ENHANCING TYPE SYSTEMS WITH FIRST-CLASS PATTERNS

Lutz Hamel, Timothy Colaneri, and Oliver McLaughlin
Dept. of Computer Science & Statistics, University of Rhode Island
Kingston, Rhode Island, USA

ABSTRACT

It has been observed that patterns and in particular first-class patterns can be used to enhance the type systems of procedural languages, especially dynamically typed procedural languages. Here we describe some recent findings of our ongoing research on the impact of first-class patterns on the type system of our Asteroid programming language, a dynamically typed, multi-paradigm programming language that supports first-class patterns. First, we show that first-class patterns allow us to specify subtypes of data types providing a light-weight mechanism for type checking function domains amongst other things. Next, we introduce the idea of a pure constraint pattern which is a pattern that functions like a regular pattern but does not introduce any spurious bindings into the current scope. This is particularly useful for preventing non-linearities in formal function arguments when using first-class patterns. Finally, we demonstrate that first-class patterns can be used to define abstract base types enabling subtype polymorphism. This is particularly powerful in object-based languages like Asteroid that lack inheritance and therefore cannot rely on that mechanism to provide subtype-supertype relationships between user defined types.

KEYWORDS

Pattern Matching, First-Class Patterns, Type Systems, Programming Language Concepts and Design

1. INTRODUCTION

Pattern matching is a powerful programming paradigm which first appeared in applicative languages such as HOPE (Burstall, et al., 1980) in the 1970's and early 1980's to make data structure analysis and decomposition more declarative. It was adopted by functional languages such as Haskell (Peyton Jones, 2003) and ML (Milner et al., 1997) in the 1990's for similar reasons. First-class patterns were introduced into Haskell in the early 2000’s (Tullsen, 2000), and were formally studied in the context of the lambda calculus later that decade (Jay & Kesner, D., 2009). First-class patterns do not increase the computational power of a programming language, but they do increase its expressiveness similar to the way higher-order programming with first-class functions increases the expressiveness of a programming language allowing for brand new ways of solving problems. Here we explore some aspects of this by using first-class patterns to enhance type systems. Today, almost all modern programming languages such as Python, Rust, and Swift incorporate some form of pattern matching.

It has been observed that patterns and in particular first-class patterns can be used to enhance the type systems of procedural languages, especially dynamically typed procedural languages such as Python and JavaScript, e.g. (Bloom & Hirzel, 2012; Kohn et al., 2020). Here we describe some recent findings of our ongoing research on the impact of first-class patterns on the type system of our Asteroid programming language (https://asteroid-lang.org), a dynamically typed, multi-paradigm programming language that supports first-class patterns. Our notion of first-class patterns as types is related to the ideas of first-class dynamic types developed in (Homer et al., 2019).

In this paper we show three applications of first-class patterns as types. First, we show that first-class patterns allow us to specify subtypes of data types providing a light-weight mechanism for type checking function domains amongst other things. Next, patterns can always be thought of as constraints in the sense that some functionality in a program is only invoked when a particular pattern is matched. This was precisely the idea behind the original design of patterns in functional programming languages. However, pattern matching, especially conditional pattern matching always introduces variable bindings into the current scope which might be undesirable due to name pollution of the current scope. Here we introduce our concept of "pure constraint pattern" where patterns behave like regular patterns but do not introduce variable bindings into the current scope. Finally, we demonstrate that first-class patterns can be used to define abstract base types (in the sense of abstract base classes) for both built-in as well as user defined types enabling subtype polymorphism. This is particularly powerful in object-based languages like Asteroid that lack inheritance and therefore cannot rely on that mechanism to provide subtype-supertype relationships between user defined types.

In terms of contributions, besides providing an implementation of these concepts in a working programming language, our ideas of pure constraint pattern and patterns as abstract base types seem to be novel.

2. FIRST-CLASS PATTERNS IN ASTEROID

In Asteroid, first-class patterns are introduced with the keyword 'pattern' and patterns themselves are first-class values that we can store in variables amongst other things and then reference them when we want to use them. Like so,

```
let p = pattern (x,y).
let *p = (1,2).
```

The first let statement assigns a *pattern value* to the variable p and on the left side of the second let statement we dereference the pattern stored in variable p and use it to match against the term (1,2) on the right side. The match introduces the bindings $x \to 1$ and $y \to 2$ into the current scope.

