Pattern-Level Programming with Asteroid

Lutz Hamel
Department of Computer Science and Statistics
University of Rhode Island
Kingston, RI 02881
USA
lutzhamel@uri.edu

Abstract

John Backus identified value-level (object-level) programming languages as programming languages that combine various values to form other values until the final result values are obtained. Virtually all our classic programming languages today including C, C++, and Java belong into this category. Here we identify pattern-level (term-level) programming languages that combine various patterns to form other patterns until the final result patterns are obtained. New patterns are constructed from existing ones by the application of pattern-to-pattern functions exploiting pattern matching and constructors. First-order logic programming languages such as Prolog, OBJ, and Maude belong into this category. Our insight that pattern-level and value-level programming gives rise to a pattern-value duality is used as the foundation of the design of a new programming language called Asteroid. Hallmarks of this new programming language design are the developer's ability to explicitly control the interpretation or model of expression terms and the notion of 'patterns as first class citizens'. In addition to a complete implementation of pattern-level programming Asteroid also supports an object-oriented style of programming based on prototypes and also subject to pattern matching.

Keywords

pattern matching, semantics, programming language design

1. Introduction

Pattern matching is a very powerful and useful device in programming [1].

Abstractly, pattern matching can be defined as:

Pattern matching is the act of checking a given sequence of tokens or structure (the subject) for the presence of the constituents of some pattern.

– Wikipedia

In the context of this definition a pattern does three things [2]:

1. Decide whether a given subject has a certain structure;
2. Extract zero or more pieces;

Listing 1: Basic pattern matching in Asteroid.

```
function postfix
    with (op,cl,cr) do -- match binary node
        return (postfix(cl),postfix(cr),op)
    orwith (op,c) do -- match unary node
        return (postfix(c),op)
    orwith (v,) do -- match leaf
        return (v,)
    end function
```

3. Bind those pieces to variables in a certain context.

The Asteroid code in Listing 1 is an example of pattern matching on function arguments: if a given pattern appearing in a (or)with-clause matches the function input then the corresponding function body is executed. This particular function recursively turns a tree structure written in Lisp-like prefix notation into its corresponding postfix notation. What is implicit in this example is that we are only allowed to pattern match on constructors, that is, functions that represent a structure rather than compute a value.

Value-level programming languages are programming languages that combine various values to form other values until the final result values are obtained. Here we identify pattern-level (term-level) programming languages as opposed to value-level languages that combine various patterns to form other patterns until the final result patterns are obtained. New patterns are constructed from existing ones by the application of pattern-to-pattern functions exploiting pattern matching and constructors.

Our insight that pattern-level and value-level programming gives rise to a pattern-value duality is used as the foundation of the design of a new programming language Asteroid. Hallmarks of this new programming language design are the developer's ability to explicitly control the interpretation or model of expressions terms and the notion of 'patterns as first class citizens'. In the context of the ability to manipulate the interpretation of expression terms we are able to develop an elegant semantics for pattern matching. In addition to a complete implementation of pattern-level programming Asteroid also supports an object-oriented style of programming based on prototypes and which is also subject to pattern-matching.

The remainder of the paper is organized as follows: Section 2. puts our work in the context of related work. We look at pattern-level versus value-level programming in Section 3.. Our notion of pattern-value duality is defined in Section 4.. An outline of the major features of the Asteroid language is given in Section 5.. We make some general observations and talk about further work in Section 6.. In Section 7. we make some final remarks.

2. Related Work

Pattern matching first appeared in functional programming languages such as SASL [3] and HOPE [4] in the 1970's and early 1980's as a way to make data structure analysis and decomposition more declarative. It was adopted by functional languages such as SML [5] and Haskell [6] in the 1990's for similar reasons. Today, many modern programming languages such as Python [7], Rust [8], and Swift [9] incorporate some form of pattern matching into the syntax and semantics of the language (as opposed to offering pattern matching as a module/library add-on, *e.g.* [10]). Furthermore, pattern matching has been studied in different formal computational settings such as the λ-calculus [11, 12] and first-order logic [13]. One of the most comprehensive implementation of pattern matching we are aware of is in the Thorn programming language [2].

If we look beyond functional and imperative programming languages then we find that pattern matching or unification is at the heart of logic programming languages such as Maude [14] and Prolog [15]. Pattern matching is at the core of term rewriting which is considered the operational semantics for equational logic languages like Maude. Unification in Prolog can be viewed as an extended version of pattern matching where not only the pattern is allowed to contain variables but also the subject term.

As useful and powerful as the pattern matching paradigm is, the implementation of pattern matching in most modern programming languages falls short. Here are a few examples,

- With the exception of Thorn none of the present day programming languages support patterns as first class citizens in the same sense that anonymous/lambda functions are now supported by virtually all modern programming languages.

- In most programming languages there is an arbitrary split between constructors that are supported in pattern matching and constructors which are not supported in pattern matching. For example, Python and Swift allow the user to pattern match on tuple and list constructors but not on constant constructors (or expression patterns in Swift terminology). To be fair, Swift does allow constant constructor patterns in a narrow context limited to its 'switch' statement. This arbitrary split between constructors that are supported by pattern matching and those that are not seems to violate the notion of orthogonality in programming language design [16, 17].

