Thorn: From Scripting to Robust Concurrent Components

IBM Research
Bard Bloom
John Field
Nate Nystrom

Purdue
Brian Burg
Johan Östlund
Gregor Richards
Jan Vitek
Tobias Wrigstad

Cambridge
Rok Strniša

IBM
Purdue University

Distributed programming today: an AJAX web app

Form + JScript Code

City
ZIP code
Credit Card Number
Submit

Zip Database
Merchant Credit Server
User Credit Servers

Zip Lookup Servlet
Form Submission Servlet
AJAX code snippet

```javascript
// Use the XML DOM to unpack the city and state data
var city = xmlDoc.getElementsByTagName('city').item(0).firstChild.data;
var state = xmlDoc.getElementsByTagName('state').item(0).firstChild.data;
```

A prettier picture: app composed from encapsulated, distributed components

---

```
FormController
```

```
InputWidget (city)
```

```
ButtonWidget (next)
```

```
SessionController
```

```
ZipController
```

```
DB
```

```
Acct (merchant)
```

```
Acct (user)
```

NB: concurrency is ubiquitous

- UI events
- client-server interaction
- data parallelism
- task parallelism
Thorn goals

An agile, high performance language for distributed applications (including web apps), reactive systems, and concurrent servers, with strong support for:

- **Concurrency**: for application scalability, real-world event handling
- **Distribution**: distributed computing is ubiquitous, but existing language support is poor
- **Code evolution**: scripting languages are justifiably popular, but don’t scale well to robust, maintainable systems
- **Security**: need to build support for data/code confidentiality/privacy into the language runtime, particularly in a distributed environment
- **Fault-tolerance**: provide features that help programmers write robust code in the presence of hardware/software faults
- **JVM implementation + Java interoperability**: build on efficient JVM platforms and Java libraries

Thorn is a scripting language

```
access command-line args

file I/O methods

split string into string list

for (l <= argv()().file().contents().split("\n"))
  if (l.contains?(argv()[1])) println(l);

iterate over elements of a list

no explicit decl needed for var

usual library functions on lists
```
fun pang(name) = spawn {
  var other;
  async volley(n) {
    if (n == 0)
      println("$name misses");
    else {
      other <-- volley(n-1);
      println("round $n: $name hits the ball.");
    }
  }
  volley sync playWith(other') { other := other'; }
  body { while (true) serve; }
} spawn;

ping = pang("ping"); pong = pang("pong");
ping <-> playWith(pong); pong <-> playWith(ping);
ping <-- volley(10);

---

Thorn is a concurrent language

create a new component (process)

isolated, mutable component state

unidirectional communication

unidirectional msg send

bidirectional communication

bidirectional msg send (RPC)

component control loop

---

Ping pong process structure

parent

spawn

volley(10)

playWith(ping)

volley(8)

spawn

ping

playWith(ping)

volley(9)

pong

volley(9)

playWith(pong)
Scripting + Concurrency:
? …or…!

- Scripts already handle concurrency (but not especially well)
- Dynamic typing allows code for distributed components to evolve independently…code can bend without breaking
- Rich collection of built-in datatypes allows components with minimal advance knowledge of one another's information schemas to communicate readily
- Powerful aggregate datatypes extremely handy for managing component state
  - associative datatypes allow distinct components to maintain differing “views” of same logical data

Thorn key features

- Concurrency & distribution
  - applications organized as collection of single-threaded processes
- Powerful core scripting language
  - patterns, queries, tables,
- Object system
  - class-based
  - multiple (but simple) inheritance
  - promotes (but doesn’t require) immutability
- Module system
  - packaging and name scoping mechanism
  - no dynamic class loading or complex class loading semantics
- Optional type annotations
  - to enable static checking
  - for code optimization
- Java interoperability
- Compiler organized as collection of plugins
  - allows modular implementation
  - allows extensibility
Thorn design philosophy

- Steal good ideas from everywhere
  - (ok, we invented some too)
  - aiming for harmonious merge of features
  - strongest influences: Erlang, Python (but there are many others)
- Adopt best ideas from scripting world
  - dynamically-typed core language
  - but no reflective or “self-modifying” features
- Assume concurrency is ubiquitous
- Seduce programmers to good software engineering
  - powerful constructs that provide immediate value
  - optional features for robustness

