DEMO
Thorn

Robust, Concurrent, Extensible Scripting on the JVM

Bard Bloom, John Field (IBM)
Nathaniel Nystrom (U.T. 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
- Provide
  - The general case
  - Some common cases

Thorn Features

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

Example One: Dice Frequencies

- The problem: given m,k,n
  - roll n k-sided dice m times;
  - graph the results

```
th -f dice.th -- 30 2 6
 2 *
 3
 4 ***
 5 ****
 6 ******
 7 ****
 8 ******
 9 ***
10 **
11 *
12
```

Example One: Code

```
[.int(nRolls), .int(nDice), .int(nSides) ] = argv();

fun roll() =
 %[nSides.rand1 | for i <- 1 .. nDice].sum;

stars = %group(t = roll())
 {s = %list "*"; | for i<- 1 .. nRolls};

for(i <- nDice .. nDice * nSides) {
 println("%3d ".format(i) +
 (s.cat if stars(i) ~ {:s:} else ""));
}
```

There's a lot going on here....

Pattern Matching

```
[.int(nRolls), .int(nSides), .int(nDice) ] = argv();
```

- `argv()` returns a list of strings, `["30", "6", "2"]`
- α `= x;` matches the value `x` against the pattern α.
  - Success: binds variables in α
  - Failure: throws exception
- `[`β`,`γ`,`δ`] = [b,c,d];` matches a three-element list
  - Failure if RHS not a 3-element list
  - `[`β`...,`γ`,`δ`...] = [l,m,n,o,p];`
- The pattern `.int(`ε`)` matches `b` if
  - ε matches non-null `b.int()`
  - `int` is a method on strings, returning number or `null`.
  - Nullary method calls don't need `()` ---- `b.int == b.int()`
- The pattern `nRolls` matches anything and binds to it

Functions, Lists, Queries

```
fun roll() =
 %[nSides.rand1 | for i <- 1 .. nDice].sum;
```

- Functions can refer to external variables
- The list comprehension `%[Z | for i <- r]`
  - list of the values of `Z` varying `i`.
  - This one makes a list of random numbers
  - `nSides.rand1` is a random number 1 to `nSides`.
- `.sum` method on lists

Group query

```
stars = %group(t = roll())
 {s = %list "*"; | for i<- 1 .. nRolls};
```

- Construct a table (dictionary, map)
  - Key: `t =` the dieroll
  - Value: `s =` one "*" for each time `t` was rolled.
  - (Could have more keys and values)
  - (Could have other accumulations than `%list`)

Printing the Bar Graph

```
for(i <- nDice .. nDice * nSides) {
 println("%3d ".format(i) +
 (s.cat if stars(i) ~ {:s:} else ""));
}
```

- Ordinary loop over a numeric range
- An ordinary `println`
- C-ish formatting operations.

Full of Stars

```
(s.cat if stars(i) ~ {:s:} else "")
```

- Pythonian if-else expression, but...
- `stars(i)` = table lookup
- `~` = pattern match returning Boolean
  - Unlike '=', no errors
- `{:s:}` pattern for "has an `s` field, bind it to `s`"
  - `s` will be a list of "*"s.
- `s` is available in the true-branch
- `s.cat` concatenates the "*"s in `s`.
- So, this is a string of stars.

Example II: The MMORPG

- **M**inimalist **M**ultiplayer **O**n-line **R**ole-**P**laying **G**ame
- Adverbial Ping-pong.
- You get to describe how you hit the ball:
  - Ping serves the ball *happily*.
  - Pong returns the ball *tiredly*.
  - Ping returns the ball *eagerly*.
  - Pong returns the ball *by bouncing it off her head*.

MMORPG Code

```
spawn ping {
var done := false;
body{
 [name, otherURI] = argv();
 otherSite = site(otherURI);

 fun play(hit) {
   advly = readln("Hit how?");
   done := advly == "";
   if (done) {
     println("You lose!");
     otherSite <<< null;
   }
   else {
     otherSite <<<
"$name $`hit`s the ball $advly.";
   }
 }play
```

```
start = thisSite().str < otherSite.str;

if (start) play("serve");

do {
 receive{
   msg:string => {
     println(msg);
     play("return");
   }
 | null => {
     println("You win!");
     done := true;
    }
   }receive
 } until(done);

}body
}ping;
```

Thorn Concurrency

- **Component** = process or thread
  - Single strand of control
  - Isolated memory: no shared data structures
    - Encapsulated local state.
  - Communication via messages & RPCs only
  - cf. Actors, Erlang
- Consequences:
  - Locking is never needed
  - No shared-memory bugs, ever.
  - Instead, you get messaging bugs,
    - Not quite as nasty

