DECLARATIVE PROGRAMMING IN MODERN IMPERATIVE LANGUAGES

Lutz Hamel
Dept. of Computer Science & Statistics
University of Rhode Island
Kingston, Rhode Island, USA

ABSTRACT

Declarative features provide new ways of solving programming problems in imperative programming languages. It is now a given that modern imperative programming languages support declarative features such as first-class functions and higher-order programming, and increasingly pattern matching. To further explore declarative programming and in particular pattern matching in the imperative programming paradigm we designed and implemented the Asteroid programming language. Asteroid is an imperative, dynamically typed programming language that not only supports common declarative features, but also supports pattern matching to a degree not found in many imperative languages and, in particular, it supports first-class patterns. In this short paper we briefly review some common declarative programming techniques and then briefly survey programming in Asteroid with first-class patterns.

KEYWORDS

Declarative programming; pattern matching; first-class patterns; first-class functions; higher-order programming.

1. INTRODUCTION

Declarative features provide new ways of solving programming problems in imperative programming languages. It is now a given that modern imperative programming languages support declarative features such as functions as first-class citizens. This includes anonymous functions and higher-order programming features such as the map function. Also supported by many imperative programming languages are declarative features such as list comprehensions and generator expressions. However, from our perspective the most interesting development is the increasing adoption of pattern matching, a declarative programming technique. This not only includes structural matching on structures like lists and tuples but also conditional pattern matching and pattern matching on objects. The latter is particularly interesting because it stands in direct conflict with the classical object-oriented notion of encapsulation and protection of object members. Pattern matching provides novel ways of solving problems analogous to the way higher-order programming provides novel ways of accomplishing things.

To explore declarative programming and in particular pattern matching in the imperative programming paradigm further we designed and implemented the Asteroid programming language (asteroid-lang.org). Asteroid is an imperative, dynamically typed programming language that not only supports common pattern matching features like the match and let statements, but it also supports pattern matching on function arguments in the style of functional programming languages like ML (Milner, 1997). Furthermore, Asteroid supports first-class patterns (Jay & Kesner, 2009). The latter opens whole new avenues of programming language research such as pattern reusability, patterns as constraints, and patterns as enhancement to type systems. The contributions of this short paper are (1) the placement of our current research into the context of the current declarative evolution of modern imperative programming languages and (2) a brief demonstration of the utility of first-class patterns including our pattern scope operator and our improved conditional pattern matching operator which allows the developer to use patterns as constraints on other patterns.

Our notion of first-class patterns is related to the ideas of first-class dynamic types developed in (Homer et al., 2019) but drastically differs from first-class patterns in Haskell (Tullsen, 2000) where they are treated as

anonymous functions. The first-class patterns in F#, also known as active patterns (Syme, 2020), are limited in their application compared to our more general notion of first-class pattern.

An imperative language similar to ours that also implements first-class patterns is Thorn (Bloom & Hirzel, 2012). Unfortunately, that language is no longer under development. Furthermore, our ideas of patterns constraining patterns as well as our pattern scope operator seem to be novel.

The remainder of the paper is structured as follows. Section 2 is a brief survey of the most common declarative features found in today's imperative programming languages and places our work in the context of those features. Section 3 introduces first-class patterns and highlights some of our findings with regards to programming with first-class patterns. The features discussed here will be available in our upcoming 2.0 release. Section 4 presents our conclusions and points to some interesting directions for further work.

2. DECLARATIVE PROGRAMMING

Declarative programming features are now implemented to varying degrees by almost every modern, popular programming language. We define popularity of a programming language as being in the top ten programming languages on indexes like TIOBE (www.tiobe.com/tiobe-index) and IEEE (spectrum.ieee.org/top-programming-languages-2022) and we define modern by the fact that a programming language was initially designed/release from the 1990's onward. This includes languages like Go (go.dev), Java (www.java.com), JavaScript (Severance, 2012), Python (www.python.org), R (www.r-project.org), Rust (www.rust-lang.org), and Swift (www.swift.org).

