Thorn

A Robust Scripting Language

Tobias Wrigstad, Johan Ostlund, Gregor Richards, Jan Vitek (Purdue)
Bard Bloom, John Field, Nathaniel Nystrom (IBM)
Rok Strnisa (Cambridge)

Outline of the Talk

- A bit of philosophy
  - scripting vs. robustness
- Focus on one feature set
  - Patterns, Queries, and Tables
  - Why it's nifty for a scripting language
  - Why it helps robustness.
- **Warning:** Thorn is a work in progress
  - Nearly everything in this talk is implemented.

The Points of Thorn

- Competitor to Python, PHP, Ruby, etc.
  - Popular constellation of scripting langs for the web.
- Path to industrial strength
  - Hope: better support for large programs
- Seduce programmers to good software engineering
  - Provide immediate value
- No particular intent of originality
  - Steal good ideas from everywhere
    - (OK, we invented some too)
  - Attempt at harmonious and powerful merge.
- Syntax for:
  - A common case – for scripting
  - The general case – for robust

Scripting Style

- **Purposes:**
  - *To quickly toss together useful little gadgets*
    - *Count #occurrences of words in a novel.*
  - *Quick prototyping*
  - *Rapid, frequent changes*
- Light Syntax
- Weak Data Privacy
- Dynamic Typing
- Powerful Data Structures

The Fate of Scripts

- Scripts don't stay small
  - Little utility programs get more features.
    - *Actually, I want a concordance, not just word counts.*
- And the features that made scripting easy make robust programming hard.
  - Inefficient, hard to maintain
  - Often, those little scripting programs grow up to be monsters...

Thorn: Script $\longrightarrow$ Robust

- **Goal:** Scripts can be gradually evolved into robust programs.
- Dynamic Types
  - *But: you* can *provide static types*
- Lightweight Syntax
  - *But: light syntax isn't a* problem *for robustness*
- Weak data privacy by default
  - *But: you* can *make things private; nice module system.*
- Powerful built-in aggregates
  - *But: that's not a bad thing.*
- **Caveat:** Little experimental evidence.

Illustration: Types and Patterns

- Thorn, like most scripting languages, is untyped
- Static types are good for robust programs
  - Error catching, better compilation, etc.
- Static types are *actually* simple static assertions
    - *f is a number; L is a list*
  - Other kinds of static assertions also useful
    - *f > 0; L is length 3*
- So ... Let's entice programmers into wanting to supply such assertions.
  - Make them useful for programming.
  - (Not just verification and good practice, which scripters don't care about.)

Thorn's Approach

- Thorn has patterns
  - Used in many places
  - Very powerful and convenient
- Patterns explain what programmer expects

```
fun f1(lst) {
  if (lst(0) == "addsq")
     return lst(1)*lst(1) + lst(2)*lst(2);
  }

fun f2(["addsq", x, y]) = x*x+y*y;

fun f3(["addsq", x:int, y:int]) = x*x+y*y;
```

- Compiler can use this information
  - (We hope)

Patterns are everywhere

- fun f(pat1, pat2) ... : function arguments
  - `fun squint(x:int) = x*x;` // integer square
- Exp ~ Pat: boolean test
  - `if (x ~ [1, y])` // match 2-el list with car=1
- pat = Exp: cf. ML's let.
  - `z = 1;` // introduce new var z, bound to 1.
  - `[h,t...] = nonemptyList();`
  - // Exception if it doesn't match.
- `match(Exp) { Pat1 ... Pat2 ... ... }`: match stmt.
- `receive` stmt.