- Overly restrictive pattern matching semantics. Consider the following 'let' statement:

  ```
  let (1, y) = (1,2);
  ```

  In Rust this is a syntactically correct program but fails to compile due to being a refutable pattern. This is analogous to saying that 'x = y/z' is a refutable computation because the undefined value due to a division by zero is usually not allowed to be assigned to a variable and therefore the statement should not compile. No programming language implements this in this way. Instead we rely on exceptions being raised in such contexts. Therefore, rather than failing to compile, the 'let' example above should generate a runtime exception if the pattern match fails. The equivalent statement in Python (no 'let' keyword and no semi-colon required) fails due to a 'cannot assign to constants' error indicating that Python treats this statement with an awkward mix of pattern matching and assignment semantics.

- Languages such as Python and Swift support object-oriented programming but do not support pattern matching on objects.

Here we introduce Asteroid, a new experimental language that employs the insight that patterns and values are dual aspects of expression structures and thereby provides a much more integrated view of programming with patterns. This pattern-value duality is most clearly visible with constants that in one instance can be viewed as values in an expression and in another instance as patterns during pattern matching depending on the current interpretation of these structures.

Not only does Asteroid address the problematic areas touched upon above but it also addresses the fact that the fixed underlying interpretation of expression structures in our current generation of programming languages interferes with the full deployment of pattern matching as a programming paradigm. Consider for example the '+' operator. In virtually all modern programming languages this has a fixed, value based meaning which can be extended via overloading but ultimately not really changed. This has consequences for pattern matching in that the fixed meaning of the '+' operator is usually a function other than a constructor and therefore operators such as the '+' operator cannot be used in patterns forcing the developer to forsake the most natural expression of a pattern and implement a desired pattern/structure via some sort of secondary (non-optimal) notation. In contrast our Asteroid language avoids attaching rigid interpretations to operators such as '+' and therefore the following Asteroid 'let' statement can be interpreted as a legal pattern matching statement:

```
let 1 + 1 = 1 + 1.
```

Under Asteroid's default model the right side of the equal sign is interpreted as a term (not a value!), the subject term, and the left side of the equal sign is interpreted as a pattern. We can paraphrase the computation by:

> *Let the expression* $1+1$ *on the right side be interpreted as a term in Asteroid's default model and pattern match it with the pattern* $1+1$ *on the left side.*

In Asteroid default model all expression level symbols are term constructors that can be used to construct term expressions or can be used as patterns. However, the developer can attach a specific behavior or interpretation to individual expression symbols in order to turn expression terms into values. This is not unlike Prolog where terms have no interpretation beyond the Least Herbrand Model term model [18] but can acquire specific interpretations by mapping terms into values, *e.g.* using the 'is' predicate. Consider the following Prolog queries,

```
?- 1 + 1 = 1 + 1.
true
?- 2 = 1 + 1.
false
?- 2 is 1 + 1.
true
?- 1 + 1 is 1 + 1.
false
```

The first two queries demonstrate that in Prolog the '+' symbol has no meaning beyond being a term constructor and therefore the $1 + 1$ term has no meaning beyond being an term structure.

The second set of queries demonstrates that the 'is' predicate assigns a standard algebraic interpretation to operator symbols such as '+' in the right side term, evaluates that term using this interpretation, and then unifies the result value interpreted as a term with the left side term. It is entirely conceivable that one could write a new version of the 'is' predicate that would provide a completely different interpretation of the right side operator symbols.

The idea that a programming language can have multiple interpretations for a set of operator symbols as in Prolog had a fundamental impact on the design of Asteroid. The

Listing 2: Pattern matching and models in Asteroid.

```
1   load "io".
2
3   load "default". -- load default term model
4   let 1 + 1 = 1 + 1.
5   try
6       let 2 = 1 + 1. -- throws an exception
7   catch _ do
8       print "pattern match failed".
9   end try
10
11  load "standard". -- load standard model
12  let 2 = 1 + 1.
13  try
14      let 1 + 1 = 1 + 1. -- throws an exception
15  catch _ do
16      print "pattern match failed".
17  end try
```

program in Listing 2 is Asteroid's equivalent of the above Prolog queries. As we have seen before, under the default term model (loaded on line 3) the 'let' statement on line 4 shows that the entity on the right of the equal sign is interpreted as a structure which can then be pattern matched to the pattern on the left side. The 'let' statement on line 6 throws an exception since the structure on the right cannot be pattern matched to the pattern on the left in the term model. On line 11 we load Asteroid's standard interpretation for arithmetic operators. We show on line 12 that in this standard model the expression '1 + 1' is interpreted as the value two. This value in turn is then interpreted in Asteroid's term model as the term '2' which is then pattern matched against the pattern on the left side of the assignment statement. The last 'let' statement "proves" that the result of '1 + 1' under the standard model is not a structure by throwing a 'pattern match failed' exception. Bear in mind that Asteroid is not a logic programming language. The similarities between Asteroid and Prolog end pretty much here.

By giving the developer the ability to directly manipulate the model/interpretation attached to "standard" operators in Asteroid the confusion and limitations of patterns versus values can be brought under control and it directly addresses the issue of *expression punning* raised in [2]. This ability to have a fully dynamic interpretation of its constructor symbols firmly sets Asteroid apart from Thorn and any of the other modern programming languages such as Python, Rust, and Swift.

3. Pattern-level vs. Value-level Programming

John Backus identified value-level (object-level) programming languages as programming languages that combine various values to form other values until the final result values are obtained. New values are constructed from existing ones by the application of various value-to-value functions [19]. The values are objects that have a hidden internal structure that only becomes explicit during the computational steps when applying a function to a value. Virtually all our classic programming languages today including C [20], C++ [21], and Java [22] belong into this category.

