Thorn

A Robust, Distributed Scripting Language

Bard Bloom, John Field (IBM)
Nathaniel Nystrom (U.T. Arlington)
Johan Ostlund, Gregor Richards, Jan Vitek (Purdue)
Rok Strniša (Cambridge)
Tobias Wrigstad (Stockholm University)

Precedents

- Steal good ideas from everywhere
  - (OK, we invented a few too)
- The art is in the harmonious and powerful merge
- Some influences:
  - Java, Scala, Perl, Python, Smalltalk, Ruby, Haskell, Erlang, ML, Scheme, Lisp, SETL, SQL, CLU, SNOBOL, Kava, Concurrent ML.

Scripting vs. Industrial Languages

Scripting

- Favor Coding Speed
  - Untyped
- Favor Common Cases
  - cons-cell lists
- Flexibility
  - Python objects
- Dynamicity
  - eval
- Convenience
  - open data structures

Industrial

- Favor Reliability
  - Typed
- Favor General Case
  - Java collections
- Straightforwardness
  - Java objects
- Staticness
  - code analysis
- Abstraction
  - access control

Thorn's Position

Scripting

- Favor Coding Speed
  - Untyped
- Favor Common Cases
  - cons-cell lists
- Flexibility
  - Python objects
- Dynamicity
  - eval
- Convenience
  - open data structures

Industrial

- Favor Reliability
  - Typed
- Favor General Case
  - Java collections
- Straightforwardness
  - Java objects
- Staticness
  - code analysis
- Abstraction
  - access control

The Points of Thorn

- Scripting language for network and web
  - cf. Python, PHP, Ruby, Clojure, etc.
- Path to industrial strength
  - Hope: better support for large programs.
- Seduce programmers to good software engineering
  - Provide immediate value
- Syntax for:
  - A common case – for scripting
  - The general case – for robustness
- Topics in this talk:
  - Distribution and Concurrency
  - Sequentially: Patterns, Queries, Tables, Classes

Scripting Style

- **Purposes:**
  - *To quickly toss together useful little gadgets*
    - *(e.g., example authentication server)*
  - *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 → 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.*

Example: Distribution/Concurrency

- Scriptily:
  - Easy construction of *component*s
  - Lightweight syntax
  - Primitives for messaging.
  - Most data is transmissible.
- Robustly:
  - Isolation
  - Single thread per component
  - Localized state
  - Messages passed by *value* (copied)
  - Localized faults; no propagation of exceptions

Code Example: Ping-Pong Game

Start a *component*

isolated, mutable state

unidirectional communication

unidirectional send

bidirectional communication

communicate forever

bidirectional send (RPC)

```
fun pp(name) = spawn {
  var other;
  async volley(n) {
    if (n == 0) println("$name misses");
    else {
        other <-- volley(n-1);
        println("$name hits.");
      } }volley
  sync playWith(p) { other := p; }
  body{ while(true) serve; }
}spawn;

ping = pp("Ping"); pong = pp("Pong");
ping <-> playWith(pong);
pong <-> playWith(ping);
ping <-- volley(10);
```

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;
```

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.
- **Approach: convenience, not minimality**

Patterns and Bindings

Match empty list

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

Match list with car x and cdr y

Match list with *integer* car x and cdr y

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

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

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

Matching and Scopes

- 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 after loop

Detail Work: Instance Variables

- Scripty style: instance variables are public
  - `a.x := b.x + c.x;` // very convenient
- Robust style: generally instance vars private
  - Change representation rectangular → polar
  - All direct field accesses become invalid
- Good Java-or-whatever style:
  - non-public instance variables
  - getters and setters for access
- Thorn approach:
  - instance variables are all protected
  - getters and setters automatically supplied
    - Unless programmer writes them
  - Syntactic sugar to look like field access

Thorn Instance Variables

Define ctor, x, y, and more

```
class Point(x, var y) {
  var color := "blue";
  def `y:=`(y') {throw "Nope!";}
  def shove! {y += 1;}
  // implicit: def x = x;
  // implicit: def `color:=`(c) { color := c; }
}

p = Point(1,3);

assert(p.x + p.y == p.color.len);

p.color := "green";
p.shove!;
assert(p.x + p.y == p.color.len) ;
// BAD: p.y := 8;
```

another instance var

no public y:=

y can be changed

access fields

call setter

mutate y

would throw "Nope!"

Programming in the large *in the small*

- Simple authentication server
  - Stores names and passwords
    - Passwords aren't stored in cleartext
  - Counts failed attempts
- Module for sharing types and functions
- Class to encapsulate data and behavior
  - *pure* class: immutable, transmissible
- Table to store local state
  - Many fields (and potentially many keys)

Module

debugging-level security

pure class

immutable field

one and only binding (in ctor)

== works on strings

short method definition

optional end token

```
module CRYPT {

  fun crypt(s) = s.capitalize;

  class Password:pure {
    val encrypted;
    new Password(e:string) {
      encrypted = crypt(e);
      }new

    def is?(e:string) = (encrypted == crypt(e));
  }Password

}module
```

Authorization Server

```
import CRYPT.*;

users = table(name){var pw: Password; var fails:int; };

sync register(name, pw:Password) {
  unless (users.has?(name)) {
    users(name) := {: pw, fails:0 :};
  }}register

sync confirm?(name, attempt:string) {
  if (users(name) ~ {: pw :}) {
    if (pw.is?(attempt)) return true;
    else {
      users(name).fails += 1;
      return false;
    }
  }
  else {return false;}
}confirm?

sync nFails(name) =
  (n if users(name) ~ {: fails:n :} else null);
```

Use module

indexed by name, two data fields

record ctor

abbrev. pw:pw

insert row

test presence and bind pw

method call

mutate table

Authorization Client

RPC

```
auth <-> register("Bard", Password("þ"));



//Now, try to guess it
pre's = ["thorn", "þ", "sythyry"];
suffs = ["", "09", "123"];
```

list comprehension

```
g's = %[p+s | for p <- pre's, for s <- suffs];
```

Seek first match and bind 'g' – same controls as comprehension

```
find(for g <- g's, if (auth <-> confirm?("Bard", g))) {
    println("Cracked it -- $g");
  }
else {
    println("No clue.");
  }
```

Status

- Interpreter:
  - Complete reference implementation
  - Goals: testbed, correctness
- Compiler:
  - Being updated to current version of Thorn
  - Compiles Thorn to JVM
  - Goals: Strong implementation of Thorn.
  - Takes advantage of types and patterns
- Web Portal
  - http://www.thorn-lang.org

Conclusion

- Scripting language designed for robustness
  - Encourages some good software engineering practices
  - (by making them easy and immediately helpful)
- Generality and Power
  - Full programming language
  - Sugar to sweeten many common patterns
- Core features
  - Distribution / concurrency
  - Multiple-inheritance object system
  - Rich set of built-in types
  - Patterns
  - Queries
  - Modules
-