Project status

- Interpreter for language design prototyping and validation
- JVM compiler for most of core language
  - no sophisticated optimizations
  - performance comparable to Python
  - compiler plugin support
- Initial prototype of (optional) type annotation system
- Planned open source release for research partners, early beta users soon
Rest of the talk: a walk through Thorn

- Scripting core
  - patterns
  - tables and queries
- Concurrency
- Modules
- Objects and classes
- Cheeper: microTwitter in Thorn

- Not covered today
  - compiler details, including plugin mechanism
  - type system
  - many details
- Disclaimers:
  - a research project, not an IBM product
  - no time to explain how Thorn feature F relates to feature F' in your favorite language L
  - some features of language subject to change as experience base grows

Why scripts?

- **Purposes:**
  - to quickly toss together useful little gadgets
  - e.g., 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...
  – …e.g., Sweden’s pension system (written in Perl!)

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
From scripts to programs via *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
  - $f > 0$; $l$ has length 3
- Entice programmers into wanting to supply such assertions
  - make them useful for programming
  - not just verification and good practice

**Thorn patterns**

- Patterns explain what a programmer expects

```haskell
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 also use this information for optimization
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 elt. list with head=1
  ```

- **pat = Exp:** immutable binding
  
  ```
  z = 1;  # introduce new var z, bound to 1
  ```

- **match(Exp) { Pat1 ... Patn ... }:** match stmt

- **receive stmt**

Patterns in code

```len=19
match value of k
bind 2nd to y
idiom for "I found it, and it's y!"

alist = [ [1, true], [15, null], ["yes", "no"] ];

fun lookup(k, [[$1, v], _,...]) = +v:
  | lookup(k, []) = null;
  | lookup(k, [ , t...]) = lookup(k, t);

if (lookup(15, alist) ~ +w) {
  assert(w == null) ;
} else assert(false) ;

if (lookup("no", alist) ~ +w) assert(false) ;
else assert(true) ;
```
Other patterns

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

Tables and maps

- Table: Thorn’s big mutable data structure
  - one or more keys
  - one or more non-keys
  - akin to maps and database tables
- Word-counting script:
  \[
  t = \text{table(word)}\{\text{var } n;\}
  \]
  \[
  t.\text{ins}( \{ : \text{word:“provenance”, n: 1 :} \} );
  \]
  \[
  t(“provenance”).n
  \]
- 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
  \[
  t = \text{table(word)}\{\text{var } n, \text{where;}\};
  \]
Queries

- Special syntax for common cases of searching and constructing
- List comprehensions:

  \[
  \{ i*i | \text{for } i <- 2 .. 4 \} = [4, 9, 16]
  \]
  \[
  \{ i*i | \text{for } i <- 2 .. 4, \text{if prime?}(i) \} = [4, 9]
  \]

- Quantifiers:

  ```
  fun prime?(n) =
  ! %some(n mod k == 0 |
  for k <- 2 .. n, while k*k <= n);
  ```

Table queries

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

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

- Pattern matching!
- Results of query
Thorn concurrency model

- All state encapsulated in a component
- Each component has a single thread of control
- Components communicate by asynchronous message-passing
- Messages passed by value
- Messages managed via a simple "mailbox" queue
- No state shared among components
- Faults do not propagate across components
- Based on Actor model [Hewitt et al.]
- No locks

Components and concurrency

```javascript
component LifeWorker {
    var region;
    async workOn(r) {region := r;
    sync boundary(direction, cells)
    body {...} # code to run Conway's life
}

regions = /* compute regions */
for (r <- regions) {
    c = spawn(LifeWorker);
    c <-- workOn(r);
}
```

- isolated lightweight process (here, with a name)
- (mutable) component state
- communication: "access point" for peer; async does not reply
- sync communication replies
- body code is run when component is created
- create a component instance
- initialize the component using async message
comp <-> m(x) timeout(n) { dealWithIt(); }

spawn {
    var done := false;
    async quit() prio 100 { done := true; }
    sync do_something_real() { ... }
    body { while (!done) serve; }
}

Fine points

- optional timeout block for sync communications
- optional communication priority
- a single communication is processed each time the body executes serve

