TOWARDS PROGRAMMING WITH FIRST-CLASS PATTERNS

Lutz Hamel[*], Timothy Colaneri[*], Ariel Finkle[+], and Oliver McLaughlin[*]
[*]*Dept. of Computer Science & Statistics* [+]*Dept. of Life Sciences*
University of Rhode Island
Kingston, Rhode Island, USA

ABSTRACT

Pattern matching is a powerful programming paradigm which first appeared in functional programming languages to make data structure analysis and decomposition more declarative. Promoting patterns to first-class status does not increase the computational power of a programming language, but it does increase its expressiveness allowing for brand new ways of solving problems. First-class patterns were studied in the context of the lambda calculus. Today, almost all modern programming languages incorporate some form of pattern matching. However, with only a few exceptions, all programming languages we are aware of that support pattern matching stop short of treating patterns as first-class citizens. Consequently, many interesting use cases of pattern matching lie beyond the reach of those languages. We have implemented first-class patterns in Asteroid, a dynamically typed, multi-paradigm programming language, in order to assess and experiment with first-class patterns. Here we report some of our initial findings. The idea of first-class patterns is not new but we feel that the insights provided here are novel and highlight the impact that first-class patterns can have on programming languages and the discipline of programming itself.

KEYWORDS

Pattern Matching, First-Class Patterns, Programming Language Design, Programming

1. INTRODUCTION

Pattern matching is a powerful programming paradigm which first appeared in functional programming 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) in the 1990's for similar reasons. Promoting patterns to first-class status does not increase the computational power of a programming language, but it does increase its expressiveness allowing for brand new ways of solving problems. First-class patterns were introduced into Haskell in the early 2000's (Tullsen, 2000), and formally studied in the context of the lambda calculus later in that decade (Jay & Kesner, D., 2009). Today, almost all modern programming languages such as Python (Kohn, et al., 2020), Rust (Matsakis & Klock II, 2014), and Swift (Apple Inc., 2020) incorporate some form of pattern matching. However, with only a few exceptions like Haskell, Thorn (Bloom & Hirzel, 2012), and Grace (Homer, et al., 2012), the programming languages we are aware of stop short of treating patterns as first-class citizens. Consequently, many interesting use cases of pattern matching lie beyond the reach of those languages.

Here we describe some of the insights we gained from implementing and programming with first-class patterns in Asteroid, a dynamically typed, multi-paradigm programming language. The idea of first-class patterns is not new but we feel that the insights provided here are novel and highlight the impact that first-class patterns have on programming languages and the discipline of programming itself.

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

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

The left side of the second let statement dereferences the pattern stored in variable P and uses the pattern to match against the term (1,2) on the right side.

2. FIRST-CLASS PATTERNS: CASE STUDIES

Pattern Factoring. Patterns can become very complicated especially when conditional pattern matching is involved. First-class patterns allow us to control the complexity of patterns by breaking them up into smaller sub-patterns that are more easily managed. Consider the following function foo written in Asteroid that takes a pair of values. The twist is that the first component of the pair is restricted to the primitive data types of Asteroid's type system which we enforce with a conditional pattern introduced with the keyword '%if',

```
function foo with (x %if (x is %boolean) or (x is %integer) or (x is %string),y) do
  println(x,y).
end
```

The complexity of the pattern for the first component completely obliterates the overall structure of the parameter pattern and makes the function definition difficult to read. We can express the same function with a first-class pattern,

```
let TP = pattern with q %if (q is %boolean) or (q is %integer) or (q is %string).

function foo with (x:*TP,y) do
  println(x,y)
end
```

It is clear now that the main input structure to the function is a pair, and the conditional type restriction pattern has been relegated to a first-class sub-pattern stored in the variable TP. Thus, first-class patterns allowed us to factor the function parameter pattern into a main pattern, the pair, and a sub-pattern which expresses the type restriction.

Pattern Reuse. In most applications of patterns in programming languages, specific patterns appear in many different locations in a program. If patterns are not first-class citizens, the developer will have to retype the same patterns repeatedly in the various locations where the patterns occur. Consider the following Asteroid program snippet defining two functions,

```
function fact
    with 0 do return 1
    orwith (n:%integer) %if n > 0 do return n * fact (n-1).
    orwith (n:%integer) %if n < 0 do throw Error("negative value").
end
function sign
    with 0 do return 1
    orwith (n:%integer) %if n > 0 d return 1.
    orwith (n:%integer) %if n < 0 do return -1.
end
```

The first function is the classic recursive definition of the factorial, and the second function is the sign function which maps positive and negative values into 1 and -1, respectively. Here we use a multi-dispatch approach with the appropriate patterns restricting the values that each of the various bodies of the functions can receive. To write these two functions we had to repeat the almost identical pattern four times. First-class patterns allow us to write the same two functions in a much more elegant way,

```
let POS_INT = pattern with (x:%integer) %if x > 0.
let NEG_INT = pattern with (x:%integer) %if x < 0.
function fact
    with 0 do return 1
    orwith n:*POS_INT do return n * fact (n-1).
    orwith *NEG_INT do throw Error("negative value").
end
function sign
    with 0 do return 1
    orwith *POS_INT do return 1.
    orwith *NEG_INT do return -1.
end
```

The relevant first-class patterns are now stored in the variables POS_INT and NEG_INT. These are then used in the function definitions to select the appropriate values for the function bodies.