3. DEFINING SUBTYPES WITH PATTERNS

First-class patterns can be used to define subtypes of existing types in a program in a lightweight, dynamic fashion. For example, in Asteroid we can use a first-class pattern to define the natural numbers as a subtype of the integers as follows,

```
let Nat = pattern (x:%integer) if x >= 0.
```

The let statement defines a pattern Nat that, given a value x, checks if that value is of type integer and if its value is non-negative. We can now use this new pattern to restrict the values that can be assigned to a variable much like a type declaration would,

```
let v:*Nat = some_value.
```

This let statement will only succeed if the variable some_value has a value that is compatible with the pattern Nat. If the value is compatible with the pattern, then an appropriate binding of the variable v is inserted into the current scope. We can combine structural pattern matching and patterns that define data types to type check the domain of functions. Consider the factorial, which is only defined on non-negative integers,

```
-- define our subtypes of the integers as patterns
let Pos_Int = pattern (x:%integer) if x > 0.
let Neg_Int = pattern (x:%integer) if x < 0.

-- define function using structural decomposition on integers
function fact
    with 0 do
        return 1
    with n:*Pos_Int do
        return n * fact (n-1).
    with n:*Neg_Int do
        throw Error("undefined for "+n).
    end
```

Here we have a structural decomposition with three cases. The first case matches the constant 0 and returns the value 1. The second case matches the positive integers and performs the recursive factorial computation. Finally, the third case matches the negative integers and raises an exception since factorial is not defined over negative values. Notice that the first-class patterns in the latter two cases act like data types restricting the kinds of values that can be assigned to the formal argument n. The structural decomposition given here is not unlike the kind of structural decomposition one would see in functional languages with one major difference: here the patterns are values!

4. PURE CONSTRAINT PATTERNS

Patterns can always be thought of as constraints in the sense that some functionality in a program is only invoked when a particular pattern is matched. However, pattern matching, especially conditional pattern matching usually introduces variable bindings into the current scope which might be undesirable. Consider the code snippet based on our earlier pattern,

```
let Nat = pattern (x:%integer) if x >= 0.
let v:*Nat = 1.
```

When these two statements are executed, they will produce two bindings in the current scope: v $\rightarrow$ 1 and x $\rightarrow$ 1. The former is an intended binding due to the second let statement where we explicitly assign the value 1 to v. The latter is a spurious binding which is a side effect from evaluating the first-class pattern as part of the assignment in the second statement. Besides cluttering the name space these kinds of spurious bindings can be harmful because they might mask critical variables that are part of the core computation.

To prevent these spurious bindings Asteroid allows the user to turn any pattern into a "pure constraint pattern," that is, a pattern that does not introduce spurious bindings into the current scope. Pure constraint patterns are indicated with the '%[…]%' brackets. For instance, turning our Nat pattern into a pure constraint pattern gives us the program snippet,

```
let Nat = pattern %[(x:%integer) if x >= 0]%.
let v:*Nat = 1.
```

which will only produce a single binding in the current scope: v $\rightarrow$ 1. The evaluation of the variable x is now local to the pattern and that binding is not introduced into the current scope. One of the main applications of pure constraint patterns is the prevention of non-linear patterns in formal arguments of functions when employing first-class patterns. The following function uses the pure constraint pattern Nat we introduced above,

```
let Nat = pattern %[(x:%integer) if x >= 0]%.

function nat_add
    with (a:*Nat, b:*Nat) do
        return a+b.
    end
```

This function accepts two natural numbers and returns their sum. If we hadn't written the Nat pattern as a pure constraint pattern, then its repeated use in the formal argument of the function would introduce a non-linearity: the repeated instantiation of x in the argument pattern. Asteroid does not support non-linear patterns and therefore the program would be rejected. Turns out that even if Asteroid supported non-linear patterns, this program would not work due to the fact that non-linear patterns express constraints on the values bound to the repeated variables – in this case all repeated instances of x need to be bound to exactly the same value which means that nat_add would be limited to computations involving pairs such as (1,1), (2,2), etc.