The power of declarative features is that they provide new ways of accomplishing programming tasks in imperative programming languages. Consider the task of transforming a list. Higher-order programming techniques allow this to be done without explicit looping as in this JavaScript code snippet that transforms an integer list into a list of squared values by mapping an anonymous function onto the integer list,

```
const numbers = [1, 2, 3, 4, 5];
const squaredNumbers = numbers.map((num) => num * num);
```

Here the anonymous function `(num) => num * num` transforms a number into its square and the map function applies this function to each member of the list producing a new list. The reduce or fold function available in virtually all functional languages and many of today's imperative languages is a variation on the map function. Here, a given list is reduced or folded into a single value. Consider the problem of checking whether all elements on a list are positive values. Instead of writing a loop which traverses the list and checks whether the elements fulfill the constraint, the reduce function allows us to accomplish this with a single expression,

```
Asteroid 2.0.1
(c) University of Rhode Island
Type "help" for additional information
ast> [1, 2, 3, 4, 5] @reduce (lambda with (acc,i) do (i > 0) and acc, true)
true
ast> [1, 2, 3, -4, 5] @reduce (lambda with (acc,i) do (i > 0) and acc, true)
false
ast>
```

In this Asteroid example the first list is reduced to the value true, and the second list is reduced to the value false. List comprehensions and generator expressions are another feature that is available in many modern imperative programming languages. These expressions allow us to construct lists again without explicit looping constructs. The following is a code snippet in Swift demonstrating list comprehensions by computing a list of squares,

```
let squaredNumbers = [number * number for number in 1...5]
```

The construction on the right of the let statement is a list comprehension. In languages that do not support list comprehensions directly, higher-order programming techniques like we have seen in the above JavaScript example can usually be employed in order to achieve similar results.

This brings us to pattern matching, an increasingly important part of declarative programming in imperative languages. For example, the match statement, which supports full structural pattern matching, was added to Python very recently (Kohn et al., 2020). The simplest form of pattern matching is found in let/assignment

statements. The following Rust code snippet illustrates accessing the components of a Point object using pattern matching (also called destructuring),

```
struct Point { x: i32, y: i32 } // define a Point structure
let p = Point { x: 5, y: 10 };  // instantiate a Point object using a constructor
let Point { x: a, y: b } = p;   // destructure a Point object using a pattern
assert_eq!(a, 5);
assert_eq!(b, 10);
```

The construction on the left of the equal sign on the third line of the code above is called a pattern. The power of this declarative programming techniques derives from the fact that a pattern mirrors the structure of the object to be destructured; in this case it is the Point object whose structure is defined on the first line. Notice that the pattern-match instantiates the two variables a and b with the contents of the given object—therefore the notion of destructuring.

In general, more advanced pattern matching techniques are available in match statements. The following Python code snippet demonstrates the use of the match statement applied to a list structure,

```
def f(data):
   match data:
        case []:
            return("Empty list.")
        case [x, *_] if isinstance(x, int)  and x > 0:
            return("List whose first element is a positive integer.")
        case _:
            return("Unknown object.")
```

Here we define a function f that uses the match statement and pattern matching to analyze the list passed to the function. The first case clause uses an empty list pattern to detect whether the data variable contains an empty list. The second case clause is interesting in that the pattern can match non-empty lists of any length and then uses a conditional pattern-match to only match lists whose first element is a positive integer. The last case clause uses a wild-card pattern which matches any object and works as a default case. All the above declarative programming features are also available in our Asteroid programming language.

Before we move on to first-class patterns it is interesting to look at function invocation. The imperative languages mentioned above except for Asteroid look at function invocation as a "function call" with a function name and a list of parameters to pass to the function. This is different from the more mathematically motivated view of function invocation as "function application" in the declarative paradigm. Here a function is applied to a single object. This single object could be a member of a cross-product set to accommodate multivariate functions. The imperative view of function invocation as function call has unexpected implications at function invocation sites. For example, in Python this means that for calls to a function foo which expects two parameters we have `foo(1,2)≠foo((1,2))`, that is, parenthesizing an expression changes its semantics at the function call site. In the declarative view of function invocation as function application to a single object this paradoxical problem does not occur. Consider the snippet of Asteroid which supports function application,

```
Asteroid 2.0.1
(c) University of Rhode Island
Type "help" for additional information
ast> function foo with (a,b) do a+b end. -- define foo
ast> foo(1,2) == foo((1,2))
true
ast>
```

Since Asteroid looks at function invocations as function applications, parenthesizing the object (1,2) has no impact on the semantics of the function application.

3. FIRST-CLASS PATTERNS.

To further explore pattern matching in imperative programming we implemented patterns as first-class entities 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 → 1 and y → 2 into the current scope.