Here we identify pattern-level (term-level) programming languages that combine various patterns to form other patterns until the final result patterns are obtained. New patterns are constructed from existing ones by the application of pattern-to-pattern functions and constructors. Constructors can be viewed as a special case of pattern-to-pattern functions. Patterns (terms) have an explicit structure that can be processed directly through pattern matching during the computational steps of a program. First-order logic programming languages such as Prolog [18], OBJ [23], and Maude [14] belong into this category.

We treat patterns and terms as synonymous since in our view when pattens are fully implemented in a programming language then any pattern can become a term and any term can become a pattern. It is clear that any term can be considered a pattern since a term has structure that can be matched against a subject term. The converse that any pattern can be considered a term is not so obvious because patterns can have variables. However, if the variables appearing in a pattern are bound to term structures then it is clear that a pattern can be considered a term.

Programming languages such as Python [7], Swift [9] and Rust [8] fully support the value-level programming model and some aspects of the pattern-level programming model. Most notably, the "pattern as first-class citizen" is missing from virtually all these languages. The same holds for most declarative languages such as SML [5] that use patterns extensively but lack the ability to manipulate patterns directly.

There is one exception: the scripting language Thorn [2] which of course implements value-level programming but also implements pattern-level programming in an imperative language setting.

An interesting observation is that most modern programming languages (*e.g.* SML, Python, Swift, Rust, Thorn, *etc.*) are value-level programming languages which support some degree of pattern-level programming and that Asteroid is a pattern-level programming language (by default only the term model is available in Asteroid) that also supports value-level programming (by loading the standard model – see Listing 2).

4. The Pattern-Value Duality

As the designers Thorn recognized in their "pattern punning" comments [2], patterns and values are often only disambiguated in the context of a computation. This gives rise to our notion of the pattern-value duality,

1. An expression structure interpreted in a term model such as the Least Herbrand Model [18] or an initial algebra-like model [24] is a term or pattern.

2. An expression structure interpreted in a value-based model such as the standard mathematical interpretation for algebraic operators is a value.

As we have seen, we can use this dual view of expression structures to give an elegant semantics to the 'let' statements appearing in Listing 2. The 'let' statement on line 4 is interpreted in the default Asteroid term model. The right side of the statement is interpreted as the term '1 + 1' whose structure can be matched directly by the pattern on the left. The 'let' statement in Listing 2 on line 12 is interpreted in the standard Asteroid model. Here the right side of the 'let' statement is first evaluated to the value two in this standard model. In preparation to the pattern matching step this value viewed as the constructor '2' is then interpreted in the Asteroid term model and now the pattern of the left side of the 'let' can be applied to the right side for a pattern match step. Many of the other pattern matching operations available in Asteroid can be given a similar semantics based on the pattern-value duality.

A noteworthy consequence of this semantics is that the only things ever associated with variables are term structures. Consider the following Asteroid code,

```
load "standard".
let v = 1 + 1.
```

Here the variable 'v' is a pattern and according to our semantics above the *term* '2' is bound to 'v' during pattern matching. Since term structures are easily reinterpreted under different models there are no semantic difficulties with switching models during the execution of a program.

In Asteroid we can fully exploit pattern-level programming making use of this duality by giving the developer explicit control over the interpretation of structures. This approach is in stark contrast to Thorn where even though the implementation of pattern matching is fairly complete their static interpretation of expressions such '1 + 1' as a value limits pattern-level programming in that language.

5. Asteroid the Programming Language

Asteroid [25] is an imperative style programming language under development that fully supports both value-level and pattern-level programming. It was highly influenced by the minimalistic approach to data structures and object-orientation in the programming language Lua [26]. The focus on readability and the "pythonic" view of programming in Python [7] had a major impact on the syntax of the Asteroid language [27, 28]. The programming language ML [5] had an influence on the function level pattern matching syntax in Asteroid. Many of the semantic issues around pattern matching with first class patterns worked out in Thorn [2] had a direct impact on the design of pattern matching in Asteroid. Finally, the idea of separating term structure from a more value-oriented interpretation was inspired by the Herbrand models in Prolog [18] as well as the initial term algebras in algebraic data type specification [24, 29].

In the following section we will highlight Asteroid functionality. Few if any of the features discussed here are available in languages such as Python, Swift, and Rust. Many of the pattern matching features including patterns as first class citizens in Asteroid are also available in Thorn [2]. However, due to the fact that Thorn has a fixed interpretation of terms many of the model based pattern matching operations Asteroid supports are not available in Thorn.

5..1 Manipulating the Model

We demonstrate how a developer can explicitly manipulate the interpretation of terms. A simple program that manipulates the interpretations of expressions is given in Listing 3. This program prints out the value of the term '4 + 3 - 2' under three different interpretations:

1. Under the default term model (line 4);

2. Under the standard model (line 10);

3. Under the standard model with the interpretations for '+' and '-' swapped (line 26).

Listing 3: Swapping the interpretation of plus and minus.

```
1  load "io". -- load io module
2
3  -- print out the value using the default term model
4  print (4+3-2).
5
6  -- load the standard model
7  load "standard".
8
9  -- print out the value using the standard model
10 print (4+3-2).
11
12 -- save the interpretations
13 let plus_op = __plus__.
14 let minus_op = __minus__.
15
16 -- detach the interpretations from constructors
17 detach from __plus__.
18 detach from __minus__.
19
20 -- reattach in opposite order
21 attach plus_op to __minus__.
22 attach minus_op to __plus__.
23
24 -- print the value of the term using
25 -- the modified standard model
26 print (4+3-2).
```