5. DEFINING ABSTRACT BASE TYPES AS PATTERNS

An interesting application of first-class patterns is the definition of abstract base types enabling subtype polymorphism. The following Asteroid program demonstrates that,

```
load system io.

structure Circle with -- define the circle shape

   data radius.
   function print_shape
      with none do
         io @println ("circle: "+this @radius)
      end
end

structure Rectangle with -- define the rectangle shape

   data a.
   data b

   function print_shape
      with none do
         io @println ("rectangle: "+this @a+","+this @b).
      end
end

-- define abstract base type Shape with Circle and Rectangle as subtypes
let Shape = pattern %[x if (x is %Circle) or (x is %Rectangle)]%.

-- define function in terms of the abstract base type Shape
-- the argument of the function is restricted to values described by the pattern Shape
function print_any
   with s:*Shape do
      s @print_shape().
   end

-- the function print_any is subtype polymorphic
print_any(Circle(10)).
print_any(Rectangle(5,20)).
```

The program defines two shapes, Circle and Rectangle, together with some properties and a member function called print_shape, The member function does nothing else than write out what kind of shape object it belongs to and what the properties of this shape object are. Next, we define the abstract base type Shape as a first-class pattern. It is a pure constraint pattern that tests whether a given object is either a Circle or a Rectangle. In effect, this makes Circle and Rectangle subtypes of Shape. We then define a function called print_any whose argument is constrained to the abstract base type. If the passed-in object is of the correct type, then the function will call the member function print_shape on that object. If the passed in object does not belong to the Shape type hierarchy, then the program will flag an error. The last two lines of the program demonstrate that the print_any function is subtype polymorphic and will generate the following output,

```
circle: 10
rectangle: 5,20
```

We say that the pattern Shape represents an abstract type in that it functions as a type, but you are not able to instantiate any objects based on that type.

6. CONCLUSIONS AND FURTHER WORK

Here we provided a snapshot of ongoing research in the context of first-class patterns viewed as extensions to the type system of Asteroid. First, we showed that first-class patterns allow us to specify subtypes of data types. Next, we introduced the idea of pure constraint patterns which are patterns that function like regular patterns but do not introduce any spurious bindings into the current scope. Finally, we demonstrated that first-class patterns can be used to define abstract base types enabling subtype polymorphism.

One issue that we are concerned with when using first-class patterns is the loss of transparency during pattern matching. Consider pattern matching without the use of first-class patterns,

```
let (x,y) = (1,2).
```

It is clear that the value on the right is matched against the structure on the left with variable bindings x $\rightarrow$ 1 and y $\rightarrow$ 2. Now, if we replace the pattern with a first-class pattern the code becomes,

```
let p = pattern (x,y).
let *p = (1,2).
```

The second line is again a pattern match statement that matches the value on the right to the structure of the pattern. However, the binding of the variables is now somewhat obscured. In order to restore some of the declarative properties of pattern matching we propose the following,

```
let p = pattern %[(x,y)]%.
let *p bind [x,y] = (1,2).
```

Here the bind operator makes it explicit that pattern matching introduces bindings for the variables x and y in the current scope. The bind operator also allows you to rename bindings on the fly which is useful when a first-class pattern appears multiple times in a match statement,

```
let p = pattern %[(x,y)]%.
let (*p bind [x as a, y as b], *p bind [x as c, y as d]) = ((1,2), (3,4)).
```

The pattern on the left of the second line describes a pair of pairs and introduces bindings for variables a, b, c, and d. Of course, in this setting the mechanisms seems to be overkill because the patterns involved are so simple but keep in mind that first-class patterns can become complex as we demonstrated above with the idea of describing an abstract base type using a first-class pattern.

REFERENCES

Bloom, B. and Hirzel, M.J., 2012. Robust scripting via patterns. *ACM SIGPLAN Notices*, *48*(2), pp.29-40.

Burstall, R.M., MacQueen, D.B. and Sannella, D.T., 1980, August. HOPE: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming* (pp. 136-143).

Homer, M., Jones, T. and Noble, J., 2019, October. First-class dynamic types. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages* (pp. 1-14).

Jay, B. and Kesner, D., 2009. First-class patterns. *Journal of Functional Programming*, *19*(2), pp.191-225.

Kohn, T., van Rossum, G., Bucher II, G.B. and Levkivskyi, I., 2020, November. Dynamic pattern matching with Python. In *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages* (pp. 85-98).

Milner, R., Tofte, M., Harper, R. and MacQueen, D., 1997. *The definition of standard ML: revised*. MIT press.

Peyton Jones, S. ed., 2003. *Haskell 98 language and libraries: the revised report*. Cambridge University Press.

Tullsen, M., 2000, January. First Class Patterns? *In International Symposium on Practical Aspects of Declarative Languages* (pp. 1-15). Springer, Berlin, Heidelberg.