Robust Scripting via Patterns

Bard Bloom Martin Hirzel
bardb@us.ibm.com hirzel@us.ibm.com
IBM Research

Abstract

Dynamic typing in scripting languages is a two-edged sword. On the one hand, it can be more flexible and more concise than static typing. On the other hand, it can lead to less robust code. We argue that patterns can give scripts much of the robustness of static typing, without losing the flexibility and concision of dynamic typing. To make this case, we describe a rich pattern system in the dynamic language Thorn. Thorn patterns interact with its control constructs and scoping rules to support concise and robust test-and-extract idioms. Thorn patterns encompass an extensive set of features from ML-style patterns to regular expressions and beyond. And Thorn patterns can be first-class and support pattern-punning (mirror constructor syntax). Overall, this paper describes a powerful pattern system that makes scripting more robust.

Categories and Subject Descriptors D3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

Keywords Pattern matching, Thorn, dynamic typing

1. Introduction

An increasing number of programmers are writing quite serious code in scripting languages, rather than more traditional languages. Some of these programmers have the opinion (which might be true) that they know what they are doing and that compilers and type systems shouldn't get in their way. Others, perhaps more of them, have the opinion that this version of the software needs to get done next week, and the next revision needs to get done next month, and that they have to code as fast as possible. A third large sector has the opinion that they need to write client-side code on web browsers or similar systems, which generally means JavaScript or some-such.

Scripting languages give certain advantages that classically styled languages do not. Scripting languages are, ultimately, designed for quick coding. They are typically dynamically typed, so that programmers don't have to fuss with types, and can work with data structures that do not fit conveniently into any straightforward type system. They typically have concise syntax, allowing coders to write more code per keyboard-hour – which can be a significant consideration in the last hours before a deadline. They are often interpreted, eliminating compilation time and increasing portability.

The negative side of the tradeoff can be harsh. Scripts tend to be less reliable than traditional code. Dynamic typing means that many errors that would be detected at compile time in traditional languages remain unseen until runtime in scripts. Concise syntax, while nice to write, can make reading and maintaining code quite painful. The lack of static types hurts code maintenance as well. Interpreted languages are typically slower than compiled ones.

Part of the craft of designing a scripting language is to ameliorate the disadvantages without losing the advantages – just as for any language design.

DLS'12, October 22, 2012, Tucson, Arizona, USA.

1.1 Patterns vs. Types

This paper discusses the use of a rich pattern system in scripting languages. "Pattern" is intended more in the sense of ML, as a way to test, dissect, and destructure composite objects of arbitrary type, than in the sense of regular expressions on strings – though both senses are intended. Ultimately, a pattern does three things:

1. Decide whether a given *subject* has a certain structure;
2. Extract zero or more pieces;
3. Bind those pieces to variables in a certain context.

For example, the following code checks the value `L` to see if it is a three-element palindromic list whose first element is a string. If so, those elements are bound to the variables `x` and `y`, and concatenated. If not, an error value is returned.

```
fun f([x:string,y,$x]) = x+y;
  | f(_)               = "Wrong";
```

Patternless code for the same task might look like:

```
fun f(L) =
if (L instanceof List && L.length()==3
    && L.get(0) instanceof String
    && L.get(0).equals(L.get(2)))
  return L.get(0) + L.get(1);
else
  return "Wrong!";
```

So, patterns fit in well with the philosophy of scripting. Patterns don't introduce any new or unnecessary work. They only require writing things that needed to be written anyways, and they usually are more concise and sometimes more efficient than the alternatives. (For comparison, types do require labor that isn't immediately going to get the code working, and makes it longer.)

But patterns provide some of the information that types do. Clearly `f` above expects a list as an argument, and, indeed, a three-element list. While this can be seen from the patternless code as well, it is obscured. A smart interpreter, or any human reading the code, might be able to take advantage of this fact. It's certainly not going to recover all the power of a type system – the best way to do that is to have a type system – but it gives many of the benefits for a cost that script writers will find tolerable.

1.2 Patterns and Control Structures

If taken seriously, patterns work nicely in concert with standard control structures. We take the operation ~ for matching, so that `x~P` matches the subject `x` against the pattern `P`, returning **`true`** iff it succeeds, and introducing bindings into the region of code reachable only if it returns **`true`**.

```
if (x ~ [a, b]) use(a, b);
else complain();
```

checks to see if `x` is a two-element list. If so, it uses its elements, from the variables `a` and `b` scoped to the then-clause. If not, it complains; `a` and `b` are out of scope and cannot be used.

Similarly, a pattern match in the test of a **`while`**-loop

```
while (x ~ [a, b])
  x := use(a, b);
```

introduces bindings in the body of the loop.

Nearly every standard control structure can benefit from pattern matching. Once you have patterns in your language, they become pervasive and very convenient.

1.3 Advanced Forms of Patterns

The pattern language familiar from ML [15], Hope [2], et al. includes a number of useful forms. Atomic patterns include variables which match and bind anything, wildcards (_) which match anything but bind nothing, and literals which match one specific thing and bind nothing. Composites include matchers for built-in and user-defined algebraic data types (or classes, as appropriate), often with special forms for extracting the head and tail of a list even if lists aren't algebraic.

Thorn adds several more pattern operators, sketched here and described in detail in Section 5.2. Type test patterns, such as `x:int`, can emulate typed function parameters. Expression tests, such as `(it % 2 == 0)?`, where `it` is bound to the subject of the pattern, make it possible to embed arbitrary predicates in patterns. List patterns can contain ellipses (`...`) to check multiple list elements against a pattern, so `[_:int...]` is a list of integers. Multiple ellipses in a list pattern are supported by back-tracking. Patterns can contain internal pattern matches of the form `E~P`. Patterns can contain conjunctions, disjunctions, and negations, enabling boolean combinations of tests and extractions. Finally, Thorn's pattern language integrates regular-expression matching on strings. For example, `"([0-9]+):([0-9]+)" / [hour, min]` extracts information from strings such as `"9:35"`.

1.4 First-Class Patterns

For an entity to be a *first-class* citizen of a programming language, it needs to satisfy three conditions. It must be possible to store it in a variable or pass it as a parameter to a function. It must be possible to use that variable or parameter where the entity would be expected. And it must be possible to create an anonymous instance of the entity at the place where it is used. For instance, first-class functions can reside in variables or parameters; can be called via variables or parameters; and can be created anonymously in place, e.g., using the **`fn`** keyword in Thorn.

Thorn supports first-class patterns: patterns can reside in variables or parameters; can be matched via variables or parameters; and can be created anonymously in place, using the **`pat`** keyword. One use-case for first-class patterns is to make large complex patterns easier to read, by defining sub-patterns separately, then using them from the main pattern. Another use-case for first-class patterns is to reduce the dependency between pattern-matching code on the one hand, and the matched data structures on the other hand. If the pattern-matching code uses a first-class pattern from a parameter, we can switch data structures and the code still works, as long as we supply the appropriate pattern.

We define *pun* to be the intentional syntactic similarity between term construction and term matching. Besides being aesthetically pleasing, puns make code easier to read. Thorn's first-class patterns support puns, since a pattern for matching a data structure can be defined in the same breath with a constructor for it.

An *input parameter* to a pattern is a value that modifies its matching behavior. For example, a pattern for splitting a string can have a parameter for the divider character. Thorn's first-class patterns can take input parameters.

1.5 Contributions

No feature of Thorn is completely without precedent, though first-class patterns and internal pattern matches are rarely seen. However, Thorn makes an unusually pervasive use of patterns, and has an unusually complete set of patterns.

The core argument of this paper is that patterns can provide a dynamic language with benefits comparable to those that static types provide in a static language. What both patterns and static types have in common is that they declaratively specify the expected properties of data. Thus, patterns can make scripts more robust, without giving up the conciseness and flexibility of dynamic typing. In support of our argument, we implemented a rich pattern system in our dynamic language Thorn. This includes first-class patterns, which enable powerful abstraction over patterns. All in all, this paper makes two main contributions: one concrete (an innovative pattern system) and one philosophical (a new perspective on patterns).

2. Related Work

ML-style patterns. A core feature of ML is pattern matching via terms over algebraic data types [15]. In the context of statically typed functional languages, pattern matching is usually synonymous with ML-style patterns. Proposed in ML's predecessor ISWIM [13], ML-style patterns are also central to Hope [2], Haskell [9], and F# [19]. When statically typed languages in other paradigms borrow features from functional languages, ML-style patterns are prime candidates, as in Scala [3] and Kotlin [12]. Thorn embraces ML-style patterns, but they are only part of Thorn's rich pattern system.