Patterns and Bindings

Match empty list

```
fun sum([]) = 0;
  | sum([x,y...]) = x+sum(y);
```

Match list with car x and cdr y

Does it match? If so, bind x,y, in 'then' clause

```
fun sum'(lst) {
     if (lst ~ [x,y...])
       x + sum(y);
     else {0;}
  }
```

Pattern Match + Control Flow

- Match bindings available in guarded code:

```
var L := [1,2,3]; var s := 0;
while (L ~ [x,y...]) {
    L := y; s += x;
}
```

Use x,y

x,y out of scope

- until guards code *after* loop:

```
p = Person();
do {
    p.seekSpouse();
} until (p.spouse ~ +q);
liveHappily(p,q);
```

q out of scope

Match non-null, bind to q

q in scope

Other Patterns

- (BoolExp)? succeeds if BoolExp evals to true
- P && Q matches things that match both P and Q.
  - cf. ML's `as`:
    - `fun f(L && [x,y...]) = g(L,x,y);`
  - Look for two elements in either order:
    - `if (L ~ [_..., 1, _...] && [_..., 2, _...])`
  - Test side condition in mid-match
    - `fun sqrt(n:float && (n>=0)? )`
- P || Q matches if either P or Q does
  - `fun f(n : int || n : string) = 3 + n;`
- !P matches if P doesn't.
  - No bindings at all.
- and a few more

Tables (and maps)

- Table: the big mutable data structure.
  - one or more keys
  - one or more non-keys.
  - akin to maps and database tables.
- Word-counting script:
  - `t = table(word){var n;};`
  - `t.ins( {: word: "provenance", n: 1 :} );`
  - `t("provenance").n`

{: ... :} is a record

- Tables are super-maps:
  - Multiple keys, multiple values.
  - Maps available as syntactic sugar on tables.
- Program evolution:
  - avoid parallel maps; add new fields to a single table
  - `t = table(word){var n, where;};`

Queries

- Special syntax for common cases of searching and constructing
- List comprehension:
  - `%[ i*i | for i <- 2 .. 4] == [4,9,16]`
  - `%[ i*i | for i <- 2 .. 4, if prime?(i)] == [4,9]`
- Quantifiers
  - ```
    fun prime?(n) =
      ! %some(n mod k == 0 |
          for k <- 2 .. n, while k*k <= n);
    ```
    ```

Table queries

Build a table with key n, whose value is i...

and non-keys for i^2 and i^3...

varying i, as usual for queries

```
powers = %table(n=i){
           sq = i*i;
           cube = i*i*i;
           | for i <- 1 .. 10};
```

Stop on the first iteration...

... and return n ...

```
cubeRootOfEight = %find(
   n
   |
   for {: cube: 8, n:n :} <~ powers)
```

iterating over rows whose cube field is 8
(and bind the n field too)

Group Query

- Iterate over stuff, *aggregating* information.

```
Words = novel.split("[^A-Za-z']+");
words = %[w.toLower() | for w <- Words];

counts = %group(word=w){
    n=%count;
    | for w <- words};

assert(counts("cloaca").n == 5);
// That's probably too high.
```

collect & lowercase all the words in the novel.

Produce a table whose keys are words w

... and whose n is the number of occurrences of that word ...
(More general aggregations available)

... iterating over the words.

Summary

- Patterns, queries, and tables are a typical Thorn feature
- Very expressive
  - Common cases, general cases built in
  - So it should appeal to scripters
- Designed with robustness in mind
  - Patterns give clues about static info, like types
  - Queries are understood by the compiler
  - Tables anticipate program evolution

The Rest of Thorn

- (Same design principles)
- Object system:
  - Multiple inheritance, but simple
  - Promotes (but doesn't require) immutability
- Module system:
  - Based on upcoming Java Module System
- Concurrency & Distribution
  - Erlang-flavored.
  - Message-based; no shared data
    - (Except constant objects)
  - High-level (rpc-ish) and lower-level (send immutable object) communication.

That's all!

- (But more details could follow)

Influences

- Object-Oriented Programming
  - Scala, Java, C++, Kava
- Concurrency
  - Erlang
- Pattern Matching / Destructuring
  - Lisp, ML, SNOBOL
- Powerful Built-In Data Structures
  - ML, CLU
- Scripting Style
  - Python, Perl, PHP, Ruby, Lua
- Queries / Comprehensions
  - SETL, SQL

Accumulators

- Another way to add up a list (*cf.* foldl)

```
%after(s | for x <- L,var s := 0 %then s+x);

// Amounts to:

{
    var s := 0;
    for (x <- L) {
       s := s+x;
    }
    s;
}
```

Accumulators in %group

- The full generality of %group:
- Given a list of pairs [x,y], compute $\Sigma y^2/n_x$ for each x, where n_x = the number of occurrences of that x.

