Thorn

Robust, Concurrent, Extensible Scripting on the JVM

Bard Bloom, *John Field (IBM)*

Nathaniel Nystrom (UT Arlington)

Johan Ostlund, Gregor Richards, Jan Vitek (Purdue)

Rok Strniša (Cambridge)

Tobias Wrigstad (Stockholm University)

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
- Generous
  The general case
    message passing, for loops
  Some common cases
    RPCs, queries

Thorn Features

- Distribution and concurrency
  - Actors-style, with messaging (and RPCs)
- Built-in types
  - Lists, records, tables, full multiplanar Unicode
- Classes
  - Multiple inheritance
- Patterns and Queries
- Module System

Scripting: Word Frequency

```
words = "story.txt".file.contents.split("\\W+");
wc = %group(word = w.toLower)
            { n = %count; | for w <- words };
sorted = %sort["%3d %s".format(n,word)
              %> n %< word
              | for {:word, n:} <- wc];
println( sorted.joined("\n") );
```

```
121 the
 85 and
 52 a
 52 of
 37 to
 30 in
 29 they
 24 that
 21 skagganerax
```

The Fate of Scripts

- Scripts don't stay small
  - Little utility programs get more features.
- Easy scripting ➠ not so robust
  - Inefficient, hard to maintain
  - Those little scripting programs grow up to be monsters...

Scripting vs. Robust

SCRIPTING	ROBUST
Coding Speed dynamic typing	Reliability static typing
Favor Common Cases cons-cell lists	Favor General Cases Java collections
Flexibility Python objects	Straightforwardness Java objects
Dynamic eval	Static code analysis & refactoring
Convenience open data structures	Abstraction access control

Thorn's Position

SCRIPTING	ROBUST
Coding Speed dynamic typing	Reliability static typing
Favor Common Cases cons-cell lists	Favor General Cases Java collections
Flexibility Python objects	Straightforwardness Java objects
Dynamic eval	Static code analysis & refactoring
Convenience open data structures	Abstraction access control

The Art of Thorn: Better than zero-sum tradeoffs

Example: Distribution/Concurrency

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

Ping-Pong Game

```
fun pp(name) = spawn {
  var other;
  async volley(n) {
    if (n >0) {
        println("$name hits.");
        other <-- volley(n-1);
    }
  }
  sync playWith(p) { other := p; }
  body{ while(true) serve; }
};
ping = pp("Ping");
pong = pp("Pong");
ping <-> playWith(pong);
pong <-> playWith(ping);
ping <-- volley(10);
```

Start a *component*

isolated, mutable state

unidirectional communication

unidirectional send

bidirectional communication

communicate forever

bidirectional send (RPC)

Example: Types

- Thorn is dynamically typed
- Static types are good for robust code
- Static types are simple assertions

    F is a number; L is a list

  Other assertions are useful

    F > 0; L.len == 3

- Entice programmers into supplying them

  Make them useful for coding

Example: Patterns (and types)

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

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

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

Patterns are everywhere

- Function Arguments

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

- Boolean Test and Bind

```
if (x ~ [1, y]) { println(y); }
```

- Bind or Die

```
  z = 1;
  [h,t...] = stuff();
```

- Match

```
match (x) {
  []    => "empty"
 |[y]   => "singleton"
 |[_...,1,_...] &&
  [_..., 2, _...]
        => "has 1 and 2"
}
```

- Receive

- **Approach: convenience, not minimality**

Example: Instance Variables

- Scriptily
  - Instance variables all public
- Robustly
  - `private` and `protected`
  - Use getters and setters
- Thornily
  - Sugar for getters and setters
  - Instance variables all protected
  - Getters and setters are generated…
  - … unless programmer supplies them

```
p.y := p.y + 1;
p.y += 1;
```

```
p.setY(p.getY()+1);
```

```
p.y := p.y + 1;
p.y += 1;
```