Scripting via patterns. In the context of scripting languages, pattern matching is usually synonymous with regular expressions (REs). This contributes to the success of Perl [22], and Ruby [5] where REs are tightly integrated with the rest of the language. Even when patterns in scripting languages are not exactly regular expressions, they are used for string matching, as in SNOBOL [7] or Lua [10]. Thorn embraces string pattern matching via REs, but they are only part of Thorn's rich pattern system.

Dynamic ML-style patterns. Scheme provides ML-style patterns as a library [24]. Geller et al. describe how to add ML-style patterns to the dynamic language Newspeak [6]. Other dynamic languages provide some form of patterns, including destructuring assignments in Python [18], rules for matching UML in Converge [20], and full-fledged term matching grammars in OMeta [23]. In comparison to these pieces of prior work, Thorn patterns are more closely knit into the fabric of the language, making their use ubiquitous and maximizing the advantages of patterns for robust scripting.

Scoping of bindings from patterns. Bindings from Thorn patterns are available in code reachable when the pattern succeeds. This is a generalization of ideas from other languages for how patterns can interact with the rest of the language. Specifically, in OCaml (and Thorn), the left-hand-side of the binding construct is a pattern. In ML-style languages (and Thorn), patterns produce bindings for function formals [15]. In JMatch (and Thorn), patterns produce bindings in if-statements and loops [14]. In Perl and Ruby, groups captured in an RE are available in special variables; in Thorn, they are available in normal variables. In Python (and Thorn), patterns interact with comprehensions [18].

Rich patterns. Besides the pattern-external features from the previous paragraph, Thorn also has an unusually large set of pattern-internal features. It improves over string-matching REs and over ML-style patterns by providing both, and then some. Prior work on rich pattern matching systems includes Views, which let a type be pattern-matched in ways it was not originally designed for [21].

Ernst et al. generalize ML-style patterns with predicates and support for objects [4]. Another rich pattern system is Tom, a preprocessor for adding pattern matching to C, Eiffel, and Java [16]. At the high end of the expressiveness spectrum is OMeta, which uses parsing expression grammars (PEGs) for matching [23]. Our own prior work on Matchete pioneered bringing together several flavors of patterns in a Java extension [8]. Our own previous paper on Thorn focused on other features, and only briefly mentioned Thorn's pattern system [1]. Furthermore, the current study includes several new insights and pattern features that came afterwards.

First-class patterns. Thorn supports first-class patterns. The active patterns of F# provide much of the same functionality, albeit in a static language [19]. Newspeak has first-class patterns as well [6]. Perl REs can be interpolated into each other at match time, making them first-class in a way [22]. And Scala's extractors also provide much of the power of first-class patterns, again statically [3]. On the theoretical side, Jay and Kesner studied first-class patterns in a calculus [11]. Thorn differs from these pieces of prior work in that it uses first-class patterns in a more expressive pattern language.

3. Philosophy and Practice of Patterns

A pattern match does two things: it checks to see if a certain value, the *subject*, has a certain structure or appearance; and it *yields* (or *produces* or *binds*) zero or more results. The simplest pattern is thus the wildcard, _ in Thorn, which matches anything and yields nothing:

```
if (x ~ _) println("Yes!");
```

Patterns start to get more useful when *variables* are introduced. A variable matches anything, but, unlike a wildcard, it binds what it matched.

```
if (3 ~ x) println("x = " + x + " = 3");
```

Patterns become nontrivial when they start *destructuring* values: checking that composite values have a particular structure, and, if they do, extracting pieces of that structure:

```
L = [1,2];
if (L ~ [x,y]) println(x + "," + y);
```

3.1 Encapsulating Programming Idioms with Patterns

Most programming languages have the two halves of pattern matching: tests (tests for a given structure can usually be programmed, even if they are not available as primitives), and binding (or some form of giving names to values). The power of pattern matching comes from combining the two into a single construct, without, in general, the linguistic and allocation overhead of returning a composite value.

Combining them enhances expressiveness. For example, the conventional Java pattern for accessing a hash table carefully is:

```
// Java
if (table.containsKey(key)) {
  Value value = table.get(key);
  use(value);
}
```

The equivalent pattern code in Thorn is:

```
if (table.get(key) ~ +value)
   use(value);
```

(The + pattern means roughly, *"it's there"*; see Section 5.2.4.) Retrieving elements is the most important operation on maps. Java maps split it between two method calls: Java's method abstraction does not fit this situation very well. Pattern-matching puts it back into a single method call.

Combining them can potentially enhance performance as well. The Java code (*pace* an extraordinary compiler) needs to compute the hash code of `key` and look it up in the table twice: once to tell if it's there via `containsKey`, and again to actually return it. Any halfway-decent pattern matching code will only need to do one hash computation and table lookup; it combines the test and the return of the value into a single operation.

Advantage	Static	Pattern
Describe shape of data	somewhat	somewhat
Static error checking	yes	occasional
Optimization	yes	maybe a little
Refactoring	yes	somewhat!
Code Assistance	yes	no
Conciseness	no	yes
Sloppy Data	no	yes
Open-ended Descriptions	no	yes

Table 1. Advantages of static typing and Thorn-style patterns.

3.2 Pattern Punning

One design principle of Thorn, and many other languages with patterns, we call *pattern punning*: that patterns look like the associated expressions. As an expression, `[11,22,33]` constructs a three-element list with elements 11, 22, and 33. As a pattern, it matches a three-element list with the same elements.

One flaw associated with pattern punning is that it is often hard to tell what is a pattern and what is an expression. (If we avoided pattern punning and its associated ambiguity, by having disjoint pattern and expression languages, the language would be unambiguous but unendurable.)

A second flaw is that Thorn programmers may rejoice in writing concise, intricate, and perhaps confounding patterns. This is an issue with human nature, and, on the whole, outside of the scope of this paper to address.

3.3 Comparing Patterns to Types

When one chooses to work in a dynamically typed language, one eliminates a heap of annoyances, from the obligation to write types to the difficulty of writing generic libraries. One also eliminates a heap of *advantages*, from catching many errors statically to having a clue what one's data looks like when one is reading or writing code. One might to wish to recover some of the lost advantages, preferably without recovering many annoyances as well.

One should not expect to recover *all* the lost advantages. The best way to have the advantages of static typing is to use static typing. The goal, then, is to recover as many of the advantages of static typing as possible – and perhaps some different advantages – with as little overhead as possible.

Table 1 summarizes some of the advantages that one might wish to have, both those which come from static types, and those which come from patterns. The rest of this section has a little more detail. In every case many books could be written on caveats, exceptions, variations, and alternatives.

Statically describe shape of data. Static types explain some facets of data structures: *e.g.*, in Java, that `x` is an array of integers. They leave out crucial features, such as the size of the array. Thorn's patterns can test for both of these. It is not uncommon to statically know, in a region of code, with no more syntactic overhead than types require (albeit with more runtime overhead), that `x` is a list of three integers:

```
fun f(x && [_:int,_:int,_:int]) = ···
fun g(x && [_:int...] && (x.len == 3)?) = ···
```

Similarly, ***static error checking*** is in principle possible with patterns. Uses of `x` in ways incompatible with a list of three `int`s could be (but are not currently) detected at compile time.

Optimization. Since the earliest days of FORTRAN, optimizing compilers have made heavy use of static type information. Patterns

sometimes provide information that could be of use to optimization, such as array sizes. Perhaps an optimizing compiler (were one to be written) for Thorn could exploit this information.

Refactoring. One of the joys of using a powerful programming environment for Java is to be able to instantly rename a method from `doit` to `sort_input_data`, and have all references to the method change, but no unrelated `doit`s . This is generally impossible without static types, and patterns do nothing to help. However, see Section 7.5 for an example of a rather more difficult refactoring – turning a list of numbers into a `Point` class – which is more easily done with Thorn patterns than with static types.

Code Assistance. Another joy of static types and powerful programming environments is the ability to press a key and get a listing of all methods defined on some object. Doing this robustly in a dynamic language would be extraordinarily challenging.

Conciseness. Patterns excel at checking the shape of data, and simultaneously extracting pieces from it. Static types separate these operations. For example, to add up the `x`, `y`, and `z` fields of a record, a static type system would need to extract each field separately:

```
// Java
int sumxyz(ThingXYZ a) { return a.x + a.y + a.z; }
// Thorn
fun sumxyz(<x,y,z>) = x+y+z;
```

Sloppy Data. In some applications, especially on the web, a program should expect that its input data is only approximately properly structured. Dynamic types are far more adept at manipulating sloppy data than most static types, especially such harmless variations as records with extra fields.