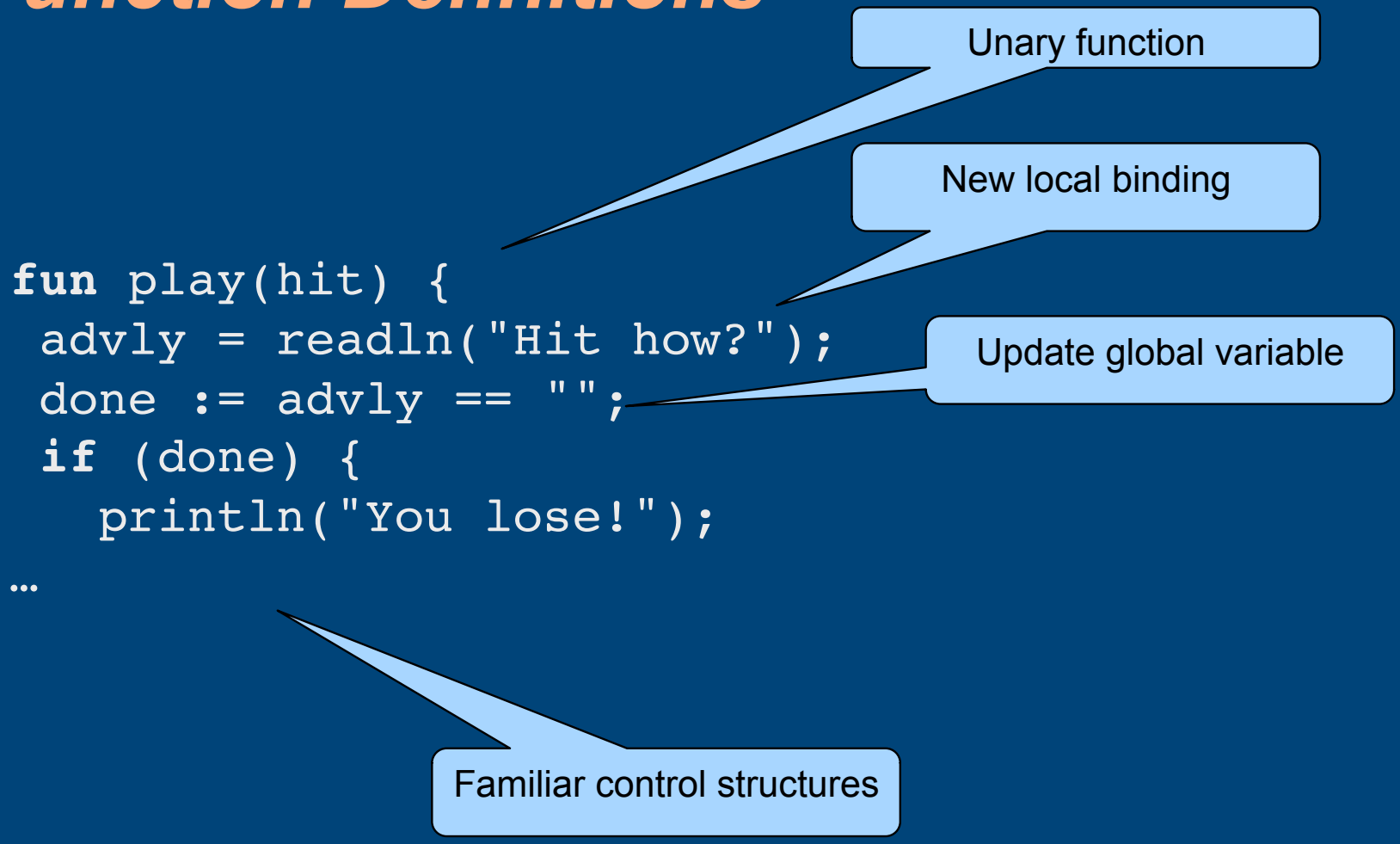
Spawn

- `p = spawn foo { /*...*/ };`
  - Optional Syntax: close-brace can match open brace
  - or `p = spawn foo { /*...*/ }spawn;`
  - or `p = spawn foo { /*...*/ }foo;`
- In the braces go:
  - Variable declarations (local state)

    ```
    .var done := false;
    ```
  - Body clause (code to execute)

    ```
    .body{ /* ... */ }
    ```
  - Various other things

Function Definitions




```
fun play(hit) {
 advly = readln("Hit how?");
 done := advly == "";
 if (done) {
   println("You lose!");
…
```

Sites

- Messages can be sent to a *site:*
  - `site("http://pingpong.somewhere.org:4260")`
- Or to a component running there
  - If you have a reference to it
  - Component references are unforgeable
    - (Unlike sites, which are public)
  - Component references are transmissible

Messages

- Thorn has two communication mechanisms
  - Messages: sending values around
  - RPC: client/server style
- We'll use messages here.

