattern match evaluation. Since the semantics are not generally difficult, we avoid the traditional approach of using inference rules. Instead we present a simple interpreter for pattern matching written as an OCaml/F# program using only well-founded recursion, pure lambda calculus constructs and simple data types.[9] We also do

[9] An inference rule presentation is trivial to derive from the one we give, but inference rules are considerably harder to type check, debug and maintain than a simple interpreter.

not give the semantics for a full calculus, but rather only the relevant pattern matching portion of the dynamic 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 recognizer and the position of the tag in the tag set of the recognizer.

In Figure 3 we give the implementation of a function that matches patterns against values. 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 Some, None, Choice1, Choice2 etc. tag. 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 (e.g. Rossberg [Ros07b]). We think this helps make patterns more readable and understandable, but it also reduces expressiveness, and may be reconsidered at a later point in our design.

We do not give a corresponding static semantics, since the type-checking is a simple extension to normal type-checking rules for patterns with an additional case for recognizers that follows the form of the case for the dynamic semantics.[10]

4.1 Optimization Assumptions

The above operational semantics shown is naive and will lead to inefficient execution due to repeated invocations of active discriminators. This is unacceptable. Okasaki has suggested that the only sensible semantics to apply to pattern match execution in the presence of side effects is to require that active discrimination functions get run at most once against any given input [Oka98]. While we appreciate this opinion in the context of the stated design goals of Standard ML, it is in many ways the opposite of the approach we take here, where we assume we are interested in pattern matching algorithms that can freely run active discriminators multiple times.

The reason we have taken this approach is that we believe that for F# it may eventually be sufficient in practice to simply fix the pattern matching algorithm used in the presence of recognizers once and for all. Until the algorithm is fixed

[10] The static semantics is simple as long we do not include specifications of redundancy and incompleteness checking, which do not normally form part of the specification of pattern matching and are rather compiler-specific features added to enhance programmer productivity. We return to this question in section 6.1.

```
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 concrete data pattern.
    | PActive of tag * exprs * pat // An active pattern
    | PEither of pat * pat         // 'pat1 | pat2'
    | PBoth of pat * pat           // 'pat1 &&& pat2'
    | 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
```

Fig. 1. Input terms and values for the operational semantics

```
val applyExpr : env -> state -> expr -> exprs -> value -> state * value

type activeDiscriminatorInfo =
   | Total of int * int
   | Partial

val resolveActiveTag : env -> string -> expr × activeDiscriminatorInfo

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

Fig. 2. Assumptions and preliminary definitions

```
let (&&&) f1 f2 (s,binds) =
    let s,bindsOpt = f1 (s,binds)
    match bindsOpt with
    | None -> s,None
    | Some(binds) -> f2 (s,binds)

let (|||) f1 f2 (s,binds) =
    let s,bindsOpt = f1 (s,binds)
    match bindsOpt with
    | None -> f2 (s,binds)
    | Some(binds) -> (s,Some(binds))

// val matchPat : env -> state -> binds -> pat -> value -> state * binds option
let rec 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)
    | PTag (s1,p'), VTag(s2,v') when s1 = s2 ->
        matchPat env p' v' (s,binds)
    | PActive (nm,args,p), _ ->
        let f,info = resolveActiveTag env nm
        let s,v' = applyExpr env s f args v
        match info with
        | Total(numChoices,choiceNum) ->
            if numChoices = 1
            then matchPat env p v' (s,binds)
            else matchPat env (PTag ("Choice"^string_of_int choiceNum,p)) v' (s,binds)
        | Partial ->
            matchPat env (PTag("Some",p)) v' (s,binds)