The Asteroid interpreter is initialized with the term model in place. The load command on line 7 loads the standard model: the model with the usual interpretations for all the standard operator symbols. It should be noted that the standard model supports overloaded symbols (*e.g.*, '+' as an addition as well as string concatenation) as well as type promotion (*e.g.*, the expression '1 + 2.3' will evaluate to the floating point value 3.3). The code from line 12 through line 22 swaps the interpretation of the '+' and the '-' operator symbols. Here the symbols '__plus__' and '__minus__' are the internal names of the corresponding operators. The program generates the following output:

```
__minus__([__plus__([4,3]),2])
5
3
```

Here the first line is the output under the term model (line 4) and shows a dump of the internal term structure of the expression '4 + 3 - 2' in prefix format. The second line is the output under the standard model (line 10). Given the usual interpretation of '+' and '-' the expression '4 + 3 - 2' evaluates to the value 5. The third line shows the output under the modified standard model with the interpretation of '+' and '- swapped (line 26). In this case the expression '4 + 3 - 2' evaluates to the value 3.

5..2 Basic Pattern Matching

The ability of manipulating the interpretation of expression terms allows the developer to pattern match on operator symbols usually reserved for value computations. We saw some of this already in Listing 2 where the '+' operator symbol can be used for pattern matching under the default term model. Listing 4 shows another version of this program where we take advantage of quoted expressions. Quoted expressions allows the programmer to treat expressions as constructor terms in the presence of a model other than the term model

Listing 4: Pattern matching, models, and quoted expressions in Asteroid.

```
1  load "standard".
2  load "io".
3  load "util".
4
5  let 1 + 1 = '1 + 1. -- quoted expression
6  let 2 = eval('1 + 1).
7  let 2 = 1 + 1.
8  try
9      let 1 + 1 = 1 + 1.  -- throws an exception
10 catch _ do
11     print "pattern match failed".
12 end try
```

Listing 5: The Quicksort in Asteroid.

```
1  load "standard".
2  load "io".
3
4  function qsort
5      with [] do
6          return [].
7      orwith [a] do
8          return [a].
9      orwith [pivot|rest] do
10         let less=[].
11         let more=[].
12
13         for e in rest do
14             if e < pivot do
15                 let less = less + [e].
16             else
17                 let more = more + [e].
18             end if
19         end for
20
21         return qsort less + [pivot] + qsort more.
22     end function
23
24 print (qsort [3,2,1,0])
```

and pattern match against that structure as shown on line 5. Quoted expressions can be interpreted in the current model using the 'eval' function as shown on line 6. The remaining program is almost identical to the code in Listing 2.

As we saw in Listing 1, Asteroid supports pattern matching on function arguments in the style of ML and many other functional programming languages. Listing 5 shows the quick sort implemented in Asteroid as another example of this classic style pattern matching. What is perhaps new is the 'head-tail' operator on line 9. Here the variable 'pivot' matches the first element of the list and the variable 'rest' matches the remaining list which is the original list with its first element removed. On lines 15 and 17 we can see that the '+' operator symbols has been overloaded in the standard model to act as a list concatenation operator as mentioned above. As expected, the output of this program is,

```
[0,1,2,3]
```

We can also introduce our own custom constructors and use them in pattern matching. The program in Listing 6 implements Peano addition on terms (`en.wikipedia.org/wiki/`

Listing 6: Asteroid implementation of Peano addition.

```
1   load "io".
2
3   constructor S with arity 1.
4
5   function reduce
6       with x + 0 do
7           return reduce(x).
8       orwith x + S(y) do
9           return S(reduce(x + y)).
10      orwith term do
11          return term.
12      end function
13
14  print(reduce(S(S(0))+S(S(S(0))))).
```

`Peano_axioms\#Addition`) using the two Peano axioms,

$$x + 0 = x$$
$$x + S(y) = S(x + y)$$

Here 'x' and 'y' are variables, '0' represents the natural number with value zero, and 'S' is the successor function. In Peano arithmetic any natural number can be represented by the appropriate number of applications of the successor function to the natural number '0'. On line 3 our program defines the constructor 'S' to represent the successor function. Next, starting with line 5, it defines a function that uses pattern matching to identify the left sides of the two axioms. If either one pattern matches the input to the 'reduce' function it will activate the corresponding function body and rewrite the term recursively in an appropriate manner. We have one additional pattern which matches if neither one of the Peano axiom patterns matches and terminates the recursion. Finally, on line 14 we use our 'reduce' function to compute the Peano term for the addition of 2 + 3. As expected, the output of this program is,

```
S(S(S(S(S(0)))))
```

Observe that due to the fact that here we operate only in Asteroid's default term model, the '+' operator symbol was available to us as a constructor which allowed us to write the Peano addition in a very natural style.

5..3 Pattern Matching in Control Structures

Control structure implementation in Asteroid is along the lines of any of the modern programming languages such as Python, Swift, or Rust. For example, the 'for' loop allows you to iterate over lists without having to explicit define a loop index counter. In this discussion we solely focus on the pattern matching aspects in control structures. We look at pattern matching in 'if' statements, 'while' and 'for' loops, and 'try-catch' statements.

Before we begin the discussion we need to introduce the 'is' predicate which is a built-in operator that takes the pattern on the right side and applies it to the subject term on the left side (not to be confused with the Prolog 'is' predicate). If there is a match the predicate will return 'true' if not then it will return 'false'. Here is a snippet that illustrates the predicate,

```
let true = 1 + 2 is x + y.
```

The subject term '1 + 2' is matched to the pattern 'x + y' which of course will succeed with the variable bindings x $\mapsto$ 1 and y $\mapsto$ 2.