Typical concurrency pattern

spawn {
    sync findIt(aKey) {
        logger <-- someoneSought(sender, aKey);
        # ... code to look it up ...
        return theAnswer;
    }
    body { while (true) serve; }
}

logger = spawn {
    var log := [];
    async someoneSought(who, what) {
        # do not answer; just cons onto log
        log ::= {: who, what :};
    }
    body { while (true) serve; }
}
Low-Level Communication

```plaintext
receive {
    {: stop_right_now: _ :} prio 1 => { return; }
| {: please: "post", data: x :} => { do_post(x); }
| {: please: "scan", want: p :} => { do_scan(p); }
| timeout(10000) => { bored := true; }
}
```

- Asynchronously sends v (any value) to c's mailbox
- Highest priority messages always matched first
- Optional (but usually necessary) timeout block

Pure values and marshalling

- In Thorn, only pure values may be passed as messages
  - Primitive values
  - Records, lists of pure values
  - Instances of pure classes
- Pure classes:
  - All fields are val, initialized to pure values
  - All methods are pure
- Pure methods/functions
  - No free references to global names
  - Other free references only to pure values
- Pure values passed as messages among components in the same virtual machine can be shared
- Functional values are not actually marshalled; sending and receiving components must load code from the same module
Modules

- Designed for allowing existing scripts to be repackaged as reusable code

```
module M;
import N;
import own S = 0;
class A extends B {};
n = A();
private var x = N.A();
public S.C;
# public N.A;
```

- this module is named M
- shared instance of N
- own instance of O (renamed S)
- B must come from M xor N xor S
- A is M.A
- x is not exported from M
- S.C not exported by default
- Error: M.A already exported

Objects in Thorn

- **Class-based**
  - less flexible (dangerous) than Lua, Self, JavaScript, …
  - more robustifiable
- **All access to data fields mediated by getter/setter methods**
  - only declaring class can access fields directly
  - as in Smalltalk
- **Parameterized classes allow pattern matching on objects**
- **Multiple inheritance**
  - method ambiguities must be explicitly resolved
  - can use to model most interface examples in Java…
  - …or mixins
- **Various safety and convenience features**
Classes

```scala
class Named {
  val theName;
  def name() = theName;
  new Named(name'){
    theName = name';
  }
}Named
kim = Named("Kim");
```

- `val`: read-only (the default)
- `var`: read-write
- `def`: denotes a constructor
- `new`: one-time binding to val field
- `this can't escape ctor (no access to unitialized fields)
- `simple ctor invocation (no 'new')`

Parameterized classes

```scala
class Point(x,y)
class NamedPoint(x,y,name)
  extends Point(x,y), Named(name)
np = NamedPoint(0,0,"Origin");
```

- `x and y are: (1) public val fields; (2) params of implicit ctor; (3) more...
- `NamedPoint's x and y are Point's x and y.`
Multiple inheritance

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

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

ambiguous method references must be explicitly disambiguated

Classes and patterns

- Classes define **extractor** patterns:
  ```scala
class Named(name){...}
if (person ~ Named(n)) { print("Name is $n"); }
if (person ~ Named("Kim")) { print("Hi, Kim."); }
```
Accessing fields

class A {
    var b;  # implied getter: def b() = b;
    # implied setter: def 'b':='(b2){b:=b2;}

    var c;  # implied getter: def c() = c;
    def 'c':=(v) { if (v.prime?) c := v; }

    val d=1;  # implied getter: def d() = d;

    var secret;
    def secret() { throw "Please don’t"; }
    def 'secret':=(x) { throw "Please don’t"; }
}

anA = A();
x = anA.d  # implicitly invokes anA.d()

Cheeper: microTwitter in Thorn
Influences

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

Experience

- Compiler boostrapped in Thorn itself
- Various medium-sized apps
  - scripting “shootout” benchmarks
  - internet relay chat application
- Larger apps in progress
To do

Work in progress
• failure recovery for components
  – via persistent state
• component-level security
  – information flow
  – access control
• fancier types
• new, optimizing compiler
• open source release

Planned
• web frameworks
• cloud frameworks
• parameterized modules
• join-style patterns for synchronization
• database integration
• system-level optimizations
  – (e.g., message traffic minimization)
• more advanced type systems and static checkers
• Eclipse plugin