    | PEither (p1,p2),_ -> (matchPat env p1 v ||| matchPat env p2 v) (s,binds)
    | PBoth(p1,p2)   ,_ -> (matchPat env p1 v &&& matchPat env p2 v) (s,binds)
    | PWild,_ ->  (s,Some binds)
    | PId nm,v -> (s,Some ((nm,v)::binds))
    | PConst c1, VConst c2 when c1 = c2 -> (s,Some binds)
    | _ -> (s,None)
```

Fig. 3. Pattern Matching: Operational Semantics

we are content to simply give strong informal guidelines about the assumptions that characterize the range of feasible pattern matching algorithms we are willing to contemplate.

The particular assumptions that we make are:

- We assume recognizer invocations will return observationally equivalent results on observationally equivalent inputs;
- We assume that all recognizer invocations within the evaluation of a set of rules are commutative, i.e. that observationally results will be returned regardless of the order of invocation of recognizers ;
- We assume that recognizer invocations do not have additional observable side effects after their first executions against observationally equivalent inputs.

This raises the question as to the notion of observational equivalence assumed. The informal specification we rely on is as follows: two expressions are observationally equivalent if no F# program context (i.e. a program with a hole in it) can distinguish between the results under any interpretation of under-specified aspects of F# execution. As with parts of all modern programming languages some parts of the execution of .NET programs are under-specified, e.g., orderings in the memory model in the presence of concurrency, the argument evaluation order, the results of "backdoor" features to access .NET representations such as unsafe C-style code and pointer equivalence checks, and the use of .NET reflection to access the compiled version of the program itself. A formal statement of this property is possible in theory, but is difficult in practice given the breadth of features supported by .NET Common IL code, including the ability to access a multitude of sophisticated and under-specified features such as reflection, libraries, concurrency and asynchronous I/O.

In practice, this means that the F# pattern compiler is effectively assuming that recognizers do not have side effects, or if they do then these effects are commutative and do not re-occur on subsequent re-executions. This places a semantic burden on the library designer, particularly if any side-effects are used in active patterns.

5 Further Examples of Active Patterns

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

5.1 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:

```
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

let rec jlist_to_list xs =
    match xs with
    | Cons (y,ys) -> y :: jlist_to_list ys
    | Nil () -> []
```

The definition of the `(|Cons|Nil|)` total recognizer is syntactically very close to the corresponding *view* definition as proposed by Wadler [Wad87]. This is pleasing: the recognizer being defined can be used within its own definition, and type inference works effectively for these definitions.

5.2 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 recognizers for fragments of XML. We focus on recognizers that traverse the immediate structure of XML nodes, rather than query operators. We believe this is just an initial step toward applying extensible pattern matching in this domain, and the talented programmer is free to define suitable new recognizers, perhaps based on advanced query tools that may be implemented by existing XML libraries such as XLinq [MB06].

```
open System.Xml
open System.Collections
open System
open System.Collections.Generic

let (|Child|_|) name (x: #XmlNode) =
    match x.Item(name) with
```

```
    | null -> None
    | res -> Some(res)

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

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

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

let parsePair (splitchars : string) (str : string) =
    match str.Split(splitchars.ToCharArray()) with
    | [| a; b |] -> Some(a,b)
    | _ -> None

let (|Num|_|)    s = try Some (Int32.of_string s) with _ -> None
let (|Float|_|)  s = try Some (Float.of_string s) with _ -> None
let (|NumHex|_|) s = try Some (Int32.Parse(s, NumberStyles.HexNumber)) with _ -> None
let (|Char|_|)   (s:string) = try Some (s.[0]) with _ -> None
let (|Pair|_|)   s = parsePair "," s
let (|PairX|_|)  s = parsePair "x" s
```

Example use:

```
open System.Xml
open XmlPatternCombinators

type scene =
    | Sphere of float * float * float * float
    | Intersect of scene list

let (|Vector|_|) = function
                   | (Attr "x" (Float x) &
                      Attr "y" (Float y) &
                      Attr "z" (Float z)) -> Some(x,y,z)
                   | _ -> None

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 ]
```

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

let inp = "<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>"
let doc = new XmlDocument()
doc.LoadXml(inp)
//print_endline doc.DocumentElement.Name
printf "results = %A" (parse doc.DocumentElement)
```

A second example is given in the appendix.

5.3 Quotations

F# allows a form of meta-programming where F# code can be reified as values at run-time and manipulated. 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 the code type a 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 perfromed, 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 Grundy's quotation matching in ForteFL [GMO06], and Taha and Sheard's code patterns in MetaML [TS97]. In particular, quotation literals in F# can be used as templates which can be provided as parameters to a parameterized matching function:

```
open Microsoft.FSharp.Quotations.Raw

let rec interp inp =
    match inp with
    | Template <@@. _ + _ .@@> (x,y) -> interp x + interp y
    | Template <@@. _ * _ .@@> (x,y) -> interp x * interp y
    | Int32(x) -> x
    | _ -> failwith "unrecognized"

printf "res1 = %d" (interp <@@ 1+3+3 @@>)
```

In this example the interpretation of the "holes" in a quotation literal are determined by the parameterized matching function `Template` [Sym06b].