Example: Thorn Instance Variables

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

Programming in the large ***in the small***

- Simple authentication server
  - Stores names and passwords
    - Not cleartext
  - Counts failed attempts
- Module for sharing types, functions, state
- Class to encapsulate data and behavior
  - *pure class: immutable, transmissible*
- *Table to store local state*
  - *Many fields*

Module

```
module CRYPT {
  fun crypt(s) = s.capitalize;

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

Authorization Server *(stripped down)*

```
import CRYPT.*;

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

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

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

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

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
  - Plugins for extensibility.
- Web Portal
  - http://www.thorn-lang.org
- Demo

Conclusion

- Scripting language designed for robustness
  - Encourages some good software engineering (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

Backup Slides

Patterns and Bindings

Match empty list

Match list with car x and cdr y

```
fun sum([]) = 0;
  | sum([x,y...]) = x+sum(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(L) {
     if (L ~ [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;
  }
```

- `until` guards code *after loop*

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

Authorization Client

RPC

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



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

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

list comprehension

Seek first match and bind 'g'

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

Precedents

- Steal good ideas from everywhere
  (OK, we invented a few)
- The art is in the harmonious, powerful merge
- Some influences:

Clu	Java	Python	SQL
Concurrent ML	Kava	Ruby	Scala
Erlang	ML	SETL	Scheme
Haskell	Perl	SNOBOL	Smalltalk

Thorn vs. Erlang

- Common features
  - Similar concurrency & messaging model
  - Thorn has RPC and priority built in.
- Thorn is a much more conventional language
  - **Semantics:** Vars, classes, data structures.
  - **Syntax:** Distinction between 'function' and 'process'
- Thorn is honest about mutability
  - Typical Erlang processes hold state in parameters

Authorization Server

```
import CRYPT.*;

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

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

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

Auth server in Erlang

```
-module(au).
-export([startDB/0, encrypt/1, named/1, add/2, confirm
    /2, au/0, test/0]).

-include_lib("stdlib/include/qlc.hrl").

-record(authrec, {name, pw, fails}).

encrypt(PW) -> {crypt, PW}.

startDB() ->
    mnesia:create_schema([node()]),
    mnesia:start(),
    mnesia:create_table(authrec, [{attributes,
     record_info(fields, authrec)}]).

do(Q) ->
    F = fun () -> qlc:e(Q) end,
    {atomic, Val} = mnesia:transaction(F),
    Val.


named(Name) ->
    do(qlc:q([E || E <- mnesia:table(authrec),
     E#authrec.name == Name])).

add(Name, CryptPW) ->
    InThereNow = named(Name),
    if InThereNow == [] ->
            New = #authrec{name=Name, pw=CryptPW,
     fails=0},
            F = fun ()  -> mnesia:write(New) end,
            mnesia:transaction(F),
            true;
```

```
confirm(Name, ClearPW) ->
    TryPW = encrypt(ClearPW),
    InThereNow = named(Name),
    if
        InThereNow == [] -> false;
        true ->
            [#authrec{name=Name,pw=RealPW,fails=Fails}]
     = InThereNow,
            if
                RealPW == TryPW ->
                    true;
                true ->
                    New = #authrec{name=Name,
     pw=RealPW, fails = Fails+1},
                    F = fun() -> mnesia:write(New) end,
                    mnesia:transaction(F),
                    false
            end
    end.

% We really should use nonces or other reliable RPC
     stuff, but not now.

au() ->
    receive
        {register, Name, CryptedPW, From} ->
            From ! add(Name, CryptedPW),
            au();
        {confirm, Name, ClearPW, From} ->
            From ! confirm(Name, ClearPW),
            au();
        quit -> false;
        X  -> io:format("au: cryptic ~w~n", [X]),
               au()
```

Syntactic Influences & Principles

- C family

  if (a==b) c(); else d();

  module AUTH { ... }

- ML family

  [1,2,3]

  patterns

- Short distinctive keywords and symbols

  fun -- define a function

  %[...] -- list comprehension

- Goal: reveal what's going on

  := vs. =

  (Important for untyped language)