Running Patterns in Reverse. One of the challenges when programming with patterns is to keep an object structure and the patterns aimed at destructuring that object structure in sync. First-class patterns solve this problem by viewing patterns essentially as "object constructors." In that way, a first-class pattern is used to construct an object structure as well as to destructure it without having to worry about the structure and the corresponding pattern getting out of sync. In order to use a pattern as a constructor, we apply the eval function to it. This turns the pattern into a value from Asteroid's point of view. Free variables in the pattern are bound by the eval function. For example,

```
   let P = pattern with ([a],[b]). -- first-class pattern
   let a = 1. let b = 2.           -- define values for free variables of pattern
   let v = eval P.                 -- use eval to construct a value from pattern
   println v.                      -- print the value
   (lambda with *P do println a. println b) v. -- first-class pattern to deconstruct value
```

The output of the program is,

```
   ([1],[2])
   1
   2
```

The first output line is the value computed by the eval function given the values associated with the variables a and b, and the first-class pattern P. The second and third lines of the output are generated by the lambda function, and are the result of destructuring the value passed to the function.

Notice that the whole program is essentially parameterized over the structure of the pattern. We could easily change the internals of this pattern without affecting the rest of the program.

3. CHALLENGES

Promoting patterns to first-class status brings with it some challenges. For example, when using patterns as constructors they turn out to behave like dynamically scoped functions. This is not very desirable due to the difficulty of predicting runtime behavior. Consider,

```
   let P = pattern with (x,y).
   let x = 1. let y = 2.
   let z = eval P.
```

The evaluation of pattern P captures the variables x and y from the current environment and constructs the value (1,2). This has all the well-documented pitfalls of dynamically scoped functions, considering that x and y can be defined anywhere in the code. One solution we are contemplating is to provide an explicit initializer list to the eval operator for use in the constructor call,

```
   let P = pattern with (x,y).
   let z = eval P with (x=1,y=2).
```

where it would be an error to not provide an initializer for each free variable in the pattern. This extended eval operator is semantically equivalent to something like this,

```
   let P = pattern with (x,y).
   let z = (lambda with (Ptn,x,y) do return eval Ptn) (P,1,2).
```

where the pattern Ptn is evaluated in the context of the local scope of the lambda function.

Another challenge is the visibility of variables declared during a pattern match with a first-class pattern. In order to see this, consider a pattern match using a traditional pattern,

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

Here we can immediately see that the pattern match introduces x and y into the local scope. Now consider using a first-class pattern in order to accomplish the same thing,

```
   let P = pattern with (x,y).
   let *P = (1,2).
   assert(x==1 and y==2).
```

Here it is no longer obvious that the pattern match introduces x and y into the local scope. This is especially true given that pattern definition and usage can be very far apart from each other, even in separate modules. A solution to this we are contemplating is explicit syntax for pattern match statements with first-class patterns such as,

```
   let x,y in *P = (1,2).
```

indicating that the pattern match introduces variables x and y into the local scope. A variation on this might be,

```
   let x as k in *P = (1,2).
```

This indicates that we are only using variable x from the pattern match, but introducing it as the variable k in the current scope. In this case, the variable y stays hidden and is not accessible in the current scope.

4. Related Work

The work most similar to ours is the work by Homer et al. (Homer, et al., 2012) and Bloom and Hirzel (Bloom & Hirzel, 2012). In both cases first-class patterns are introduced into languages that are not strictly functional programming languages. However, our work differs from theirs in many details. For example, Homer et al. consider first-class patterns to be functions in the vein of (Tullsen, 2000) whereas we consider them to be structures. In our case, this allows us to view patterns as constructors. The Thorn language by Bloom and Hirzel has many similarities with Asteroid but also differs in its design philosophy, pattern sub-language, and its view of patterns as constructors. The insights and challenges we addressed here are novel and were not addressed in these works. Active patterns of F# (Syme, et al., 2016) are first-class elements of that language. However, active patterns in F# are essentially anonymous union discriminators and therefore differ substantially from the kind of first-class patterns we discuss here. Pattern synonyms in Haskell (Pickering, et al., 2016) allow the user to declare structural equivalences to a given pattern and use the declared equivalent pattern instead of the original one during pattern matching. Our first-class patterns seem to be more general than that, especially since we can also view our patterns as constructors.

5. CONCLUSIONS

Here we provided a snapshot of ongoing research with the Asteroid programming language in the context of first-class patterns. Promoting patterns to first-class status increases the expressiveness of a programming language and enables novel ways of solving problems. We have illustrated that with our use cases for the Asteroid programming language. First-class patterns also present a set of challenges which we outlined and proposed solutions for. The implementation of these solutions is ongoing research.

REFERENCES

Apple Inc., 2020. *Swift Language Reference.* [Online] Available at: https://docs.swift.org/swift-book/ReferenceManual/AboutTheLanguageReference.html

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., Noble, J., Bruce, K.B., Black, A.P. and Pearce, D.J., 2012. Patterns as objects in Grace. ACM SIGPLAN Notices, 48(2), pp.17-28.

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).

Matsakis, N.D. and Klock, F.S., 2014. The rust language. *ACM SIGAda Ada Letters*, *34*(3), pp.103-104.

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

Pickering, M., Érdi, G., Peyton Jones, S. and Eisenberg, R.A., 2016, September. Pattern synonyms. In *Proceedings of the 9th International Symposium on Haskell* (pp. 80-91).

Solodkyy, Y., Dos Reis, G. and Stroustrup, B., 2013, October. Open pattern matching for C++. In *Proceedings of the 12th international conference on Generative programming: concepts & experiences* (pp. 33-42).

Syme, D. et al., 2016. *The F# Language Specification.* [Online] Available at: https://fsharp.org/specs/language-spec

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