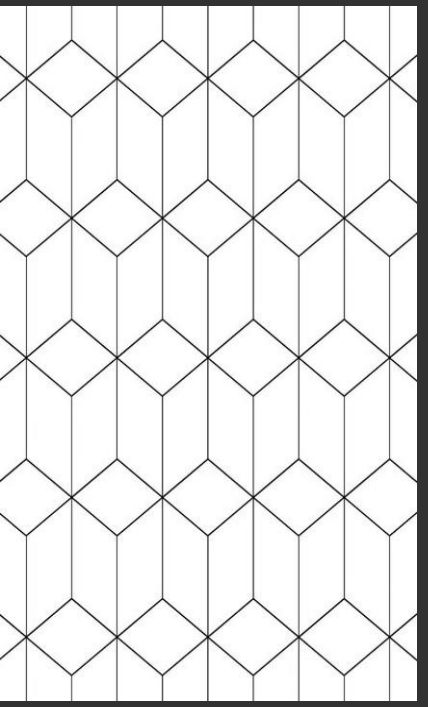
Detecting Overlapping Patterns in Asteroid, a programming language which supports both first-class and conditional pattern matching.

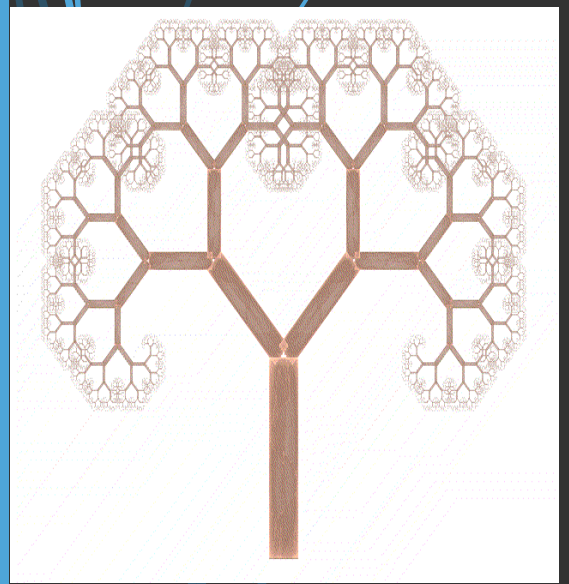
Timothy Colaneri, A.S.
Dept. of Computer Science and Statistics
University of Rhode Island
Kingston, Rhode Island, USA
tcolaneri@uri.edu

What is a Pattern?

- A pattern is a configuration of the elements of something, whether it be a human design or an abstract idea, that repeat in a predictable manner.
- In the context of programming language design, we have two main classes of patten matchings
  - Regular Expressions are patterns found in text consisting of characters.
  - Structural Patterns are patterns found in the structure or data, or how the data is formed.



What is Pattern Matching?



- Pattern Matching is the act of inspecting a series of given elements, or tokens, of something for the presence of a pattern.
- Pattern Matching, or Pattern Recognition, is one of the fundamental ways in which human beings make sense of the universe around them.
- In the context of programming languages, pattern matching is when we inspect input or processing data and make decisions based on the data's pattern.

The Useless Clause Problem

- A Useless, or Redundant, clause is one which can never logically be reached as its logic is inferred by a previous clause.

Take for example the following logic clause:

if x <= 0 **OR** if x > 0 **OR** if x > 10

When we evaluate this CNF formula, or multi-clause statement, we will never make it to the third clause, if x > 10.

Useless Clause's and Pattern Matching

- Useless Clauses are almost always the result of programmer error.
- While we can catch the example above by eye, it is not so intuitive when dealing with structural patterns and regular expressions.
- Many programming languages which support pattern-matching offer a built-in method of detecting this easy-to-make mistake.
- My project seeks to add this functionality to Asteroid, a programming language which supports two novel methods of pattern-matching, first-class and conditional pattern-matching.

Detecting Useless Clause's in the context of First-Class and Conditional Patterns

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

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 n:*NEG_INT do
        throw Error("factorial is not defined for "+n).
    end

println ("The factorial of 3 is: " + fact (3)).
```

First-Class Patterns

Asteroid elevates patterns to first-class citizens by viewing patterns as a value which can be assigned to a variable, passed to and returned from functions, etc.

Detecting redundant First-Class Patterns brings one significant problem to the table. They may not be defined until runtime!