Open-Ended Descriptions. Pattern programming is all about defining and using new patterns that apply to existing objects and data. Few static type systems allow the definition of new types that apply to existing data, much less use them effectively.

4. Around Patterns: Control and Scope

Name	Syntax / example
Binding statement	`P = E;`
If statement	`if (E ~ P) S1 else S2`
`&&` expression	`(E1 ~ P1) && E2`
For statement	`for (P<-E) S` or `for (P<~E) S`
Comprehension	`%[ E1 | for P<-E2 ]`
While and until loops	`while (E ~ P) S` etc.
Match statement	`match(E){P1=>E1 | ··· |Pn=>En}`
Function definition	`fun ID(P1)=S1 | ··· | ID(Pn)=Sn`
Anonymous function	`fn P1=S1 | ··· | Pn=Sn`

Table 2. Overview of Thorn features that interact with patterns, where `P` is a pattern, `E` is an expression, `S` is a statement, and `ID` is an identifier.

Thorn has a powerful and well-integrated pattern language. It has a potent collection of pattern primitives, and, perhaps more interestingly, a convenient set of ways to *use* patterns (Table 2).

4.1 Binding statement

A central design principle of Thorn (and many other pattern-rich languages) is that patterns should be allowed wherever variables are bound. This is particularly clear in the binding statement itself:

```
x = 1;
```

introduces a new variable `x` and binds it to the value `1` for the rest of the current scope. Some languages have a keyword, often **`let`** or **`val`**, or a type name, for this concept. Thorn encourages immutable binding by giving it the shortest syntax. (Mutable variables are introduced as **`var`** `m := 1;` and updated with `m := 2;` — not painful, but it's clear which one is favored.)

The left-hand side of a binding = can be an arbitrary pattern. For example,

```
[x,y,z] = L;
```

checks that `L` is a 3-element list. If so, it binds the first element to `x`, etc. If not, it throws an exception.

4.2 `if` statement and `&&` expression

Thorn, like most pattern-rich languages, has the expression `E~P`, which returns a Boolean result of whether a value matches a pattern. In many languages, using matching expressions loses the bindings that the match might induce; one must use a `match` construct to capture the bindings.

By contrast, Thorn takes as a design principle that a match should introduce bindings into the region of code that will be executed only if the match succeeds. For example, a match appearing in the test of an **`if`** introduces bindings into the then-clause. Similarly, in a conjunction `(E~P) && B`, variables bound in `P` are available in `B`.

In this example, we check that `L` is a one-element list `[x]` and `M(x)` a two-element list, and use the elements.

```
if (L ~ [x] && M(x) ~ [y,z]) {
  use(x,y,z)
} else {
  println("x,y,z not in scope here.");
}
```

In this example, `x` is bound in `L~[x]`, and thus can be used as an argument to the function call `M(x)` in the next conjunct, as well as the then-clause. Similarly, `y` and `z` are in scope in the then-clause. None of these is available in the **`else`**-clause.

Note that writing this with **`match`** statements would require two matches, and duplication of the **`else`**-clause.

4.3 `for` statement

`for` loops, like other binding contexts, allows patterns. For example, iterating over a list of three-element lists can be done by:

```
for ([x,y,z] <- L)
  use(x,y,z);
```

As always in pattern matching, one must treat the case of failure in some sensible way. There are two reasonable choices: (1) strict looping, which requires all elements of the list to match the pattern, and throws an exception if one does not, or (2) lax looping, which only executes the body for those elements that match the pattern. We had lots of two-symbol combinations lying around unused, so we took both choices. A loop using <- is strict; a loop using <~ is lax.

Quite often, script programmers are under the impression that they have a data structure with a uniform structure: say, a list of records with pairs `[a,b]`, say, coming from a user input file. In many cases, the programmers are incorrect, perhaps due to users who have filled in their input file haphazardly. A strict loop over the data structure is a simple way to state and check a desired invariant. It will throw an exception if the data structure is not precisely as the programmer expected, which is the best that can be done in an untyped language.

```
for ([a,b] <- listOfABs)
  use(a,b);
```

Equally often, programmers want to search through a list for suitable values, say pairs `[a,b]`, and do something to them. This could be written as:

```
for (x <- listOfWhatever)
  if (x ~ [a,b])
    use(a,b);
```

But the lax match is shorter and arguably clearer:

```
for ([a,b] <~ listOfWhatever)
  use(a,b);
```

4.4 Comprehension

Thorn has a rich set of comprehensions. For example, the list of squares of elements of `L` can be written succinctly as:

```
%[n*n | for n <- L]
```

Comprehensions thrive on conciseness. The list of squares of integers in `L` could be written as follows where `:?` is Thorn's syntax for a type test:

```
%[n*n | for n <- L, if n :? int]
```

But comprehensions using lax matching are often cleaner:

```
%[n*n | for n:int <~ L]
```

4.5 `while` and `until` loops

In the basic while loop `while(E) S`, the only way that `S` can be executed is if the expression `E` is true. So, variables bound by `E` are available in `S`. For example,

```
var L := something();
while (L ~ [x,y,z]) {
  L := somethingElse(x,y,z);
}
```

The `do S while(E)` loop, disappointingly, doesn't cooperate with matching at all. On the first iteration, `S` is executed without testing `E`, so matches in `E` can't get into `S`. There is no region of code that is only executed if `E` is `true`, and thus, no place that pattern bindings can be used. (Bindings in `S` could in principle be propagated to after the loop, at the unacceptable cost of letting bindings escape from braced statements.)

One of the Thorn implementers added `until` loops on a lark. We had never thought that `until(E)` could be anything but an abbreviation for `while(!E)`, saving an all-important character or three in those all-important cases where the loop test needed to be negated. We didn't remove it from Thorn, largely because the half-hour it would take to do so could better be spent by making fun of it.

We mocked it too soon. `until`'s binding behavior is quite unlike `while`'s, in a very useful way. Consider an `until`-loop and the following statement: `until(E){S}; T`. Inside the body `S`, `E` has been evaluated to `false`, so there are no bindings in `S` from `E`. However, if control ever passes to `T`, then `E` must have evaluated to `true`.

That is, the test in an `until` loop introduces bindings *after the loop*. This behavior is precisely what is desired when one is searching for something that can be described by a pattern `desired[y]` which binds a variable `y` – and, as we shall see, any predicate can be phrased as a pattern. The general idiom is:

```
var x := init;
until (x ~ desired[y]) x := next(x);
weFound(y);
```

`do S until(E)`, unlike its dual `do S while(E)`, introduces bindings from `E` into the following code. This is natural for searching when at least one attempt must be made, *e.g.*, requesting the name of an existing file from a user.

4.6 `match` statement

Thorn has a statement devoted to pattern matching, a chimera of the C-style `switch` statement and the ML-derived notation for function definition. It consists of a subject expression, and a sequence of pairs of a pattern and a code block. Dynamically, the subject's value is matched against each of the patterns in turn; when the first one succeeds, its code body is evaluated and its value returned. It is an error for all to fail. A wildcard pattern _ is often used as a catch-all for the last clause.

```
match (thing) {
    [x]   => {println("Singleton $x");}
  | x:int => {println("An integer");}
  | _     => {println("Something else");}  }
```

4.7 Function definition

Function definitions, method definitions, and closure expressions allow pattern matching in the argument list. In the simplest case,

```
fun two(x) = [x,x];
```

uses a variable pattern, which matches anything and binds it to `x`. Functions whose arguments have to be some particular type quite naturally use a type pattern. `x:int` matches an integer and binds it to `x`, and so the squaring function on integers can be defined:

```
fun square(x:int) = x*x;
```

The full power of patterns can be used in function definitions. For example, a function taking a list of two numbers `[a,b]` with `a<b` can be defined as:

```
fun interval([a,b] && (a<b)?) = "good";
  | interval(_)               = "bad";
```

(When one defines a function with nontrivial patterns as arguments, it is often wise to give a catch-all case, perhaps throwing a well-tuned exception precisely explaining what went wrong, or perhaps, as in this case, just whining.)

5. Inside Patterns: Rich Sub-Patterns

Name	Syntax / example
Familiar patterns	
Variable	`V`
Wildcard	`_`
Literal	`"hi"`
Interpolate	`$V` or `$(E)`
List, fixed-length	`[P, Q]`
List, by head and tail	`[P, V...]`
Record	`<F=P, G=Q>`
Type constructor	`T(P, Q)`
Fancy patterns	
Type test	`V : T`
Expression test	`(E)?`
List, general	`[P..., Q, R...]`
Positivity test	`+P`
Internal pattern match	`E ~ P`
Boolean combination	`P && Q` or `P || Q` or `!P`
Regular expression	`RE / P`

Table 3. Overview of Thorn patterns, where `V` is a variable, `E` is an expression, `P`, `Q`, and `R` are patterns, `T` is a type, `F` and `G` are fields, and `RE` is a string with a regular expression.