6 Implementation

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

6.1 Pattern Match Compilation

For pattern match compialtion F# uses Scott and Ramsey's Generalized Pattern Match Algorithm [SR00] with a simplistic left-to-right heuristic. This has proved effective in practice. Modifying this algorithm to implement a valid interpretation of active patterns was fairly straight-forward.

In this algorithm, the heuristic chooses a *point of investigation* from a collection of *frontiers*. A point of investigation roughly corresponds to a single switch on an integer or tag and is represented by a path, which is a sequence of integers roughly indicating a sub-term of the input term. From this, the frontiers are divided into those edges that are *relevant*, i.e. where information from the investigation may result in the failure of the rule, and those that are *tips*. A decision tree is then constructed that incorporates the test and has subtrees corresponding to *projecting* the success/failure of the investigation through the relevant edge. A default case is added for the tips. The process is then repeated until all frontiers are exhausted, and match incompleteness warnings can be given if a final "dummy" rule is ever exercised.

Modification 1: Choosing the Edge Set In pattern matching without active patterns you can be sure that all edges with any kind of concrete pattern are actually relevant to the investigation. With active patterns this assumption is no longer valid. 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 recognizer. If the recognizer has no *identity*, e.g. is a parameterized active recognizer, then only the first relevant edge is chosen.

Modification 2: Recognizer Identity and Path Identifiers The second modification related to the fact that uses of active recognizers without identity must be considered to have different "paths". 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` recognizer succeeds but its `true` sub-pattern fails then no information is gained about the success of failure of the `false` sub-pattern of `Bit 2 false`. This is because the parameter to the recognizer 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, the notion of *path* was extended so that different instances of parameterized recognizers encountered through pattern match compilation are allocated fresh, unique integers and these integers are used within paths.

Modification 3: Rule Chunking The extensive use of active recognizers (particularly partial recognizers) can quickly lead to significant (even exponential) blow up in the size of decision trees [Oka98]. 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 Ramsey's algorithm 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 recognizers being called more times than may be expected, but reduces code size substantially on some real-world examples.

6.2 Representation of Return Results

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

- We believe that even a naive implementation of the constructs described here increases expressive power sufficiently to justify their inclusion in a language.
- In practice the strong assumptions we have made with regard to recognizers in 4.1 are sufficient to allow single calls to multi-way discrimination functions in the majority of situations.
- Important cases such as "conversion patterns" (i.e., recognizers such as `(|Complex|)`) do not occur any overhead: they are just function calls that can be inlined or subject to whole-program analysis.
- Prior papers in this areas have often only reported a proposed design. We have also reported on an implementation and its use on novel real-world examples. Reporting on performance is thus the subject for later work.

That said, we know of several techniques that should, in theory, substantially improve the performance of the code generated for uses of recognizers but which we have not yet implemented. In particular, one performance consideration is

the representation used for return results of recognizers. For partial recognizers the current F# implementation uses `null` for a failing partial recognizer (i.e. `None` is represented as `null`), a boxed value for a succeeding partial recognizer (i.e. `Some(1)` results in a boxed integer). Single-tag total recognizers return a simple unboxed value. Multi-tag total recognizers return a boxed tagged value such as `Choice3_1(1)`. Tuples in these boxed return values also currently require an extra allocation. This means the current implementation does perform allocations on many recognizer 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.[11] However such a change must be thoroughly performance tested as it has ramifications well beyond the scope of this report.

7 Issues and Feasible Generalizations

7.1 Types for Recognizers

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

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

However, unlabeled sum types are not a particularly useful extension to functional languages. It is evident that OCaml-style labeled sum types might 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 no doubt an interesting one for the OCaml community.

7.2 Multi-discrimination Partial Patterns

A natural extension to the mechanisms defined in this paper is to allow partial recognizers with multiple tags:

```
let (|A|B|C|_|) inp =  ...
```

This extension is fairly straight-forward to add to the system as described. However, we have not yet found cases where this appears particularly useful.

[11] The designers of Nemerle [Con06] have reported corresponding performance improvements for tuples in private correspondence.