- Sending:

```
otherSite <<< "$name $`hit`s the ball $advly.";
```

- Messages can be
  - Built-in immutable scalars (like strings)
  - Built-in immutable structures (lists, records)
  - User-defined classes, if *pure*

Receive

- Look for incoming message matching a pattern.

```
receive{
  msg:string => {
    println(msg);
    play("return");
  }
| null => {
    println("You win!");
    done := true;
  }
}receive
```

- Optional timeout clause (not shown)
- Unmatched messages stay in inbox

Body of the MMORPG:

```
start = thisSite().str < otherSite.str;
if (start) play("serve");

do {
  receive{
    msg:string => {
      println(msg);
      play("return");
      }
    | null => {
      println("You win!");
      done := true;
      }
  }receive
} until(done);

}body
}ping;
```

Arbitrary choice of 1st player

First player serves

Wait for other player

Other player hit it

Other player missed

Game over

Example III: Cheeper – mini-Twitter in Thorn

- Little client/server program
- Users can
  - ***Chirp*** wise sayings ("cheeps").
    - `Chirp: I like numbers`
  - Vote for or against cheeps by number.
    - `Chirp: +3`
    - `Chirp: -1`
  - Read each others' cheeps (ranked by love).
    - `Chirp: /`
- And of course it's multi-user
- Imagine much more functionality
  - And add it!

Transcript of a run

```
./run-client
Welcome to Cheeper!
? for help
Who are you? Bard
Chirp: I like numbers
You chirped '(0) "I like
  numbers" -- Bard'
Chirp: I like words
You chirped '(1) "I like
  words" -- Bard'
Chirp: I like spices
You chirped '(2) "I like
  spices" -- Bard'
Chirp: /
(0) "I like numbers" --
  Bard [+0/-0]
(1) "I like words" -- Bard
  [+0/-0]
(2) "I like spices" -- Bard
  [+0/-0]
Chirp: +1
Thanks
Chirp: /
(1) "I like words" -- Bard
  [+1/-0]
(0) "I like numbers" --
  Bard [+0/-0]
(2) "I like spices" -- Bard
  [+0/-0]
```

Client-Server Style Communication

- Server defines *communications*:
  - `sync chirp!(text,phil) { ... }`
    - RPC
  - `async stopRightNow() prio 100 {...}`
    - Signal
- Client can call these
  - `response = server <-> chirp!("Hey!", "Phil");`
  - `server <-- stopRightNow();`
  - Timeout option available on `<->`.
- Server has explicit control
  - `serve;` // respond to *one* communication
  - ... timeout / administrative options.

Modules

```
module CHEEPER {
 class Chirp(text, phil, n) :pure{
   def str = '($n) "$text" -- $phil';
   def hashCode = text.hashCode;
 }
}
```

- Modules: sharing of code between components
  - Can encapsulate (component-local) state
  - Based on Java Module proposal

Classes

```
module CHEEPER {
 class Chirp(text, phil, n) :pure{
   def str = '($n) "$text" -- $phil';
   def hashCode = text.hashCode;
 }
}
```

- Classes:
  - Parameters (text, phil, n) give...
    - instance variables of those names
    - constructor
    - pattern match
    - (available a la carte too)
  - pure means "immutable" and "transmissible"
  - Multiple inheritance
- Methods with `def`.

Client Code

```
spawn chclient {
import CHEEPER.*;
server = site(argv()(0));

fun help() {
 println("? = help");
 println("/ = read");
 println("+N = vote for");
 println("-N = vote against");
 println("other = chirp that");
 }

fun read() {
 c's = server <-> read();
 for({:chirp, plus, minus:}  <- c's) {
   println(
     "$chirp [+$plus/-$minus]");
   }
 }
```

```
body{
 println("Welcome to Cheeper!");
 println("? for help");

 phil = readln("Who are you? ");
 while(true) {
   s = readln("Chirp: ");
   match(s) {
     "?" => help()
   | "/" => read()
   | "\\+([0-9]+)" / [.int(n)] =>
     println( server <-> vote(n, true))
   | "\\-([0-9]+)" / [.int(n)] =>
     println(server <-> vote(n, false))
   | _ =>
     println(server <-> chirp!(s,phil))
   }
 }while
}body
}chclient;
```

Client Code: Main Loop

```
body{
/// ... snipped ...
phil = readln("Who are you? ");
while(true) {
  s = readln("Chirp: ");
  match(s) {
    "?" => help()
  // ... snipped ...
  | _ =>
    println(server <-> chirp!(s, phil))
  }
}while
}body
}chclient;
```