Table 3 lists the various patterns that Thorn supports. The following subsections describe them in detail.

5.1 Familiar Patterns

Nearly every language of extraction patterns has a core of patterns, corresponding to the extractors for the data types of the language. One can program happily for years using just the core constructs, as ML programmers do. We list these for the sake of completeness, and for the sake of a few minor language design points that apply to them; but on the whole they are standard.

Variable. A variable, `V`, matches anything, and binds `V` to the value that was matched.

Wildcard. A wildcard, _, matches anything and doesn't bind anything. This can be used to ignore irrelevant parts of a data structure.

Literal. A literal, a constant expression, matches the value of that expression, and binds nothing. For example, the pattern `"hi"` matches the string `"hi"`.

Interpolate. The pattern `$V`, where `V` is a variable, or `$(E)`, where `E` is an arbitrary expression, matches the value of `V` or `E` and nothing else. So `$(1)` is equivalent to `1`, and if `x=1`, then `$x` is also equivalent. The ability to compute values avoids an awkward feature of some pattern systems, such as Prolog and Scala, which rely on capitalization to distinguish whether an identifier is a constant or a pattern variable. .

Fixed-length list. The pattern `[P,Q]` matches a two-element list whose first element matches `P` and whose second element matches `Q`.

List by head and tail. The pattern `[P, V...]` matches a list whose first element matches the pattern `P`. The remainder of the elements, of which there may be zero or more, are bound to the variable `V`. This is an instance of a more intricate form for matching on lists in Section 5.2. For better pattern punning, Thorn allows `[E, F...]` as an expression as well, consing the value of `E` onto the head of the list `F`. For example, the function to apply `f` to every element of a list `L` can be written:

```
fun map1(f, [x, y...])) = [f(x), map1(f, y)...];
  | map1(f, [])         = [];
```

Record. Thorn records are labelled, unordered tuples. Records are constructed with the syntax `<x=1,y=2>`. Record patterns have the form `<F=P,G=Q>`, which matches a record with fields `F` and `G` (and possibly others), where the `F` field matches `P` and the `G` field matches `Q`. Arbitrary patterns are allowed in the fields, so

```
if (pair ~ <x=[_,x1,_],y=y1>) use(x1,y1);
```

extracts the second element from a list in the `x` field to use.

As a convenience, a field name alone, like `x`, is interpreted as `x=x`. So,

```
if (pair ~ <x,y>) use(x,y);
```

extracts the `x` and `y` fields from `P` into the variables `x` and `y`, and uses them.

The same syntax, in Thorn, can match objects and look at their nullary methods. If `o` is an *object* with a nullary method `x`, then `o~<x=xx>` will succeed and bind `xx` to the value returned by the method call `o.x`.[1]

Thorn has no simple pattern syntax for describing a record that has an `x` and a `y` field and *no* others. (The pattern Boolean operators (Section 5.2.6) can be used to express that it has an `x` and a `y` but no `z`, as `<x,y> && !<z>`.) This is an intentional philosophical point. Such a pattern would amount to a test passed by a instance of a class, but not by a instance of any subclass: a test that most object-oriented languages either exclude, or allow via some dirty-but-necessary library functions but certainly reject as a language construct. Indeed, an exact field test can be performed by dirty but necessary library functions in Thorn as well. (Similarly, we would abandon this philosophical point in an instant given a persuasive use case for exact matching, but for now the glint of purity is amusing.)

Type constructor. Thorn is a class-based language. Every class has a default constructor, given with arguments to the **`class`** declaration: **`class`** `Point(x,y)`$\cdots$. Class constructors have the syntax `Point(1,2)`. Objects are destructured by a similar syntax: **`if`**`(P~Point(x,y))use(x,y);`

Classes can have constructors in addition to the default constructor. The non-default constructors are not automatically deconstructable. First-class pattern members can be used to provide deconstructors for non-default constructors.

[1] Thorn does not require `()` on nullary methods, and does not expose instance variables, so the `o.x` syntax unambiguously refers to a method invocation on an object `o`. The choice that **all** nullary methods appear to be fields is debatable, and often-debated.

5.1.1 Variable Availability

Patterns are matched from left to right. Variables bound early in the pattern may be used later in the pattern. For example, `[x, $x]` matches a two-element list with equal elements: the first element is bound to `x`, and then its value is used in the subpattern `$x` which matches a subject equal to the value of `x`.

5.2 Fancy Patterns

At approximately this point, a pattern-rich language can either stop or proceed. Stopping gives a useful but constrained pattern language. In a typed language, the constraints are useful, allowing efficient compilation of patterns *a la* ML. In a dynamic language, such as Thorn, no such compilation is possible. There is no reason not to give patterns any capability we can imagine.

So Thorn's first-order pattern language, while hardly unprecedented, is unusually generous. There are minor features like type tests, specialized but very useful features like list and regexp patterns, and broadly general features that evaluate expressions and perform boolean combinations. The individual features are of only moderate interest by themselves; their use in combination is what counts.

5.2.1 Type test

The pattern `x:`**`int`** matches any integer, and binds `x` to its value. More generally, `P:T` fails if its subject is not of type `T`; if the subject is of type `T`, it matches it against `P` and produces whatever bindings `P` does. This is frequently used to give the appearance of typed function arguments:

```
fun sumsq(x:int, y:int) = x*x+y*y;
```

The most common choice of pattern `P` other than a variable is a wildcard: **`_:int`** is a pattern that matches any integer.

5.2.2 Expression test

The pattern `(E)?` evaluates the expression `E`. If its value is **`true`**, the pattern match succeeds without binding any values. If it is **`false`**, the match fails. If it is anything else, it raises an exception.

Inside of `E`, the variable `it` represents the subject of the match. For example, `(it > 0)?` is a pattern that matches positive numbers.

Test patterns alone don't do very much. In conjunction with the pattern `&&` operator, or other operations, they provide side conditions and considerably more.

With great power comes great uncomputability. The familiar patterns of Section 5.1 can in many cases be compiled quite efficiently. The compilation schemes of ML, Scala, and so on sometimes take advantage of this, building finite-state automata to check objects against patterns. Such compilation schemes are harder or impossible in the presence of test patterns — which is, perhaps, why side conditions are generally relegated to the side in many languages. Given that we wanted Thorn to have a rich set of pattern constructs, and given that the language is dynamically typed, tight compilation schemes like ML's are out of the question anyways, so we put our tests into patterns for maximum expressiveness.

5.2.3 General list pattern

Thorn supports pattern matching over lists. In a list pattern, `x...` matches an arbitrary sublist, binding it to `x`. So,

```
[x..., true, y...]
```

matches a list containing **`true`** as an element somewhere in it. `x` is bound to the list of elements before the first **`true`**, and `y` to the list of elements after the first **`true`**. Similarly, `_...` matches but does not bind a sublist.

There may be any number of `x...` elements in a list pattern, and they will all be explored, in a straightforward backtracking way. The pattern match may not be the most efficient in all cases. It is $O(n^{k-1})$ where n is the length of the list and k is the number of `v...` subpatterns, and, for **`true`**-finding, a linear algorithm is

possible. But this pattern is certainly easy to write, and may be adequate for many situations.

As a further elaboration, a pattern component may have the form `x && P ...`, where `x` is a variable and `P` is a pattern. This form matches a sequence of elements each of which match `P`, and binds them to `x`. For example, to match a list consisting of some `1`'s followed by some `2`'s, use:

```
if (L ~ [ ones && 1 ..., twos && 2 ... ])
   use(ones,twos);
```

5.2.4 Conditional Return, Positive Tests, `+P`

Quite commonly, a subprocedure can *conditionally return*: either succeed and produce some values, or fail. For example, popping a stack can succeed and yield the prior top of the stack, or fail if the stack was empty. Thorn's idiom for conditional return takes advantage of Thorn's dynamic typing. It uses a unary 1-1 operation `+`, with the connotation that a function returning `+(v)` is saying that it is successfully returning the value `v`. It returns a distinguished value **`null`** to indicate failure; of course, `+(v)` must never be **`null`**. For example, the function to find the position of `v` in list `L` appears in Figure 1. Note that the middle clause of the auxiliary function `loop` returns `+n` rather than simply `n`.

```
fun index(v, L) {
   fun loop([], n) = null;
     | loop([$v, _...], n) = +n;
     | loop([_, tail...], n) = loop(tail,n+1);
   loop(L,0);
}
if (index(2, [0,1,2,3]) ~ +n) assert (n==2);
```

Figure 1. Computing Index in Thorn

To use the result of a conditionally returning function, match it against the pattern `+P`. If the subject is not in the range of `+(·)`, in particular if it is **`null`**, the match fails. Otherwise, the subject must be `+v` for some value v; the matcher matches v against pattern `P`, succeeding if `P` does, and binding whatever `P` binds.