7.3 Tag-Bound Existentials and GADTs

Existentials are a natural extension to pattern matching in languages with subtyping and generics, e.g., the following is a natural syntax for an extension to F# where existential types can be quantified at pattern matches involving type tests (in this case the existentials would be witnessed by solving the type tests w.r.t. the runtime type of the input object):

```
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 recognizers that might 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 recognizers:

```
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 $\exists\boldsymbol{\alpha}.\ tau$, τ_1 `-?>` τ_2 is used as a shorthand for τ_1 `->` τ_2 `option`, and we have 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. 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 [SCPD07]:

```
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 $\exists\ C$ `=>` tau, 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 recognizers. 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 recognizers have a more special status in the language than they currently do in the design we have described. Adding recognizers as another "kind" of value is unfortunate but not unparalleled: in many languages there is a distinction between simple values and other syntactic value-like elements such as object or type-class members.

7.4 Generalization to Monadic Pattern Matching

So far we have observed that ad hoc patterns are functions of type `'a -> 'b option`. The choice of the `option` is arbitrary and many other types could be used. As observed by Tullsen it is possible to generalized the return type of a pattern to anything that implements Haskell's MonadPlus type class [Tul00]. Using a MonadPlus the "zero" operation indicates match failure and the "plus" operation composes case alternatives in a match.

In F# syntax a monadic type for patterns would require an extension for higher-kinded polymorphism and a suitable syntax might be:

```
type Pattern<'M,'a,'b> = 'a -> 'M<'b> when 'M :> MonadPlus
```

where the `MonadPlus` constraint ensured the existence of values `'M.zero`, `'M.plus`, `'M.bind` and `'M.return`. A *monadic match* expression `matchm<`*ty*`>` can then be introduced and becomes syntactic sugar for a monadic expression which performs the matching. The desugaring of a simple class of patterns is given by the rules in Figure 4.

For our purposes the `option` type can be considered to be an instance of MonadPlus. Other useful instances of MonadPlus are lazy lists for backtracking evaluation and the software transactional memory monad for transactional evaluation [HMPH05]. For example, interpreting pattern matching under the backtracking monad appears is useful when pattern matching encodes a series of Prolog-style rules. Consider a function that wants to search a list to find zero to three elements that match three given criteria, returned as a lazy list:

```
matchm<LazyList> l with
  | P1 x1 -> x1
  | P2 x2 -> x2
  | P3 x3 -> x3
```

Match expressions:

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

Rules:

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

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

Patterns:

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

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

where each t is a fresh variable.

Fig. 4. Monadic desugaring of simple patterns

The monadic interpretation of the match syntax would now yield a lazy list containing 0, 1, 2 or 3 elements.

Transactions The order of side-effects in a desugaring of moadic pattern matching can be well-specified. As such, one might be tempted to exploit the possibility of side-effects in pattern recognizers to perform shared-state concurrent operations. For example consider a pattern that extracts a value from a Haskell-style MVar, blocking if necessary.

```
type MVar<'a>
val (|ReadMVar|) : MVar<'a> -> 'a // blocks if necessary
let f mv1 mv2 =
    match mv1, mv2 with
    | ReadMVar x, ReadMVar 0 -> x
    | _, readMVar y -> y
```

If initially `mv1` contains 1 and `mv2` contains 2, then evaluating the first match case will remove these values from `mv1` and `mv2` and subsequently fail because 2 <> 0. When the second cases is evaluated `mv2` will be empty causing the pattern to block. What is needed is a way rollback the effects of a failed cases before the succeeding case is executed.

A way to do this is using software transactional memory (STM). In Haskell, STM can be exposed through a MonadPlus [HMPH05]. Let's assume a language is extended with monads, active patterns and a monadic pattern matching integration of the two. Then STMs could be used in monadic pattern matching. The zero operation of STM causes a transaction to potentially block and re-execute

and the plus operation rolls back the effects of the first transaction if it fails and then executes the second.

```
val (|ReadMVar|_|) : MVar<'a> -> STM<'a>
let f mv1 mv2 =
    atomically (matchm<STM> mv1, mv2 with
                | ReadMVar x, ReadMVar 0 -> x
                | _, ReadMVar y -> y)
```

The monadic interpretation of this code would be semantically equivalent to the 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
```

8 Related Work

Since this work first began in mid-2006 there has been a mini-explosion in discussions, designs and prototypes of view-like mechanisms in functional programming languages [EOW06,Ros07b,Joc07,Jam07,Sym06a]. Our represents a novel contribution to this body of work and wee believe it achieves the best overall functionality for a simple extension to the core fundamental statically-typed functional programming model yet proposed.