```
function g
    with (x) %if x < 100 do
        return 2.
    orwith (x) %if x > 10 do
        return 3.
    orwith (x:%integer) do
        return 3.
    end.
```

Conditional Patterns

Conditional Patterns add an additional layer of decision structure to pattern matching. This functionality allows us to ask, or evaluate, a Boolean expression at the same time as the pattern matching expression.

Evaluation of redundancies in conditional patterns require us to check two different situations by which a clause may become redundant.

Thank You!

Links:

Asteroid Repository:

https://github.com/lutzhamel/asteroid

Special Thanks:

Professor Lutz Hamel

The Asteroid Programming Language.

Asteroid is a dynamically typed, multi-paradigm programming language that seeks to add more expressiveness to one of the core traits of functional programming languages, pattern matching.

Pattern matching is a simple yet powerful conditional programming construct in which we can make decisions based on the structure of data.

Asteroid adds two new methods of expression to pattern matching, first-class patterns and conditional patterns.

First-Class Patterns

In the design of programming languages, a first-class citizen is an entity in a given programming language which has full support and access to all of the standard operations and features of the language.

Asteroid elevates patterns to first-class citizens by viewing patterns as a value which can be assigned to a variable, passed to and returned from functions, etc.

Conditional Patterns

Conditional Patterns add an additional layer of decision structure to pattern matching. This functionality allows us to ask, or evaluate, a Boolean expression at the same time as the pattern matching expression.

This allows us to de-structure our input or processing data and then evaluate a Boolean expression before determining a match.

Examples: First-class and conditional patterns.

Below is an example of a function in Asteroid which utilizes **first-class conditional pattern expressions**.

This code shows two type-match conditional patterns that are declared on lines 4 and 5. These patterns are stored into variables to be later dereferenced and used in the fact() function definition.

This is a definition of a factorial function.

```
1   load "io".
2   load "util".
3
4   let POS_INT = pattern with (x:%integer) %if x > 0.
5   let NEG_INT = pattern with (x:%integer) %if x < 0.
6
7   function fact
8       with 0 do
9           return 1
10      orwith n:*POS_INT do
11          return n * fact (n-1).
12      orwith n:*NEG_INT do
13          throw Error("factorial is not defined for "+n).
14      end
15
16  println ("The factorial of 3 is: " + fact (3)).
```

Below is an example of a function in Asteroid which utilizes **conditional pattern expressions**.

This example shows that data can be de-structed and then have an expression evaluate the data's contents before recognizing a match.

This code shows the definition of the whichQuadrant() function. It determines the quadrant in which an (x,y) node exists.

```
1   load "io".
2   load "util".
3
4   function whichQuadrant
5       with (x,y) %if x > 0 and y > 0 do
6           return 1.
7       orwith (x,y) %if x < 0 and y > 0 do
8           return 2.
9       orwith (x,y) %if x < 0 and y < 0 do
10          return 3.
11      orwith (x,y) %if x > 0 and y < 0 do
12          return 4.
13      orwith x do
14          throw Error("whichQuadrant exepcts a 2-tuple of integers.").
15      end.
16
17  let x1 = tointeger(input("Enter an integer value for the x-coordinate: ")).
18  let y1 = tointeger(input("Enter an integer value for the y-coordinate: ")).
19
20  println("The node is located in quadrant: " + whichQuadrant(x1,y1)).
```

The Useless Clause Problem and First-Class Patterns

- When we evaluate a decision structure with First-Class patterns, we have to dereference the variable that the pattern is stored in before we can evaluate the pattern expression. This presents a complication; we may have no way of knowing what that variable is until the program is executing.
- All of the previous examples of redundant/useless pattern detectors ran during the syntactic, or parsing, phase. This is the most efficient as we would only execute the redundancy evaluation a single time.
- Evaluating redundancies during the semantic phase means we will evaluate each time the decision is made. This is expensive.

Standard ML
Error #69

[69] **redundant patterns in match**

In a multi-clause pattern match, if one of the later patterns can only match cases that are covered by earlier patterns, then the later pattern is redundant and can never be matched. In SML '97 it is an error to have useless (redundant) patterns.

```
4 handle Match => 5 | e => 6 | Bind => 7;
stdIn:1.1-20.15 Error: redundant patterns in match
          Match => ...
          e => ...
    -->   Bind => ...
```

Haskell
A warning

-Woverlapping-patterns

By default, the compiler will warn you if a set of patterns are overlapping, e.g.,

```
f :: String -> Int
f []     = 0
f (_:xs) = 1
f "2"    = 2
```

where the last pattern match in `f` won't ever be reached, as the second pattern overlaps it. More often than not, redundant patterns is a programmer mistake/error, so this option is enabled by default.

OCaml
A warning, #11

Warning numbers and letters which are out of the range of warnings that are currently defined are ignored. The warnings are as follows.

1 Suspicious-looking start-of-comment mark.

2 Suspicious-looking end-of-comment mark.

3 Deprecated feature.

4 Fragile pattern matching: matching that will remain complete even if additional constructors are added to one of the variant types matched.

5 Partially applied function: expression whose result has function type and is ignored.

6 Label omitted in function application.

7 Method overridden.

8 Partial match: missing cases in pattern-matching.

9 Missing fields in a record pattern.

10 Expression on the left-hand side of a sequence that doesn't have type `unit` (and that is not a function, see warning number 5).

11 Redundant case in a pattern matching (unused match case).

To the right we have an artificial example meant to demonstrate that a clause's redundancy cannot be determined until runtime.