```
stat = %group(x = x') {
        n = %count;
        sy =  %first y*y
              %then sy + y*y
              %after sy/n ;
        | for [x',y] <- L
        };
```

Value for first iteration for given x'

Value for later iterations for same x'

After all iterations are done

Thorn Object System

- Class-based
  - Less flexible than Lua, Self
  - More robustifiable.
- Multiple Inheritance
  - Troublesome cases forbidden
    - No data diamonds
    - Method ambiguities must be resolved
  - What's left is good for
    - Mixins
    - Many simple examples
- Various safety and convenience features.

Object Example

```
class Named {
  val theName;
  method name() = theName;
  new Named(name'){
    theName = name';
  }
}Named

kim = Named("Kim");
```

val: read-only
var: read-write

(1) no shadowing, ever
(2) primes allowed in id

one-time binding to val

`this` can't escape ctor

optional name on close brace

functional syntax for ctor call

Objects, cont.

x and y are: (1) public val fields; (2) params of implicit ctor; (3) more...

```
class Point(x,y){}

class NamedPoint(x,y,name)
    extends Point(x,y),
    Named(name)
    {}

np = NamedPoint(0,0,"Origin");
```

NamedPoint's x and y are Point's x and y.

Multiple Inheritance

Inserting values into strings. (Scripty)

```
class Computer(sn) {
   method name() = "Comp$sn";
}

class NamedComputer(sn,name')
  extends Computer(sn), Named(name')
  {
    method name() = super@Named.name();
  }
```

A name() method is required to break ambiguities, since both parents have one.

"Use the one from Named"

Classes and Patterns

- Classes define extractor patterns:
  - `class Named(name){...}`
  - induces a pattern Named(p):
  - `if (person ~ Named(n)) { print("Name is $n");}`
  - `if (person ~ Named("Kim")) {print("Hi, Kim.");}`

Yes+ / No Idiom

- Lots of functions are *partial*.
  - Return either "I found one, and here it is" or "No, there is none."
- Other languages express this chaotically:
  - Many Java methods return null for "no"
    - *Fails if you can find "null" – as in Map.*
  - Java's s.index(t) = -1 for "no"
    - *Only works when there's a value that cannot be an answer*
  - Java's Integer.parseInt(s) throws an exception for "no"
    - *Heavy in "no" case*
  - Have two functions: Map.containsKey(k), Map.get(k)
    - *Typically requires two searches*
  - ML: "t option" type with values "some(x)" and "no"
    - *Good, but constructs a new object for each "some"*

Thorn's Yes+/No Idiom

- Expression +e is a non-null encoding of e
- Pattern +x undoes + and binds result to x
  - fails on null

Match value of k

"Yes, it's v"

"Not there"

```
fun assoc(k, [[$(k), v],_...]) = +v;
  | assoc(k, [])                = null;
  | assoc(k, [wrong, rest...]) = assoc(k, rest);

clues = [ [1, true], [2, null], [3, "fish"] ];
```

- Idiom for getting answer:

```
 if (assoc(2, clues) ~ +cl) {
    assert(cl == null);
 }
```

Detect positive answer; undo + operation; bind result to cl

Fine Points of +

- Any 1-1, non-onto function would do...
  - *e.g.*, ML's some(e) and none
- We want to avoid allocating as much as possible
  - +x is designed to support this idiom
  - +x == x for most x's (constants, lists, objects, etc.)
    - Extra benefit: quick to compute.
  - +null $\neq$ null.
  - +null is an otherwise boring value
    - "Nullities" +null, + + null, + + + null, etc.
      - All distinct
      - All boring
    - Flyweight pattern so they don't need to be allocated often.