We can see this as follows. Peyton Jones et al. have started a lengthy and useful design note on a possible extensible pattern-matching design for Haskell [Joc07]. In this discussion Peyton Jones highlights 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 report effectively has all five of these features, which correspond as follows:

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

To our knowledge other proposed designs in this area don't achieve this combination of features, at least with a single, simple and consistent extension to the language.

Two other interesting recent points in the design spectrum for languages close in spirit to F# are Rossberg's views and ad hoc recognizers for HamletS

[Ros07b], and Emir and Odersky's ad hoc recognizers ("unapply" or "extractor" methods) for Scala [EOW06]. Ignoring differences between object-oriented and functional syntax, the Scala proposal essentially matches the F# design for ad hoc pattern matching, though the potential to combine the mechanism with the rich object constraint and composition system of Scala opens interesting possibilities. Rossberg's proposal introduces views as a new type-like definition construct, as in Wadler's initial proposal for views. Various functions can be used as partial recognizers. In some ways the proposal is richer (e.g., views are named and view aliases are supported), in other ways it is weaker (e.g., the introduction of views as a new kind of semantic element complicates the language). We are grateful to authors of both of these papers for productive conversations and insights during the course of this work.

The discussion by Peyton Jones *et al* gives a good review of related work in this area, as does the recent paper by Emir *et al* [Joc07,EOW06]. Many previous proposals to tackle the problem of pattern matching and abstraction have concentrated primarily on the supporting the definition of either views or partial patterns. Notable amongst the proposals for views are Wadler's original proposal for views for Haskell and related extensions and design proposals [Wad87,BC93], Okasaki's proposal for Standard ML [Oka98]. The notion of partial patterns has come up in a number of settings with slight variations [GPN96,Erw97,EJ00]. Fähndrich at al. have looked at statically checked definitions of patterns in terms of existing patterns (as opposed to defining views by arbitrary functions) [FB97]. Using extensible patterns as first-class values was first proposed by Tullsen [Tul00], where he also observed the monadic generalization we consider in 7.4, though not its potential application to transactions.

9 Summary

This report has presented the first design for extensible pattern matching to incorporate both ad hoc pattern matching 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.

Acknowledgements

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

References

[BC93] F. Warren Burton and Robert D. Cameron. Pattern matching with abstract data types. *Journal of Functional Programming*, 3(2):171–190, 1993.

[BKR06] Nick Benton, Andrew Kennedy, and Claudio Russo. SML.NET website. http://www.cl.cam.ac.uk/research/tsg/SMLNET/, 2006.

[Con06] Nemerle Contributors. Nemerle website. http://nemerle.org/, 2006.

[EJ00] Martin Erwig and Simon Peyton Jones. Pattern Guards and Transformational Patterns. In *Haskell Workshop*, 2000.

[EOW06] Burak Emir, Martin Odersky, and John Williams. Matching Objects with Patterns. Technical report, EPFL, 2006.

[Erw97] Martin Erwig. Active patterns. In *Implementation of Functional Languages*. Springer, 1997.

[FB97] Manuel Fähndrich and John Boyland. Statically checkable pattern abstractions. In *International Conference on Functional Programming*. ACM, 1997.

[GMO06] 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.

[GPN96] 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.

[Hic06] Jason Hickey. Introduction to the Objective Caml Programming Language. http://www.cs.caltech.edu/courses/cs134/cs134b/book.pdf, 2006.

[HMPH05] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *Principles and Practice of Parallel Programming*. ACM, 2005.

[Jam07] Martin Jambon. Micmatch. http://martin.jambon.free.fr/micmatch.html, 2007.

[Joc07] Simon Peyton Jones and other contributors. View patterns: lightweight views for Haskell (wiki entry). http://hackage.haskell.org/trac/ghc/wiki/ViewPatterns, 2007.

[MB06] Erik Meijer and Brian Beckman. XLinq: XML Programming Refactored. http://research.microsoft.com/ emeijer/Papers/XMLRefactored.html, 2006.

[Ode06] Martin Odersky. Scala website. http://scala.epfl.ch/, 2006.

[Oka98] Chris Okasaki. Views for Standard ML. In *ML Workshop*, 1998.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Principles of Programming Languages*. ACM, 1997.