Promoting patterns to first-class status separates pattern definition points from pattern usage points enabling new use-cases for pattern matching. Consider the following Asteroid code snippet,

```
let pos_int = pattern x if (x is %integer) and (x > 0).
let neg_int = pattern x if (x is %integer) and (x < 0).

function fact
   with 0 do return 1.
   with *pos_int do return x*fact(x-1).
   with *neg_int do throw Error("illegal value: "+tostring(x)).
end

function sign
   with 0 do return 1.
   with *pos_int do return 1.
   with *neg_int do return -1.
end
```

The first two lines define first-class patterns for positive and negative integers, respectively. Next, we define the function fact which computes the factorial of an integer. We use pattern matching on the function argument to determine what value to return. Most notably, we use the patterns defined earlier to accomplish this. The first two with-clauses define the usual behavior for a factorial computation. Observe that the constant 0 is also considered a pattern. The last with-clause matches if the function is called with a negative integer and throws an exception indicating so. If the function is called with a value that does not match any of the patterns, then the runtime system will throw an exception indicating that the function was called with a value that the function does not recognize.

Next, we define the mathematical sign function which returns the value 1 for integer input values greater than or equal to zero. Otherwise it returns the value -1. Again we use pattern matching on the function argument and, most notably, we reuse our positive and negative integer patterns.

First-class patterns promote pattern reuse and encourage developing interesting and perhaps complex patterns since they are no longer “one shot deals” but can be reused in many different contexts. We leverage this insight in our current library design for Asteroid where we plan to publish a pattern module containing reusable patterns.

It is interesting to observe that by restricting the scope of the variables instantiated by a pattern to just the pattern itself using the %[…]% scope operator, patterns essentially become constraints and as such behave almost like data types. The latter is particularly appealing in dynamically typed languages like Asteroid as it allows the developer to recover some sort of type safety. The following Asteroid code snippet demonstrates this,

```
let nat = pattern %[x if (x is %integer) and (x >= 0)]%.

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

The first line defines a first-class pattern for the natural numbers. Observe the %[…]% scope operator which ensures that x will not be bound into the current scope when the pattern is used but is only visible within the pattern.

The function definition for nat_add needs some explanation. First observe that the function expects a pair of values given by the pattern (a,b). Next, we use our colon pattern construction which is a shorthand notation for a conditional pattern match: The match of pattern (a,b) is only successful if this pattern matches a pair of natural numbers (*nat,*nat). In that way, the pattern (*nat,*nat) acts like a data type restricting the values that can be passed to the function. Note, here we have patterns constraining other patterns. The pattern (*nat,*nat) also demonstrates that first-class patterns allow us to construct patterns in a modular fashion by assembling complex patterns from simpler ones.

4. CONCLUSIONS

We discussed declarative programming in today's imperative languages. Many declarative features have been seamlessly adopted and enable new problem solutions in the imperative paradigm. One interesting note is that function invocations in imperative languages have resisted the shift to declarative programming. Function invocations are still viewed as function calls with a function name and a parameter list. This leads to paradoxical issues as shown above. We should note that Asteroid supports the more declarative view of function invocation via function application.

Pattern matching is being increasingly adopted by imperative languages and first-class patterns as implemented in the Asteroid programming language are a natural extension to the current declarative programming evolution within modern, imperative programming languages. Pattern reuse and patterns viewed as constraints are interesting applications of first-class patterns.

We are currently implementing a pattern library to be shipped with the next release of Asteroid. One of the more intricate patterns in this library is a pattern that describes integer lists,

```
pattern %[(x:%list) if x @reduce (lambda with (acc,i) do (i is %integer) and acc, true)]%
```

For dynamically typed languages where lists are inherently polymorphic this is an important pattern because it allows for some type safety on lists. Also notice that as a first-class pattern we only have to specify it once and then use it wherever we need it.

Another topic we are exploring are parameterized patterns. That is, patterns that take other patterns as parameters. The integer list pattern above is clearly a good candidate for this, where the data type of the list elements specified as the built-in pattern %integer could be replaced by a pattern parameter specifying the element type.

REFERENCES

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

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).

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.

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

Severance, C., 2012. Javascript: Designing a language in 10 days. *Computer*, *45*(2), pp.7-8.

Syme, D., 2020. The early history of F#. *Proceedings of the ACM on Programming Languages*, *4*(HOPL), pp.1-58.