5..3.1 Pattern Matching in 'if' Statements

In Asteroid an 'if' statement consists of an 'if' clause followed by zero or more 'elif' clauses followed by an optional 'else' clause. The semantics of the 'if' statement is fairly standard. The 'if' and 'elif' clauses test the value of their corresponding expressions for the term 'true' and execute their corresponding set of statements if it does evaluate to 'true'. If none of the expressions evaluate to 'true' then the 'else' clause is executed if present.

In order to enable pattern matching in 'if' statements we use the 'is' predicate. We can rewrite the 'reduce' function from Listing 6 using pattern matching in 'if' statements as an illustration,

```
function reduce
    with term do
        if term is x + 0 do
            return reduce(x).
        elif term is x + S(y)  do
            return S(reduce(x + y)).
        else do
            return term.
        end if
    end function
```

One thing to note is that the variable bindings of a successful pattern match are immediately available in the corresponding statements of the 'if' or 'elif' clause.

5..3.2 Pattern Matching in 'while' Loops

Pattern matching in 'while' loops follows a similar approach to pattern matching in 'if' statements. The 'while' statement tests the evaluation of the loop expression and if it evaluates to the term 'true' then the loop body is executed. Again we use the 'is' predicate to enable pattern matching in 'while' loops.

Listing 7 shows a program that employs pattern matching using the head-tail operator in the 'while' expression in order to iterate over a list and print the list elements. Note that the 'if' statement on line 8 is necessary because applying the head-tail operator to an empty list throws an exception. As one would expect, the output of this program is,

```
1
2
3
```

Listing 7: Pattern matching in 'while' loop.

```
1   load "io".
2
3   let list = [1,2,3].
4
5   while list is [head|tail] do
6       print head.
7       let list = tail.
8       if list is [] do
9           break.
10      end if
11  end while
```

Listing 8: Pattern matching in ‘for’ loop selecting substructures.

```
1  load "standard".
2  load "io".
3  load "util".
4
5  constructor Person with arity 2.
6
7  let people = [
8      Person("George", 32),
9      Person("Sophie", 46),
10     Person("Oliver", 21)
11     ].
12
13 let n = length people.
14 let sum = 0.
15
16 for Person(_,age) in people do
17     let sum = sum + age.
18 end for
19
20 print ("Average Age: " + (sum/n)).
```

5..3.3 Pattern Matching in ‘for’ Loops

Of course Asteroid supports ‘for’ loops indexed over integers,

```
for x in 1 to 3 do
    print x.
end for
```

or loops that iterate over lists,

```
for bird in ["turkey","duck","chicken"] do
    print bird.
end for
```

Actually, in the integer example above the loop also iterates over a list because the operator ‘1 to 3’ returns the list ‘[1,2,3]’.

In addition to these canonical examples we can expand the loop variable into a pattern and do pattern matching while we are iterating. This allows us to access substructures of the items being iterated over in a direct and succinct way. Listing 8 shows such a program. The program constructs a list of ‘Person’ structures that consist of a name and an age (line 7). The ‘for’ loop on line 16 iterates over this list while pattern matching the ‘Person’ constructor at each iteration binding the age variable to the appropriate value in the structure. In the loop body it carries a running sum of the age values which it then uses to compute the average age of the persons on the list (line 20). The output of this program is,

```
Average Age: 33
```

We can also use pattern matching on the index variable of a ‘for’ loop to select certain items from a list. Suppose we extend the ‘Person’ structure of the program in Listing 8 with an additional field capturing the sex of a person. The program in Listing 9 does just that. That additional field is then used by the ‘for’ loop on line 11 to select only male members on the list and print out their names. As expected, the output of this program is,

```
George
Oliver
```

Listing 9: Pattern matching in ‘for’ loop used for filtering.

```
load "io".

constructor Person with arity 3.

let people = [
    Person("George", 32, "M"),
    Person("Sophie", 46, "F"),
    Person("Oliver", 21, "M")
    ].

for Person(name,_,"M") in people do
    print name.
end for
```

Listing 10: Basic exception handling in Asteroid.

```
1  load "io".
2  load "standard".
3
4  try
5  let i = 10/0.
6      print i.
7  catch e do
8      print e.
9  end try
```

5..3.4 Pattern Matching in ‘try-catch’ Statements

Excpetion handling in Asteroid is very similar to exception handling in many of the other modern programming language available today. Listing 10 shows an Asteroid program that performs basic exception handling. On line 5 it attempts a division by zero which will throw an exception. The exception is caught by the ‘catch’ clause on line 7 and its value printed on line 8. The output of the program is the value of the exception,

```
[Exception,integer division or modulo by zero]
```

By default, exceptions in Asteroid are pairs where the first component is an exception specifier and the second component is the value of the exception. In Asteroid we can pattern match on the structure of exceptions in the ‘catch’ clause. Listing 11 shows the same program from above where the ‘catch’ clause on line 7 has been modified to match the structure on the exception explicitly. Here we pattern match on the exception specifier and print out the value of the exception. As expected, the output of the program is,

```
integer division or modulo by zero
```

The structure of the exceptions as shown in the previous examples are by convention only and all internally generated exceptions in Asteroid follow that convention. However, there is nothing to prevent the user to create his or her own exception structures and objects and pattern match on them in ‘catch’ clauses. Listing 12 shows a program that throws an exception using the ‘MyException’ constructor on line 6. That exception structure is pattern matched in the ‘catch’ clause on line 7 and its value is printed on line 8. The output of this program is,

```
Hello There!
```