In particular, the pattern `+x` succeeds if the subject is `+(v)`, and binds `v` to `x`, and fails if the subject is **`null`**. So the idiom **`if`** `(f(a,b)~+x) use(x);` expresses conditional return. This appears in the last line of Figure 1.

This will work for any function `+(·)` that is 1-1 but not onto. Scala uses the `Option` type for this purpose, incurring an object allocation each time.

Thorn, which is dynamically typed, can be more economical. `+(·)` is already defined, from FORTRAN days, as the identity function on numbers: `+2` evaluates to `2`. We generalize this so that it is the identity function on nearly all values.

`+(`**`null`**`)`, of course, can't be **`null`**, as **`null`** is outside the range of `+(·)` by hypothesis. In Thorn, it is a value with no other interesting behavior or structure, beyond its connotation of a function which successfully conditionally returns the value **`null`**. `+(+`**`null`**`)` is a still different value, and, in general, the values $+^n$`(`**`null`**`)` for $n \geq 0$ are all distinct. These are called the *nullities*. Except for **`null`** and `+(`**`null`**`)` they rarely appear in code.

And the nullities are the *only* values for which $+(v) \neq v$. This makes computing and inverting `+(·)` very cheap: usually nothing needs to be done, and, with a bit of caching, the nullities rarely need allocation.

5.2.5 Internal Pattern Matching: $E \sim P$ as a Pattern

Thorn pattern matches can contain pattern matches. The *pattern* `E~P` behaves like the *expression* `E~P`: it evaluates `E`, and matches the result against `P`, introducing bindings from `P` if it succeeds. Like the test expression `(E)?`, internal pattern matches bind the subject of the pattern to the variable `it`.

Internal pattern matches are particularly useful for sneaking getters and other method calls into the middle of patterns. For example, consider a function that accepts a list of maps, each of which has an integer at key `"a"`. The pattern `it.get("a")~+_:int` is one way to test whether a map has a number at key `"a"`. So, the following pattern tests for a list of such maps:

```
[ (it.get("a") ~ +_:int) ... ]
```

Internal pattern matching can almost be imitated by test patterns. `(E~P)?` uses the pattern matching *expression*; the test pattern construct `(···)?` doesn't bind any values. The internal pattern match `E~P` succeeds or fails under the same circumstances, but binds the values bound by `P`.

As an extreme and surprisingly useful case, an internal pattern match can be used to simply bind a value. `1~x` always succeeds, and binds `x` to `1`. This appears in complex patterns, especially first-class ones.

5.2.6 Boolean Combinations of Patterns

Patterns can be combined with the short-circuiting Boolean combinators `&&`, `||`, and `!`. The success or failure of a Boolean pattern is based on the success or failure of the subpatterns: `s` matches `P && Q` iff it matches both `P` and `Q`; it matches `P || Q` if it matches `P` or it matches `Q`, and it matches `!P` iff it does not match `P`.

Pattern matching does more than return a success or failure; in the event of success, it binds variables. Thorn's rules of what is bound are fairly generous. `P && Q` binds everything that either `P` or `Q` binds; furthermore, bindings from `P` are available in `Q`. `P||Q` binds everything that *both* `P` and `Q` bind. And `!P` binds nothing at all.

`&&` is far and away the most useful Boolean combination of patterns. It provides the main pattern construct that we have not mentioned, often appearing as **`as`** or `@`. In languages that have it, `x` **`as`** `[y,z]` is a pattern that matches a two-element list like `[y,z]`; if the match succeeds, `x` is bound to the whole list and `y` and `z` to the components. In Thorn, this is expressed as:

```
x && [y,z]
```

Pattern `m && (test(it))?` matches a value `m` which answers true to `test`. Since variable binding is done from left to right, `m && (test(m))?` does just the same, but with the occasionally obscure variable `it` given the same name `m` that it has in the rest of the program.

`&&` has uses beyond **`as`**, though. For example, one may match a list containing `1`, `2`, and `3` in any order with:

```
[_...,1,_...] && [_...,2,_...] && [_...,3,_...]
```

`&&` and the test-pattern `(E)?` described above allow side conditions to be taken off the side, and put inside patterns — often in the places where they can do the most good. For example, the function `hype` in Figure 2 takes a function `f`, and a nonzero value `x` such that `f(x)` and `f(1/x)` are both singleton lists. It is important to check that `x!=0` before computing the reciprocal, which would be hard to do with a side condition: the condition has to be in the middle.

```
fun hype(f, x && (x != 0)? &&
            f(x)     ~  [a] &&
            f(1/x)   ~  [b])
   = [a,b];
```

Figure 2. Inside Condition

`||` is much less often used than `&&`. It does show up now and then, for instance, in balancing functional red-black trees [17]. A function might take either an integer or a string as argument:

```
fun f(x:int || x:string) = use(x);
```

One may also use a record to send named arguments to a function.

```
fun draw(<color, style>) {use(color,style);}
```

To make the arguments optional, one can supply default values with `||`:

```
fun draw(
        (<color> || "purple"~color)
     && (<style> || "square"~style)
    ) {use(color,style);}
```

The pattern `<color>` matches a record with a `color` field, and binds the variable `color` to that field's value. If that doesn't match, `"purple"~color` gets to try; it unconditionally succeeds, and binds `color`.

The negative pattern `!P`, which succeeds if `P` fails, is only rarely useful. To calculate the sublist of `L` before the *first* occurrence of `1`, despite possible matches that could be achieved by backtracking, use:

```
if (L ~ [ a && !1..., 1, b && _:int ...])
   use(a,b);
```

Negation – and, usefully, *double* negation – has the property of encapsulating bindings. The pattern `![x,$x]` matches everything but a list of two identical values, and it produces no bindings. (It does use an *internal* binding of `x`, but that binding does not escape past the scope of the `!`.) So, `!![x,$x]` matches a list of two identical elements, and produces no bindings. This idiom can be useful if one does not wish to introduce any variables – or to emphasize that variables are not needed outside of the pattern.

(This gives an amusingly intuitionistic flavor to Thorn's Booleans, both Boolean expressions and patterns. A positive match, `E~P` returns true or false, and yields some bindings. Negated once, either as `!(E~P)` or `E~!P`, it returns the opposite truth value and no bindings. Doubly negated, `!!(E~P)` or `!(E~!P)` or `E~!!P`, it returns the original truth value but, unlike the positive, no bindings.)

5.2.7 Regular Expressions `RE / P`

Thorn's regexp pattern has the form `RE / P`. The expression `RE` should evaluate to a string containing a Java-style regular expression. `P` is a pattern, generally a list pattern. The match succeeds if the subject is a string that matches the regexp `RE`, and the list of substrings from the groups appearing in `RE` matches the pattern `P`.

The most common pattern match uses a list pattern to bind the capture groups. To see if `x` consists of some `a`'s followed by some `b`'s, write:

```
if (x ~ "(a*)(b*)" / [xa,xb]) use(xa,xb);
```

But in practice the typical parsing problem isn't simply to chop a string up into pieces. The pieces are often needed in forms other than strings: as numbers, booleans, files, etc. We illustrate with the problem of parsing a pair of numbers separated by a colon: `"12:34"`. Here we use internal pattern matches and the `.int` method on strings to avoid needing the intermediate strings.

```
if (x ~ "([0-9]+):([0-9]+)"/
          [it.int ~ +i, it.int ~ +j])
  println("$i colon $j");
```

If we were coding an *expression* that became too large and intricate, we would instinctively extract subexpressions into variables or functions to simplify it. In most languages, there is no way to extract or use sub*patterns*. Thorn's pattern abstraction mechanisms let us write it more cleanly.