```
1   load "io".
2   load "util".
3
4   function selectUserPattern
5       with (x) %if x >= 0 do
6           let ptrn = pattern with [head|tail].
7           return ptrn.
8       orwith (x) %if x < 0 do
9           let ptrn = pattern with [h1|h2|h3|h4|tail].
10          return ptrn.
11      end.
12
13  function f
14      with [] do
15          return 0.
16      orwith [h1|h2|h3|h4|h5|h6|tail] do
17          return 1.
18      orwith *USER_PATTERN do
19          return 2.
20      orwith [h1|h2|tail] do -- Useless
21          return 3.
22      end.
23
24  -- Get a value from the user
25  let i = tointeger(input("Enter an integer value: ")).
26
27  -- Set the user pattern
28  let USER_PATTERN = selectUserPattern( i ).
29
30  -- Evaluate the function with a test input.
31  println( "The output is: " + f([1,2,3]) ).
```

The function selectUserPattern() takes in a numerical value and then returns a pattern based on if the input was positive or negative. In this context we can observe that the actual patterns are either a head-tail pattern with a single head or four heads.

The function f() takes in a list. Its clauses determine which body of code should be executed depending on the size or structure of the list. The second clause attempts to extract the six leading nodes from a list and separate them from the remaining contents of the list. The final clause attempts to extract two leading node from a list and separate them.

The third clause is a first-class pattern. The actual pattern the we will receive when we deference this variable will depend on the input this program received from its user:

- If we received a positive value, the pattern will be a head-tail pattern where we try to pull out a single leading node. The presence of this pattern in this position will render the following clause redundant as any pattern that could have two leading nodes pulled out could have a single leading node pulled out. No list passed into this function will ever reach the final clause in this case.
- If we received a negative value, the pattern will be head-tail pattern that pulls out the first four leading nodes. All clauses are still reachable.

The Useless Clause Problem and Conditional Pattern Matching

- Conditional patterns add another layer to the pattern matching decision structure.
- As the new layer is itself another clause, this means we now have a new avenue by which a clause may be made redundant.
- To check for the presence of redundant patterns with conditional pattern matching, we will essentially have to evaluate the useless clause problem twice.

- There also exist situations in which redundancy cannot be evaluated as a complication of conditional pattern matching.
- Consider the case of a function call as the expression in a relational pattern clause. We have no method of determining what the function code is rendering redundant, as it is a collection of statements as opposed to patterns and relational expressions.
- Additionally, the conditional clause function code may alter values used by the function whose clauses we are currently evaluating. This would make evaluating a function while evaluating redundancy an unsafe operation.
- It may also be that case that a pattern may be made redundant by an enumeration or limitation defined outside the program.

Conditional Pattern Matching Cont.

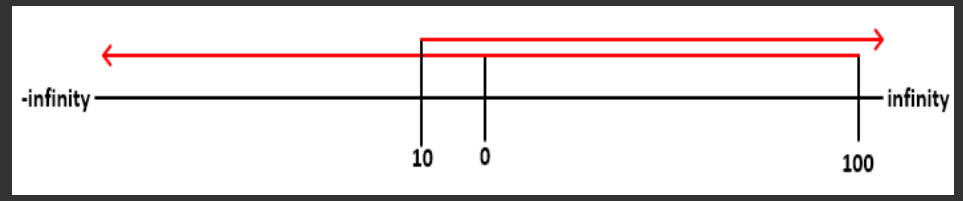
When we evaluate for redundancies in patterns, we only need to worry about a single clause rendering other clauses useless.

```
1   function g
2       with (x) %if x < 100 do
3           return 2.
4       orwith (x) %if x > 10 do
5           return 3.
6       orwith (x:%integer) do
7           return 3.
8       end.
```

However, in the case of conditional pattern matching, it may be the case that several preceding conditional relational statements, when combined, render a pattern useless. In order to keep track of when this situation occurs, we will need to keep track of subsumed intervals for a conditional pattern's (pattern, evaluated-variable(s)) identity,
Our method of detecting patterns made redundant through conditional pattern clauses inspects a number-line each time we are evaluating redundancy between two clauses to determine if the current clause will be reachable or not. Let's see what this might look like for the function to our right, g().

Step 1: We visit *with (x) %if x < 100 do* and add its range for its pattern/compared variables identity.

Step 2: We visit *with orwith (x) %if x > 10 do* and add its range for its pattern/compared variables identity.



Step 3: We visit *orwith (x:%integer) do* , as this is a type-match, a function is called that will evaluate the range associated with the pattern/variable key to determine if it has been completely covered/subsumed. This function will observe the number line shown above and then determine that both a 'to infinity' interval, and a 'to negative infinity' interval exists. The function then checks to see if the two values from these intervals overlap, which they do, and a redundant pattern is detected.

Conclusion

- There is no way to evaluate redundant first-class patterns at parse time. We can only evaluate the useless clause problem within the context of first-class patterns in the semantic phase.

- Evaluation of redundancies in conditional patterns require us to check two different situations by which a clause may become redundant.
- If the pattern of a clause can subsume a following clause's pattern(first situation), and the former or both are conditional patterns, then we have to evaluate relational subsumption between them(second situation). This can often be expressed on a number-line, although not all values may be numerical.