Listing 11: Basic exception handling in Asteroid with pattern matching.

```
load "io".
load "standard".

try
    let i = 10/0.
    print i.
catch ("Exception", v) do
    print v.
end try
```

Listing 12: Exception handling in Asteroid with custom structures.

```
load "io".

constructor MyException with arity 1.

try
    throw MyException("Hello There!").
catch MyException(v) do
    print v.
end try
```

5..4 Pattern Matching on Objects

We introduce Asteroid's objects using the dog example from the Python documentation (`docs.python.org/3/tutorial/classes.html`). Listing 13 shows that Python example translated into Asteroid. Asteroid's object system is prototype based. In Asteroid it is the convention that object members are given as name-value pairs. That also includes function members in addition to data members. On line 8 of our example we define our prototype object with three members: two data members (lines 9 and 10) and one function starting on line 11. Object members are accessed in a Python dictionary style syntax. What makes this truly object-oriented is the fact that when an object function is accessed in the context of a function call, like on line 21, Asteroid generates an implicit object reference as the first argument to the called function. Notice that at the call site (line 21) we only provide a single arguments whereas the function definition (line 11) has two arguments; the first one capturing the object reference. The output of this program is,

```
Fido: [roll over,play dead]
Buddy: [roll over,sit stay]
```

In order to demonstrate pattern matching with object we added a list of dogs to our program. The resulting program in Listing 14 shows this and starting with line 6 we also added code that iterates over the list of the dogs and prints out the names of the dogs whose first trick is 'roll over'. The filtering of the objects on the list is done via pattern matching on the loop variable on line 6.

The pattern matching on objects is straight forward due to the fact that objects like other structures consist of nested constructors. This also includes function constructors. In Asteroid function constructors are purely syntactic in nature. Asteroid does not compute any function closures and therefore only supports dynamic scoping. This makes sense in an enviroment where patterns as first class citizens are also dynamically scoped objects. We are currently experimenting with the idea on being able to pattern match on function constructors.

Listing 13: Object-oriented programming in Asteroid.

```
load "standard".
load "io".
load "util".

constructor Dog with arity 3.

-- assemble the prototype object
let dog_proto = Dog (
  ("name", ""),
  ("tricks", []),
  ("add_trick",
     lambda
       with (self,new_trick) do
         let self@{"tricks"} =
           self@{"tricks"}+[new_trick])).

-- Fido the dog
let fido = copy dog_proto.
let fido@{"name"} = "Fido".

fido@{"add_trick"}("roll over").
fido@{"add_trick"}("play dead").

-- Buddy the dog
let buddy = copy dog_proto.
let buddy@{"name"} = "Buddy".

buddy@{"add_trick"}("roll over").
buddy@{"add_trick"}("sit stay").

-- print out the tricks
print ("Fido: " + fido@{"tricks"}).
print ("Buddy: " + buddy@{"tricks"}).
```

There is an elegant way of rewriting the last part of the code of the example in Listing 14 starting with line 4 using the fact that in Asteroid patterns are first class citizens. In Listing 15 we associate our pattern with the variable 'dog' on line 4. The quote at the beginning of the pattern is necessary otherwise Asteroid will try to dereference the variable 'name' as well as the anonymous variables '_'. We use the pattern associated with 'dog' in the 'for' loop on line 9 to filter the objects on the list. The '*' operator is necessary in order to tell Asteroid to use the pattern associated with the variable 'dog' rather than using the variable itself as a pattern.

5..5 Patterns as First Class Citizens

We have shown in Listing 15 that patterns can be associated with and dereferenced from variables. Listing 16 illustrates that we can also pass patterns to functions where they can be used for pattern matching. Here we define a function 'match' on line 3 that expects a subject term and a pattern. It proceeds to pattern match the subject terms to the pattern using the 'is' predicate and returns whatever the predicate returns. Observe the '*' operator in front of the 'pattern' variable stating that we want to use the pattern associated with that variable. On line 8 we call the function 'match' with subject term '1+1' and pattern '_+_'. The output of this program is the term 'true'.

We can also construct patterns on-the-fly as shown in Listing 17. Here we construct

Listing 14: Pattern matching and object-oriented programming in Asteroid.

```
⋮
-- print out all the names of dogs
-- whose first trick is 'roll over'.
let dogs = [fido, buddy].

for Dog(("name",name),
        ("tricks",["roll over"|_]),
        _) in dogs do
  print (name + " does roll over").
end for
```

Listing 15: Storing Asteroid patterns in variables.

```
⋮
let dogs = [fido, buddy].

let dog = 'Dog(
  ("name",name),
  ("tricks",["roll over"|_]),
  _).

for *dog in dogs do
  print (name + " does roll over").
end for
```

two subpatterns on lines 3 and 4. These two subpatterns are used to construct the full pattern on line 5 when the pattern is evaluated during a pattern match. Finally, we check whether our pattern is assembled correctly on line 7. The output of the program is 'true' meaning our pattern has the same structure as the subject term '1+2+3' on line 7.

A couple of observations:

1. The quotes on lines 3 and 4 are not strictly necessary because we are working in the default term model.

2. The quote on line 5 is necessary because we don't want to evaluate the dereference operators at this point.

3. From this example it is obvious that patterns with dereference operators are dynamically scoped structures. The variables 'cl' and 'cr' on line 5 will capture their closest associations when the pattern is evaluated during a pattern match as on line 7.