Pick first matching clause

local function call

RPC call

Print result of RPC call

Aside: More Pattern Matching

- `"\\+([0-9]+)" / [.int(n)]`
- Regexp matching:
  - `s ~ regexp / [`β`,`γ`,`δ`]`
  - Succeeds:
    - If the string matches the regexp
    - And the first capture group matches β
    - And the second capture group matches γ
    - And the third capture group matches δ
    - And there are precisely three capture groups.
- And `.int(n)` as before binds n to int value

Client Code

```
/// ... redacted ...
 while(true) {
   s = readln("Chirp: ");
   match(s) {
     "?" => help()
   | "/" => read()
   | "\\+([0-9]+)" / [.int(n)] =>
     println(server <-> vote(n, true))
   | "\\-([0-9]+)" / [.int(n)] =>
     println(server <-> vote(n, false))
   | _ =>
     println(server <-> chirp!(s, phil))
   }
 }while
/// ... redacted ...
```

Pick first matching clause

Do RPC, print result

Server Code

```
spawn chserver {
import CHEEPER.*;

phils = table(phil){var chirps;};
chirps = table(n){chirp; var plus,
   minus;};

sync chirp!(text, phil){
 n = chirps.num;
 c = Chirp(text,phil,n);
 chirps(n) := {:
   chirp:c,
   plus:0,
   minus:0 :};
 if (phils.has?(phil))
   phils(phil).chirps ::= c;
 else
   phils(phil) := {: chirps:[c] :};
   "You chirped '$c'";
 }chirp!

fun love({: plus, minus :}) = plus -
   minus;

sync read() =
  %sort[row
    %> love(row)
    %< chirp.n
    | for row && {: chirp :} <- chirps];

sync vote(n, plus?) {
 if (plus?)
   chirps(n).plus += 1;
 else
   chirps(n).minus += 1;
 "Thanks";
 }

body{
 println("Cheeper server here!");
 while(true) {
   println("Server ready...");
   serve;
 }
}body
}chserver;
```

Tables

```
phils = table(phil){var chirps;};
chirps = table(n){chirp; var plus, minus;};
```

- Tables are high-power maps/dictionaries
- One or more keys
  - Of any type with == and hashcode
- One or more values
  - Mutable (**var**) or not.
  - Adding a new row is easy
    - No need for objects or parallel tables
  - tracking what you've seen:
  - `phils = table(phil){var chirps; var seen;};`
- Variatiations: ordered, map-style
- Rows are records ...

Records

- Immutable name-value bindings
  - `r = {: a:1, b:2, c:table(x){y} :}`
- Access via selectors
  - `r.b == 2`
- Access via pattern matching
  - `if (r ~ {: a:1, b:b :}) println(b);`
  - Partial match works
  - Name `b` alone abbreviates `b:b`
  - `if (r ~ {: a:1, b :}) println(b);`
- Non-ASCII alternate syntax
  - `r = ‹ a:1, b:2, c:table(x){y} ›`

Records to Objects

- You can upgrade a record to an object
  - `r = {: a:1, b:2, c:table(x){y} :}`
  - `class Abc(a,b,c) { def aplusb() = a+b; }`
    `r = Abc(1,2, table(x){y});`
- And things still work
  - Access via selectors
    - `r.b == 2`
  - Access via pattern matching
    - `if (r ~ {: a:1, b:b :}) println(b);`
- Plus, you get method calls
    - `r.aplusb() == 3`

Chirping

```
sync chirp!(text, phil){
 n = chirps.num;

 c = Chirp(text,phil,n);

 chirps(n) := {: chirp:c, plus:0, minus:0 :};

 if (phils.has?(phil))
   phils(phil).chirps ::= c;
 else
   phils(phil) := {: chirps:[c] :};

 "You chirped '$c'";
 }chirp!
```

Do this on "server <-> chirp!(t,p)"

unique number for convenience

Constructor call

Insert a row into a table

Modify a field of a row

Return value sent to caller

Queries, redux: sorting

```
fun love({: plus, minus :}) = plus - minus;

sync read() =
 %sort[row
 %> love(row)
 %< chirp.n
 | for row && {: chirp :} <- chirps] ;
```

- Produces a list of rows,
  - Sorted by decreasing love
  - And, given equal love, by index number
- Pattern α && β matches if both match
  - Can be used like ML's as, or other things
  - Binds chirp and row

Call and Response

- Client:

```
r = server <-> vote(4,true)
```

wait
wait
wait

- Client gets "Thanks"

- Server defines

```
sync vote(n, plus?) {
 if (plus?)
   chirps(n).plus += 1;
 else
   chirps(n).minus += 1;
 "Thanks";
 }
```

- Server evals `serve;`
- which calls body of `vote`.
- which returns "Thanks"