[Ros07a] Andreas Rossberg. Generalizing layered patterns to conjunctive patterns. http://successor-ml.org/index.php?title = fGeneralizing_layered_patterns_to_conjunctive_patterns, 2007.

[Ros07b] Andreas Rossberg. Hamlet S: To become or not to become successor ml. http://www.ps.uni-sb.de/hamlet/hamlet-succ-1.3.0S4.pdf, 2007. Appendix B.17 and B.19.

[SCPD07] Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Types in Languages Design and Implementation*. ACM, 2007.

[SM06] Don Syme and James Margetson. F# website. http://research.microsoft.com/fsharp, 2006.

[SR00] Kevin Scott and Norman Ramsey. When Do Match-compilation Heuristics Matter? Technical Report CS-2000-13, University of Virginia, 2000.

[Sym06a] Don Syme. Active patterns in F#. http://blogs.msdn.com/dsyme/archive/2006/08/16/ActivePatterns.aspx, 2006.

[Sym06b] 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*, 2006.

[TS97] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Partial Evaluation and Semantics-Based Program Manipulation*. ACM, 1997.

[Tul00] Mark Tullsen. First class patterns. In *Practical Aspects of Declarative Languages*. Springer, 2000.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Principles of Programming Languages*. ACM, 1987.

A The System.Type Example without Active Patterns

Here is the `formatType` example from Section 3.2 without the use of active patterns:

```
open System
let rec formatType (typ : Type) =

    if typ.IsGenericParameter  then
        sprintf "!%d" typ.GenericParameterPosition
    elif typ.IsGenericType or 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)
```

In comparison, here is the code assuming the definition of the active pattern shown in Section 3.2.

```
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)
```

B A Second Example of Matching XML with Active Patterns

Multiple examples of a technique often help to clarify the difference between what is general and what is application-specific, and to reveal recurring patterns. Below we show a second example of matching in XML using the recognizers defined in 5.2.

```
type GlyphInfo =
     bitmapID : int
      originX : int
      originY : int
      width : int
      height : int
      advanceWidth : int
      leftSideBearing : int

// Match the attributes of a glyph element
let (|GlyphElem|_|) inp =
    match inp with
    |  Attributes
           (Attr "ch"     (Char ch) &
            Attr "code"   (NumHex code) &
            Attr "bm"     (Num bm) &
            Attr "origin" (Pair (Num ox,Num oy)) &
            Attr "size"   (PairX (Num sw,Num sh)) &
            Attr "aw"     (Num aw) &
            Attr "lsb"    (Num lsb)) ->

        Some bitmapID = bm; originX = ox; originY = oy;
             width = sw; height = sh; advanceWidth = aw;
             leftSideBearing = lsb

    | _ -> None

// Look for a number of glyph elements
let (|GlyphElems|) (inp: #XmlNode) = [ for GlyphElem y in inp.ChildNodes -> y ]

// Match the contents of a bitmap element
let (|BitmapElem|_|) inp =
    match inp with
    | Attributes
        (Attr "id" (Num id) &
```

```
          Attr "size" (PairX (Num x,Num y)) &
          Attr "name" name) ->
        Some (id , filename = name; x = x; y = y)
    | _ -> None

// Look for a number of bitmap elements
let (|BitmapElems|) (inp: #XmlNode) = [ for BitmapElem(y) in inp.ChildNodes -> y ]

// Wrap it up by looking for the 'font' element in the top
// structure of the input XML
let parse inp =
    match inp with
    | Elem "font" (Attributes (Attr "base" (Num b) & Attr "height" (Num h))  &
                   Child "bitmaps" (BitmapElems bitmaps) &
                   Child "glyphs" (GlyphElems glyphs)  ) ->
         b,h,bitmaps, glyphs
    | _ -> failwith "not a font file"

// Test the program!
let inp =
  "<?xml version='1.0' encoding='utf-8' ?>
     <font base='20' height='26'>
       <bitmaps>
         <bitmap id='0' name='comic-0.png' size='256x256' />
       </bitmaps>
       <glyphs>
         <glyph ch=' ' code='0020' bm='0' origin='10,10' size='1x27' aw='5' lsb='0' />
       </glyphs>
     </font>"
let doc = new XmlDocument()
doc.LoadXml(inp)
printf "results: %A" (parse doc.DocumentElement)
```