6. First-Class Patterns

Now that we have seen how patterns tightly integrate with the rest of Thorn, and how Thorn provides a rich pattern system, we take things one step further by also providing first-class patterns.

```
fun sum(t, t.Fork(left, item, right))
      = sum(t,left) + item + sum(t,right)
  | sum(t, t.Leaf(item)) = item;
```

Figure 3. Explicit Representation Parameter

6.1 Towards First-Class Patterns

The authors are not alone in feeling that, some months, two-thirds of the code they write consists of an endless series of blocks of the form:

1. Acquire a data structure `X`;
2. Determine if `X` satisfies some sanity properties;
3. If it does, rip it apart and put the pieces in variables $a, \cdots, z$.
4. (Optional: if it doesn't, produce a clear and informative error message.)

When one is unlucky enough to program in most mainstream languages, one is condemned to do this by a series of tests, conditionals, method calls, and bindings. If one is lucky enough to program in a language such as ML or its various progeny and inspirees, among others, one can use pattern matching, and combine the testing and extraction into a single function definition:

```
fun sum(Fork[left, item, right])
      = sum(left) + item + sum(right);
  | sum(Leaf[item]) = item;
```

The input to `sum` is destructured. In the first line, it is inspected to see if it is a `Fork` node. If it is, it must have three fields; these are bound to the variables `left`, `item`, `right`. If not, it is inspected to see if it is a `Leaf`, and, if so, its single field is bound to `item`. This definition is a marvel: concise without being terse, explicit without being redundant. Would that all code could be like this!

Of course, one might not be programming with a specialized datatype or class of trees. Especially in a scripting language, Python or Perl or PHP or Ruby or a hundred others, one might choose to use a built-in type to represent one's trees. Perhaps fork nodes are represented as three-element lists `[left,item,right]` and leaf nodes as one-element lists `[item]`. Pattern matching still works perfectly well:

```
fun sum([left, item, right])
      = sum(left) + item + sum(right);
  | sum([item]) = item;
```

This is basically the same code as the `Fork` example. The only difference is that the trees are represented differently. Being computer scientists, our immediate instinct is to try to abstract away the representation into a separate parameter `t`, as in Figure 3. That is, we want to make patterns *first-class*, so that they can be manipulated as data, as well as invoked as patterns. Few languages let us write this.

And of course one might not want to keep track of the representation parameter `t` by hand. Suppose that we have a function `rep(tree)` that takes a tree and returns its representation — as a record containing at least first-class patterns `Fork` and `Leaf`. Further, suppose that pattern selectors, such as `Fork` in Figure 3, can be expressions, using a variable `it` as the subject of the pattern match. We can then write a quite general tree walk, as in Figure 4, which can sum any sort of binary tree whose representation is registered with `rep` – even, say, trees represented by an array and an index n into it, with the children of the node at n being $2n$ and $2n+1$. Few if any current languages, save the extended Thorn of this paper, are capable of expressing this.

Making *functions* first class was not terribly difficult, and proved to be an extraordinarily powerful and fruitful topic. Doing the same to *patterns* is less revolutionary, especially in languages that already have first-class functions. But it is still not terribly difficult, and provides a pleasing increment of expressive power that has a variety of uses, as we shall demonstrate.

```
fun sum(rep(it).Fork[left, item, right])
    = sum(t,left) + item + sum(t,right)
  | sum(rep(it).Leaf[item]) = item;
```

Figure 4. Dynamic Representation

We have seen several reasons for wanting first-class patterns: to break complex patterns into simpler ones, to unify procedures that operate over similar but differently structured data, and so on. We will need an abstraction mechanism, an application mechanism, and a few odds and ends.

6.2 Function/Pattern Duality

There is a beautiful and elegant duality between (partial) functions and patterns. A partial function accepts n inputs, and produces one result, or fails. A pattern match accepts one subject, and produces n outputs, or fails.

This duality suggests that a *function* abstraction, like the following which accepts three inputs

```
fn (x,y,z) = [x,<why=y, zee=z>]
```

should be dualized to a pattern, like the following which produces three outputs named `x`, `y`, and `z`:

```
pat [x,y,z] = [x,<why=y, zee=z>]
```

Similarly, pattern test is dual to function application. Let `f` and `p` be the function and pattern above. Then a basic use of `f` is to apply it to three variables, say `a`, `b`, and `c`, and produce one value `r`:

```
r = f(a,b,c);
use1(r);
```

and a basic use of `p` is to match one value `r` and produce three variables, say `s`, `t`, and `u`:

```
if (r ~ p[s,t,u])
   use3(s,t,u);
```

Of course, the arguments to `f` can be any expression:

```
r = f(1+a(2), [b,b], m*c*c);
use1(r);
```

Dually, the arguments to `p` can be any pattern. Consider:

```
if (r ~ p[1, [b, $b], 2])
   use(b);
```

This first attempts to match the value of `r` against the body of the pattern `p`. If this fails, matching fails. Otherwise, it will produce three results, as given by the three formals of `p`'s definition, `x`, `y`, `z`. (These variables are *not* bound; indeed, they don't appear in this code at all.) It then tries to match the first result, formally `x`, against the first subpattern `1`. If this fails, matching fails. Then the second result, formally `y`, is matched against the second subpattern `[b, $b]`, which succeeds if it is a two-element list with equal elements. If this fails, matching fails. If it succeeds, it binds `b`. Finally, the third result is matched against the constant `2`. If this succeeds, the whole match succeeds, and `use(b)` executes with `b` suitably bound.

6.3 On Beyond Duality

Besides duality, we have two other demands on pattern abstractions. First, we want to have pattern abstractions usable to construct values as well as decompose them, as in Section 3.2. Second, we want to be able to provide input parameters to patterns, allowing patterns like *"match two strings concatenated with separator `sep`"*.

6.3.1 Pattern Punning in Thorn

Class names, suitably applied, can be used both as constructors to make values, and patterns to take values apart. For example, the code in Figure 5 squares every element of a binary tree. Note that the `Leaf` and `Fork` on the left-hand side of `sq` are used as patterns, and those on the right-hand side are used as constructors.

```
class Leaf(item){}
class Fork(left, item, right){}
fun sq(Leaf(i)) = Leaf(i*i);
  | sq(Fork(l,i,r) = Fork(sq(l), i*i, sq(r));
```

Figure 5. Pattern punning with classes

```
fun demo(Leaf) {
  t = Leaf(11);
  assert(t ~ Leaf[x] && x == 11);
}
Leaf1 = pat [x] = [x]
        new (x) = [x];
Leaf2 = pat [x] = <leaf=x>
        new (x) = <leaf=x>;
demo(Leaf1); demo(Leaf2);
```

Figure 6. Invertable Leaf

It is not absolutely necessary to do the same thing with first-class patterns. We could perfectly well keep track of some closures which would serve as constructors that go with patterns.

Indeed, not all patterns *have* associated constructors. The "familiar patterns" of Section 5.1 tend to have them; the others tend not to. Inverting `Leaf(x)` or `[x]` is easy; inverting `Leaf(x) || [x]` is not.

Still, in the spirit of making patterns as powerful as possible, we allow an optional constructor abstraction to be attached to a pattern abstraction. Constructors other than the default ones in Thorn are introduced (but not invoked) with the **`new`** keyword, so we use the same for the constructor attached to a pattern abstraction.

In Figure 6, we have a function `demo` that accepts a punned pattern abstraction `Leaf`, which, like the `Leaf` class of Figure 5, can either be used as a unary constructor, or as a unary pattern. `demo` first constructs a `Leaf` node `t` containing `11`, and then extracts the value in `t` and checks that it was `11`. `demo` is then called with two representations of leaf nodes: `Leaf1` represents them as singleton lists, and `Leaf2` represents them as records with a `leaf` field. Both of these pass the `assert` in `demo`.

In many cases, the **`pat`** and **`new`** clauses will have the same body. (We considered having syntax for the case where they did, but have not yet arranged for it.) A simple yet sensible case where the two do not have the same body appears in Section 6.3.3.

Since these values are **p**atterns plus f**un**ctions, and used for pattern punning, we call them *puns*.

6.3.2 Inputs to Patterns

Some advanced pattern constructs require input values, which govern or modify the pattern matching, as well as the more usual output values which are bound upon success. Consider a pattern that succeeds if the subject is a map with a value at key `k`, and binds that value. The key `k` is not a typical pattern variable in the sense of Section 6.2. While there is a sensible interpretation for matching on `k`, the typical use of this pattern will be with some *value* for `k`.

Since we have two kinds of arguments to patterns, we need some way to distinguish them. Marking them as **`in`** and **`out`** in the **`pat`** construct wouldn't work in an untyped language. There's no way to ensure that the same positional parameter is always input or always output. Since output parameters introduce new variable bindings, this approach makes it impossible to tell what variables are bound. We deemed this unacceptable.