With Asteroid's ability to manipulate patterns we can rewrite the program implementing Peano addition from Listing 6. In the rewritten version the pertinent Peano axioms are stored as rules in a rule table which the program will access during execution. Listing 18 shows the rewritten program. Our two Peano axioms appear as rules in the rule table on lines 9 and 10. Note that each rule is written as a pair where the first component is the left side of the corresponding rule and the second component is the right side of the corresponding rule. The left sides of the rules represent the patterns that need to match the subject term and therefore it is not surprising that they are written as quoted expressions. We also need to write the right sides of the rules as quoted expressions because we want

Listing 16: Passing Asteroid patterns to functions.

```
load "io".

function match
    with subject, pattern do
        return subject is *pattern.
    end function

print (match(1+1, '_+_)).
```

Listing 17: Assembling Asteroid patterns on-the-fly.

```
load "io".

let cl = '1 + 2.
let cr = '3.
let pattern = '*cl + *cr.

print (1+2+3 is *pattern).
```

to delay their evaluations until their corresponding patterns have matched an appropriate subject term (see line 18).

The function ‘reduce’ searches through the rule table for a match to the current subject term ‘term’. If a match is found the corresponding right side of the rule is evaluated. If no match is found then the term is returned unmodified. The output of the program is of course the Peano term ‘S(S(S(S(S(0)))))’.

Observe that the variables of the right sides of the rules in the rule table do not need to be preceeded by a ‘*’ dereference operator because we are not in a pattern matching context. There is no ambiguity here on how a variable should be interpreted – it is always to be dereferenced.

This example demonstrates that Asteroid’s ability to manipulate both its model (line 5) and patterns (line 8) allows pattern-level programming (*e.g.* the rule table and ‘for’ loop body) to coexist seamlessly with value-level programming (*e.g.* the ‘for’ loop expression).

5..6 Advanced Model Manipulation

Here we look at a couple of examples involving interesting aspects of model manipulation in Asteroid. The first program in Listing 19 shows how straight forward it is to switch between pattern- and value-level programming in Asteroid. We define a constructor ‘S’ and an increment function ‘inc’ on lines 4 and 6, respectively. We then continue to print out the value of the term ‘S(S(S(0)))’ on line 13 which will print exactly the same way on the output because ‘S’ is a constructor. Next, on line 14, we attach the ‘inc’ function as an interpretation to the constructor ‘S’. We then continue to print out the value of the same term ‘S(S(S(0)))’ on line 15. However, now ‘S’ has an interpretation as an increment function so the value printed to the output is ‘3’. Next, on line 16, we detach the ‘inc’ function from the constructor and then print the same term again on line 17. Since at this point ‘S’ is again just a constructor the output generated is ‘S(S(S(0)))’.

The example in Listing 20 shows that models do not always have to be value-oriented. Instead we can interpret one structure with another. Observe that in this example we do not load the standard model and only work in the default term model. We define our by

Listing 18: Peano addition implementation using a lookup table for the rewrite rules.

```
load "standard".
load "util".
load "io".

detach from __plus__ . -- '+' is a constructor
constructor S with arity 1.

let rule_table = [
    ('x + 0,     'reduce(x)),
    ('x + S(y), 'S(reduce(x + y)))
    ].

function reduce
    with term do
        for i in 0 to length(rule_table) - 1 do
            let (lhs, rhs) = rule_table@[i].
            if term is *lhs do
                return eval(rhs).
            end if
        end for
        return term.
    end function

print (reduce('S(S(0)) + S(S(S(0))))).
```

now familiar constructor 'S' on line 3 and an increment function 'inc' on line 5. Because we did not load the standard model the 'inc' function returns a structure rather than a value ('+' is treated as a constructor). On line 10 we print out the interpretation of the term structure 'S(S(S(0)))' which under the default term model is just the structure 'S(S(S(0)))'. Next we attach the function 'inc' as an interpretation to the constructor 'S' on line 12. On line 13 we again print out the interpretation of term 'S(S(S(0)))'. In this case, because 'S' now has an interpretation, the value is the structure,

```
__plus__([1, __plus__([1, __plus__([1,0])])])
```

Here we can see that we interpreted one structure with another.

6. Remarks and Further Work

As we have seen in the previous section, there is an intricate interplay between the ability to pattern match structures and the kind of model that is used for the structures. If we are using a value-based model (like the Asteroid standard model) then only limited pattern matching and construction is possible because here many of the expression-level constructors and operators tend to represent functions that compute values and therefore are not available for pattern matching and construction. On the other hand, if we choose a term-based model (like the Asteroid default model) then virtually any expression-level constructor or operator is available for pattern matching or construction. The strength of Asteroid is that the developer has complete control over which model to deploy (or create) and therefore has complete control over the amount of pattern- versus value-level programming is available for a particular problem domain.

The problem with our current generation of "general purpose" programming languages like Python, Swift, and Rust is that they have a fixed interpretation of their expression-level structures which limits pattern-matching and in general inhibits the full deployment

Listing 19: Switching back and forth between pattern- and value-level programming in Asteroid.

```
load "standard".
load "io".

constructor S with arity 1.

function inc
    with n do
        return 1 + n.
    end function

-- switch between pattern- and
-- value-level programming
print (S(S(S(0)))).
attach inc to S.
print (S(S(S(0)))).
detach from S.
print (S(S(S(0)))).
```

Listing 20: Interpreting structure with structure.

```
load "io".

constructor S with arity 1.

function inc
    with n do
        return 1 + n.
    end function

print (S(S(S(0)))).

attach inc to S.
print (S(S(S(0)))).
```

of pattern-level programming.

In terms of further work; semantic details such as the scope of a particular model and the scope of a particular attach/detach operation need to be further investigated.