Wrapup

• Concurrency is everywhere
• Scripting + concurrency = power!
• Patterns, modules, classes all work together to help make scripts robust
For more information…

• [http://www.thorn-lang.org](http://www.thorn-lang.org)
  – links to documentation
  – online interpreter demo coming soon

• Upcoming papers:
  – OOPSLA ’09 (language design)
  – POPL ’10 (optional type system, v1)

Thanks! Questions?
Backup Material

Thorn application domains

**Targeted**
- Networked software services
- Reactive embedded applications
- Event-driven and task-oriented server applications
- Client and server code for mobile apps
- Client and server code for web apps

**Not targeted**
- Data parallel apps
- Scientific apps
- Extreme throughput
- Embedded code with device-level control
Application development landscape

- Many devices
  - cell phones, GPS receivers, PDAs
  - embedded systems (automotive, aircraft, home appliances)
  - sensors / actuators / webcams

- Many servers/services in the "cloud"
  - compute services
  - data services
  - network appliances

- Systems software and embedded software must work together
  - server support for embedded devices
  - embedded devices usually networked (sensors, transport sense/control)

- Web programming and non-web distributed programming more and more alike
  - AJAX apps are lightweight concurrent "servers"
  - RESTful style being adopted for software services not connected to a browser

How do we enable programmers to build and compose agile software in such an environment?

Do we really need another programming language?

- Distribution, concurrency, and security are at best afterthoughts in current mainstream languages
  - addressing these issues entirely through libraries is complex, prone to obscure errors, and significantly inhibits high-level optimization

- Attempting to bolt significant new features on existing languages is likely to yield diminishing returns
  - concurrency constructs interact with other languages features in surprisingly subtle ways

- Scripting languages are a fertile area for innovation; programmers are willing to experiment with new approaches
Fancier queries

words = novel.split("[^A-Za-z']+");

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

sorted = %sort(r => n => word |
    for r && {: n, word :} <- counts);

for (r <- sorted) {
    println(r);
}

Tables and queries: more

bio = table{name,day}{map var weight; val bp; val hair; };

bio("kim",1) := {: bp: 120, hair: "black", weight: 120 :};

bio["kim", 1] := 130;
assert(bio["kim", 1] == 130) ;

bio.ins({: name:"kim", day: 4, hair: "black", weight: 132, bp: 125 :});

d = %find( day |
    for {: name: "kim", day, hair:"blue" ;} <- bio );
assert(d == 8) ;

bio("kim", d) := null;
assert( %every(hair == "black" | for {: name:"kim", hair :} <- bio) );
Patterns and bindings

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

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

match empty list
match list with head x and tail y
does it match? if so, bind x,y in then clause

Pattern variable scope

● 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 ~ +g);
match non-null, bind to g
g out of scope
g in scope
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
- **Java (and many other languages) express this chaotically:**
  - variously return null, or -1, or throw exception. …
  - sometimes two related functions are used, e.g.
    \(\text{Map.containsKey}(k), \text{Map.get}(k)\)
- **ML’s Some(e) and None are pleasant**
  - but can require extra boxing
- **In Thorn, \(+x == x\) for most x’s**
  - extra benefit: quick to compute
- **\(+null != null\)**
  - \(+null\) is an otherwise boring value
- **Nullities: \(+null, ++null, +++null, etc.\)**
  - this (relatively rare) case requires boxing

---

**Journaler: mini-blog**

```javascript
journaler = spawn{
  journals = table(user, number) {var entry, comments;};
  sync newUser(name) {
    if ( %some(true |
      for {: user:$\{name\} } <- journals) ) {
      return false; # name taken
    }
    else {
      journals(name, 0) := {: entry: "Started", comments: [] :};
    }
  }newUser

  sync getEntry(user, number) {
    if (journals(user,number) ~ +{:entry, comments:}) return +entry;
    else return null;
  }

  body { while(true) serve; }
}spawn;
```
Talking to Journaler

```plaintext
var i := 0;
var username;
do {
i += 1;
    username := "bard"+i;
} until (journaler <-> newUser(username));

if ((journaler<->getEntry(username, 0)) ~ +entry) {
    # yay, I've got an entry.
}
```