So, for Thorn, we make the inputs and outputs explicit in the use of the pattern abstraction, separated by the symbol `~>`. In Thorn, input values appear to the left of the `~>`, and output values appear to the right. Figure 7 shows how to build an immutable map as an association list, *viz.*, a list of pairs whose first elements are keys and

```
fun memq(x, [[$x,y], tl...]) = +y;
  | memq(x, [[_,_],  tl...]) = memq(x,tl);
  | memq(x, [])              = null;
mem = pat[x ~> y] = memq(x,it) ~ +y;

a = [ [1,11], [2,[22]] ];
if (a ~ mem[1 ~> z])
   println("1 maps to $z ");
if (a ~ mem[2 ~> [_]])
   println("2 maps to a singleton");
```

Figure 7. Association-list Map and Accessor Pattern

```
between = pat[ [low,high] ~> x] =
  [_..., (low<=it && it<=high)? && x, _...];
```

Figure 8. Input pattern in abstraction

second elements are values. The `mem` pattern allows matching over the list; the `1` test retrieves an element if it's there (which it is), and the `2` test simply tests the value in the map against another pattern.

Note that, in a pattern *abstraction*, the input arguments are patterns and the output arguments are identifiers. The abstraction in Figure 8 takes a range, written for the sake of illustration[2] as a two-element list `[low,high]`, and searches the subject for a value within that range.

Conversely, in a pattern *application*, the input arguments are expressions and the output arguments are patterns. When `between` is used, the input argument must evaluate to a two-element list. The output, however, can be a pattern, which is matched against the value found by `between`. Here we bind it to a fresh variable in one line, and compare it to a constant in another:

```
if ([1,2,3,4,5] ~ between[[2,3] ~> x])
   assert(x==2);
if ([1,2,3,4,5] ~ between[[2,3] ~> 2])
   println("Yes!");
```

The input arguments of a pattern application may be any value. The core idea of first-class patterns is that pattern abstractions are values. So, input arguments allow one to write pattern abstractions that take other patterns as *formals*. *E.g.*, a pattern abstraction that matches a list of three elements, each of which matches the same input pattern `P`, is:

```
three = pat[P ~>] = [P[],P[],P[]];
```

(Output arguments are much more common than inputs, so `pat[x]=P` abbreviates `pat[~>x]=P` rather than `pat[x~>]=P`.)

A list of three ones can be matched thus.

```
isOne = pat[] = 1;
assert([1,1,1] ~ three[isOne ~>] );
```

(`isOne` is to the pattern `1` as the function $\lambda().1$ is to the value 1: a nullary abstraction of a constant.)

The concept of input arguments to patterns even appears in the standard list of Thorn patterns. In the regexp pattern `RE/P`, Section 5.2.7, the string `RE` is treated as an input, while the pattern `P` is treated as an output.

6.3.3 Example: Joining Strings

The pattern `sep[s~>x,y]`, appearing in Figure 9, matches a string whose body consists of the strings `x` and `y` with `s` between them.[3] It has one input and two outputs. The **`new`** side of the pattern

```
sep = pat[s ~> x,y] = "(.*)$s(.*)"/[x,y]
      new(s,   x,y) = x+s+y;
if ("10/6" ~ sep["/" ~> shillings, pence])
  assert(shillings=="10" && pence=="6");
ks = sep("/", "Kirk", "Spock");
assert(ks ~ sep["/" ~> "Kirk", "Spock"]);
assert(ks == "Kirk/Spock")
// Regexp separators violate the abstraction:
assert(ks ~ sep["../.." ~> "Ki", "ock"]);
```

Figure 9. Joining and Separating Strings

is ternary, taking the separator and the two strings to be joined. In this case, as in most cases where construction or extraction does much computation, the **`pat`** and **`new`** clauses are different, and have different (and non-dual) arguments.

7. Experiences

We can observe the use of patterns in a nontrivial corpus of Thorn code. This comprises most of the Thorn code (excluding only test cases) written at IBM, by programmers ranging from the implementer of the Thorn interpreter to summer interns with little relevant background. Most of this was written before first-class patterns were introduced, so they are underrepresented.

This corpus cannot be considered a scientific sample in any way. Some of the code was written to be exemplary and appear in tutorials, and may be regarded either as "the ideal for Thorn coding" or "cheating". Some of the rest was written as one-shot scripts, and may be regarded as either "degenerate" or "realistic". All results in this section should be taken with a colossal saltshaker.

The corpus is 24K lines of code. It contains 13K uses of patterns total – but 7K of those are variables, such as `x` in **`fun`** `f(x)=x*x` or `y` in `y=1`. The remaining 6K are nontrivial patterns, which thus appear every four lines of code on average. They are summarized in Table 4.

7.1 Pattern Usage by Kind

Kind	Fancy	Count	Fraction
Record `<···>`		2,455	41.2%
List `[···]`		1,068	17.9%
Wildcard _		641	10.8%
Literal		329	5.5%
Positivity test `+P`	Yes	278	4.7%
Interpolate `$(E)`		253	4.2%
Type test `P:T`		210	3.5%
Internal pattern match `E~P`	Yes	207	3.5%
Boolean `&&`	Yes	179	3.0%
Regular expression `RE/P`	Yes	157	2.6%
Type constructor `T(···)`		103	1.7%
Expression test `(E)?`	Yes	64	1.1%
Boolean `||`	Yes	11	0.2%
Boolean `!`	Yes	5	0.1%

Table 4. Nontrivial pattern usage by type (see also Table 3)

The **Fraction** is the fraction of the 6K nontrivial patterns.

Dissecting records (42%) and lists (18%) were by far the most common pattern operations, perhaps because most Thorn data comes in records and lists.

The fancy patterns of Section 5.2 collectively make up about 15% of nontrivial pattern use. This is a substantial fraction, and shows that some, at least, of the fancy patterns are fundamental programming tools in Thorn. Non-null testing embodies a core Thorn idiom (Section 5.2.4), and is the most frequent of these. Internal pattern matching is a pattern-matching duct tape, useful for nearly anything. And-patterns are used like ML's **`as`**, giving a name

[2] Or, perhaps, a misguided attempt to imitate closed intervals $[a, b]$ from analysis. In any case, there is no reason why `low` and `high` could not be presented as two separate arguments `pat[low,high~>x]=...`, save that does not illustrate input patterns.

[3] For the sake of illustration, we ignore the cases in which the separator appears in `x`, or when regexp-active characters appear in `s`.

to a pattern and dissecting it further, and are a fundamental tool. The other fancy patterns are less useful. Though even the rarest, negation, allowed convenient expression of things that would have been awkward to say in other ways.

7.2 Pattern Usage By Context

Another way to look at pattern usage is to see which kinds of pattern-bearing statements used nontrivial patterns. For example, we count `for(x <- L)` as a trivial use of patterns; it's an ordinary `for` loop iterating over a list. But `for(<a,b> <- L)` is a nontrivial use; it iterates over a list of records, and extracts two fields from those records. We may also measure the complexity of these patterns. We choose to count the number of nontrivial syntax nodes appearing in the pattern. So, `[a]` would have a complexity of 1; the list pattern is nontrivial, the variable is trivial. `[[[a]]]` would have a complexity of 3. The results are in Table 5. For this table, the `for` in a list comprehension `%[x | for x <- 1..10]` is counted in with `for` loops, and similarly for other comprehension pieces.

Context	Uses	Nontrivial	Complexity
Binding statement	2,758	3%	2.0
Function definition	2,120	19%	1.6
If statement	726	37%	1.7
For statement	499	43%	1.3
Match statement	434	87%	1.7
While and until loops	430	0.9%	4.3
`&&` expression	46	51%	1.4
Anonymous function	28	18%	1.3

Table 5. Pattern usage by context (see also Table 2)

Binding statements use patterns the most, but usually use them in uninteresting ways. This is a result of defensive programming. `x = E` cannot fail. `<a,b> = E` can fail, if `E` turns out not to evaluate to a suitable record. An experienced programmer may choose to write it as `if(E~<a,b>)` as a matter of routine caution.

And, indeed, a third of `if` statements have a nontrivial pattern, and most of the `&&`'s appear in `if`s. Even though all the standard uses of `if` are used in the normal way in Thorn, a substantial fraction of them were used with patterns in nontrivial and often idiomatic ways.

Function declarations (including methods and communications) and anonymous functions sometimes used patterns. An unscientific glance through the corpus suggests that in Thorn code written to be robust or exemplary, function declarations often used patterns, for documentation and for pseudo-typing. More casually written code, one-off scripts and such, tended to omit them.