Another issue we would like to explore is to extend models or interpretations to non-arithmetic constructors such as lists. Currently the list contructor '[ ]' has a fixed, term-based interpretation. It would be interesting to be able to attach a semantics other than the term-based model to lists.

As mentioned before, in Asteroid function constructors are purely syntactic objects and therefore it would be interesting to explore the ability to pattern match on them. The non-trivial part here is that unless we restrict ourselves to very simple functions that only compute on expression-level structures we might be forced to be able to pattern match on arbitrary flow of control structures such as 'for' loops and 'if' statements.

We need a more powerful expression parser. Eventhough in Asteroid the models for expression-level structures are under the developer's control the precedence and associativy of the respective operators are fixed in the current parser. We would like to develop a parser that brings all that under the control of the developer in a similar fashion to ISO compatible Prolog implementations using the 'op' predicate to extend the parser.

7. Conclusions

Here we identified pattern-level (term-level) programming languages as languages that combine various patterns to form other patterns until the final result patterns are obtained. New patterns are constructed from existing ones by the application of pattern-to-pattern functions exploiting pattern matching and constructors. Our insight that pattern-level and value-level programming gives rise to a pattern-value duality that was used as the foundation of the design of our new programming language called Asteroid. Hallmarks of this new programming language design are the developer's ability to explicitly control the interpretation or model of expressions terms and the notion of 'patterns as first class citizens'. We have shown that Asteroid supports many pattern-level programming techniques not available in our current generation of programming languages such as Python, Swift, and Rust. We have also shown that Asteroid seamlessly integrates pattern- and value-level programming.

References

[1] L. Augustsson, "Compiling pattern matching," in *Functional Programming Languages and Computer Architecture*, pp. 368–381, Springer, 1985.

[2] B. Bloom and M. J. Hirzel, "Robust scripting via patterns," in *ACM SIGPLAN Notices*, vol. 48, pp. 29–40, ACM, 2012.

[3] D. Turner, "Sasl language manual, st," *Andrews University, Fife, Scotland*, 1976.

[4] R. M. Burstall, D. B. MacQueen, and D. T. Sannella, "Hope: An experimental applicative language," in *Proceedings of the 1980 ACM conference on LISP and functional programming*, pp. 136–143, ACM, 1980.

[5] R. Milner, *The definition of standard ML: revised.* MIT press, 1997.

[6] S. P. Jones, *Haskell 98 language and libraries: the revised report.* Cambridge University Press, 2003.

[7] G. VanRossum and F. L. Drake, *The python language reference.* Python Software Foundation Amsterdam, Netherlands, 2010.

[8] N. D. Matsakis and F. S. Klock II, "The rust language," in *ACM SIGAda Ada Letters*, vol. 34, pp. 103–104, ACM, 2014.

[9] J. Goodwill and W. Matlock, "The swift programming language," in *Beginning Swift Games Development for iOS*, pp. 219–244, Springer, 2015.

[10] Y. Solodkyy, G. Dos Reis, and B. Stroustrup, "Open pattern matching for c++," in *ACM SIGPLAN Notices*, vol. 49, pp. 33–42, ACM, 2013.

[11] J. Barry, "The pattern calculus," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 26, no. 6, pp. 911–937, 2004.

[12] J. Barry, *Pattern Calculus: Computing with Functions and Structures.* Springer, 2009.

[13] G. Roşu, "Matching logic," *Logical Methods in Computer Science (LMCS)*, vol. 13, no. 4, 2017.

[14] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. F. Quesada, "Maude: Specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.

[15] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "Swi-prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[16] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang, "An empirical study of programming language trends," *IEEE software*, vol. 22, no. 3, pp. 72–79, 2005.

[17] C. Hoare, "Hints on programming language design," tech. rep., STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1973.

[18] M. H. Van Emden and R. A. Kowalski, "The semantics of predicate logic as a programming language," *Journal of the ACM (JACM)*, vol. 23, no. 4, pp. 733–742, 1976.

[19] J. Backus, "Can programming be liberated from the von neumann style?: a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.

[20] B. W. Kernighan and D. M. Ritchie, *The C programming language, 2ed.* Prentice Hall, 1988.

[21] B. Stroustrup, *The C++ programming language.* Pearson Education, 2013.

[22] K. Arnold, J. Gosling, and D. Holmes, *The Java programming language.* Addison Wesley Professional, 2005.

[23] J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud, "Introducing OBJ," in *Software Engineering with OBJ*, pp. 3–167, Springer, 2000.

[24] J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright, *et al.*, "Abstract data types as initial algebras and the correctness of data representations," *Computer Graphics, Pattern Recognition and Data Structure*, pp. 89–93, 1975.

[25] L. Hamel, "The asteroid programming language on github." `https://github.com/lutzhamel/asteroid`, 2018.

[26] R. Ierusalimschy, L. H. De Figueiredo, and W. Celes Filho, "Lua-an extensible extension language," *Softw., Pract. Exper.*, vol. 26, no. 6, pp. 635–652, 1996.

[27] H. Rashidi, "A fast method for implementation of the property lists in programming languages," *International Journal of Programming Languages and Applications (IJPLA)*, vol. 3, no. 2, pp. 1–9, 2013.

[28] P. N. Kumar, "Impact of indentation in programming," *International Journal of Programming Languages and Applications (IJPLA)*, vol. 3, no. 4, pp. 21–30, 2013.

[29] J. V. Guttag and J. J. Horning, "The algebraic specification of abstract data types," *Acta informatica*, vol. 10, no. 1, pp. 27–52, 1978.