`for` loops frequently involve some nontrivial dissection of the value being looped over. For example, iterating over a map with key/value pairs often had the form `for(<k,v> <- m)···`.

`match` statements, `catch`-blocks, and `receive` statements are grouped under the "Match statement" line. They generally had multiple clauses, most of which were nontrivial. For example, a `catch` might test for an I/O exception, then a null exception, and then use a trivial match to catch any other sort of exception. Communication, similarly, tended to inspect and categorize incoming messages, with many nontrivial clauses.

`while` loops rarely take advantage of the binding behavior of Section 4. This may be due in part to other Thorn constructs, such as the `find` statement, which are better suited for searching than a vanilla `while` loop. Indeed, nearly all the loops had the form `while(flag)` or, in reactive code, `while(true)`.

7.3 Conclusions on Pattern Usage

Many of the fancy pattern constructs find substantial use in Thorn code. Nearly every pattern construct gets noticeable usage, and nearly every construct that supports patterns often uses them in nontrivial ways.

```
fun norm([x,y]) = (x*x+y*y).sqrt;
fun plus([x1,y1], [x2,y2]) = [x1+x2, y1+y2];
```

Figure 10. Concrete Representation of Points

The "patterns as untyped types" concept is at least partially substantiated by this corpus – unsurprisingly, as it was at least partially inspired by the experience of coding this corpus. Some fraction of the pattern checking is done by function declarations, where types would be declared. More of it is done in `if` statements, as a matter of defensive programming.

The zoo of pattern constructs consists largely of useful workhorses. Boolean `||` and `!` patterns, and the interaction of patterns with `while`, and perhaps test patterns, are probably most useful as a matter of intellectual consistency, rather than important programming devices. Everything else gets used enthusiastically.

7.4 Implementation

Thorn is currently implemented as a proof-of-concept interpreter, designed for correctness and ease of modification rather than efficiency. Performance was a secondary consideration. A casual investigation suggests that the performance of pattern matching is comparable to that of the rest of the interpreter.

In the Thorn interpreter, implementing first-class patterns was a routine matter, a morning's worth of coding. First-class patterns are similar enough to first-class functions so that a solid implementation of one leads naturally to a solid implementation of the other. The remaining pattern constructs can be implemented straightforwardly.

The problem of optimizing pattern matches in a dynamically typed language, especially with a rich set of pattern primitives and first-class patterns, remains to be investigated. As with optimization of closures, it is likely to be a challenging research topic.

7.5 Use Case for First-Class Patterns: Refactoring

Scripts frequently start off using concrete data structures, lists and maps and such. For example, let's represent points in the plane as two-element lists `[x,y]`, as in Figure 10. We list a few of the many functions that might be involved in plane geometry.

At some point in the program's evolution, one may wish to convert the concrete points to a more abstract representation, using a class `Point`. The traditional approach to this refactoring is approximately:

1. Avoid doing it for as long as possible. Consider it a victory if the program is abandoned before it needs to be refactored.
2. Make sure the program passes a comprehensive test suite.
3. Introduce principled new classes for the needed abstractions, but don't use them. The test suite should still pass.
4. Change all uses of two-element lists, or variables that refer to them, so that they use the new classes.
5. Run the test suite on the updated program. Notice that it's all terribly broken. There is no realistic chance of making hundreds of changes to a script in a single huge lump and having it work.
6. Spend a long and miserable eternity trying to fix the program.

Making this change is an inherently difficult chore. You must look at every list, and everything that could be a list, and see if it is a point or not. If you are lucky, you will generally be able to tell, and there won't be too many things that are sometimes points and sometimes not in ways that matter.

The chore is made far worse by the difficulty of testing. Once the refactoring has been started, the code cannot possibly work. Some places refer to points as `[x,y]` and others as `Point(x,y)`. The program won't run.

Even when it is possible to run the test suite, errors will be very hard to localize. When one changes a program and introduces an error, the first place to look is at the changed code. When the whole program has been modified, there's no good starting place.

This is one of the most painful code changes to make. Avoiding it – or, for the painfully idealistic, designing one's program from the beginning with the right abstractions for its unknown future evolution – is often the best strategy.

7.5.1 Abstraction with First-Class Patterns

With pattern abstraction, you can do much better. You won't save any of the obligatory work, but you can do it gradually and test it as you go, saving some of the optional pain.

First, define `Point`, not as a class, but as a pattern abstraction with constructor. This should match the original, concrete representation. In our example, this is:

```
Point = pat [x,y] = [x,y]
        new (x,y) = [x,y];
```

Now, change all the uses of the concrete representation into uses of the abstraction. This is the same intellectual labor that would be done without first-class patterns.

Note, crucially, that this change can proceed gradually. The abstraction simply provides another name for the concrete representation. Mixing the two is no more problematic than mixing the expressions `x*x` and `square(x)`. Partially changed code will look like Figure 11, where abstract and concrete representations coexist.

```
Point = pat [x,y] = [x,y]
        new (x,y) = [x,y];
// The following code has been abstracted
fun norm(Point[x,y]) = (x*x+y*y).sqrt;
// The following has not been
fun plus([x1,y1], [x2,y2]) = [x1+x2, y1+y2];
```

Figure 11. Partially abstracted points

In particular, the test suite can be run as often as desired, and, since no data manipulations have changed, should run without error every time. In practice errors always happen, but they will be normal small coding errors.

Finally, the pun may be replaced by a class `Point`. Semantic errors show up at this point, but at least they are not accompanied by droves of other errors.

8. Conclusions

This paper makes two primary contributions. The first contribution is a new way of thinking about patterns. We argue that pattern-matching (in the sense of ML) makes scripting in dynamically typed languages (unlike ML) more robust. The second contribution is a rich pattern system, consisting of a diverse set of interacting features. Each individual feature, taken by itself, represents only a minor contribution, but when they are all taken together, a powerful and novel way of scripting emerges. We implemented our pattern system in Thorn, and we have written thousands of lines of code in Thorn that exercise patterns, sometimes in surprising or entertaining ways. Our pattern system (contribution 2) supports our philosophy of Robust Scripting via Patterns (contribution 1).

Thanks

Thanks to John Field and Dave Grove for many helpful discussions on the ideas in this paper.

References

[1] Bard Bloom, John Field, Nathaniel Nystrom, Johan Östlund, Gregor Richards, Rok Strniša, Jan Vitek, and Tobias Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 117–136, October 2009.

[2] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language, 1980.

[3] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 273–298, 2007.

[4] Michael D. Ernst, Craig S. Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *European Conference for Object-Oriented Programming (ECOOP)*, pages 186–211, 1998.

[5] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008.

[6] Felix Geller, Robert Hirschfeld, and Gilad Bracha. Pattern matching for an object-oriented dynamically typed programming language. Technical Report Hasso Plattner Institute Technical Report 36, University of Potsdam, 2010.

[7] Ralph E. Griswold, J. F. Poage, and Ivan P. Polonsky. *The SNOBOL 4 Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1968.

[8] Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. Matchete: Paths through the pattern matching jungle. In *Practical Aspects of Declarative Languages (PADL)*, pages 150–166, 2008.

[9] Paul Hudak, Simon L. Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph H. Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Richard B. Kieburtz, Rishiyur S. Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[10] Roberto Ierusalimschy. *Programming in Lua*. Lua.org, second edition, 2006.

[11] Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming (JFP)*, 19(02):191–225, 2009.

[12] JetBrains. Kotlin language project page. `http://confluence.jetbrains.net/display/Kotlin/`.

[13] P. J. Landin. The next 700 programming languages. *Communications of the ACM (CACM)*, 9(3):157–166, March 1966.

[14] Jed Liu and Andrew C. Myers. JMatch: Iterable abstract pattern matching for Java. In *Practical Aspects of Declarative Languages (PADL)*, pages 273–298, 2003.

[15] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML - Revised*. The MIT Press, rev sub edition, May 1997.

[16] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In *International Conference on Compiler Construction (CC)*, 2003.

[17] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, June 1999.

[18] Python programming language – official website. `http://python.org/`.

[19] Don Syme, Gergory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *International Conference on Functional Programming (ICFP)*, pages 29–40, 2007.

[20] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *Transactions on Programming Languages and Systems (TOPLAS)*, 30(6), 2008.

[21] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM.

[22] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly, third edition, 2000.

[23] Alessandro Warth and Ian Piumarta. OMeta: An object-oriented language for pattern matching. In *Dynamic Languages Symposium (DLS)*, pages 11–19, 2007.

[24] Andrew K. Wright. Pattern matching for Scheme, 1996. The match special form is part of PLT Scheme's MzLib library.