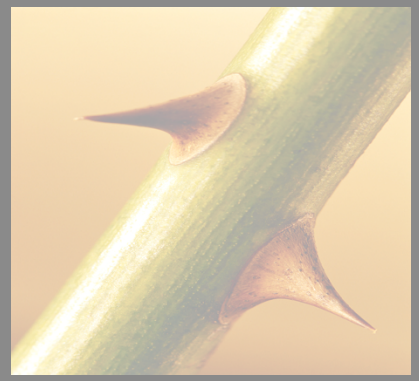
Thorn: Robust Concurrent Scripting



IBM Research
Bard Bloom
Jakob Dam
Julian Dolby
John Field
Emina Torlak

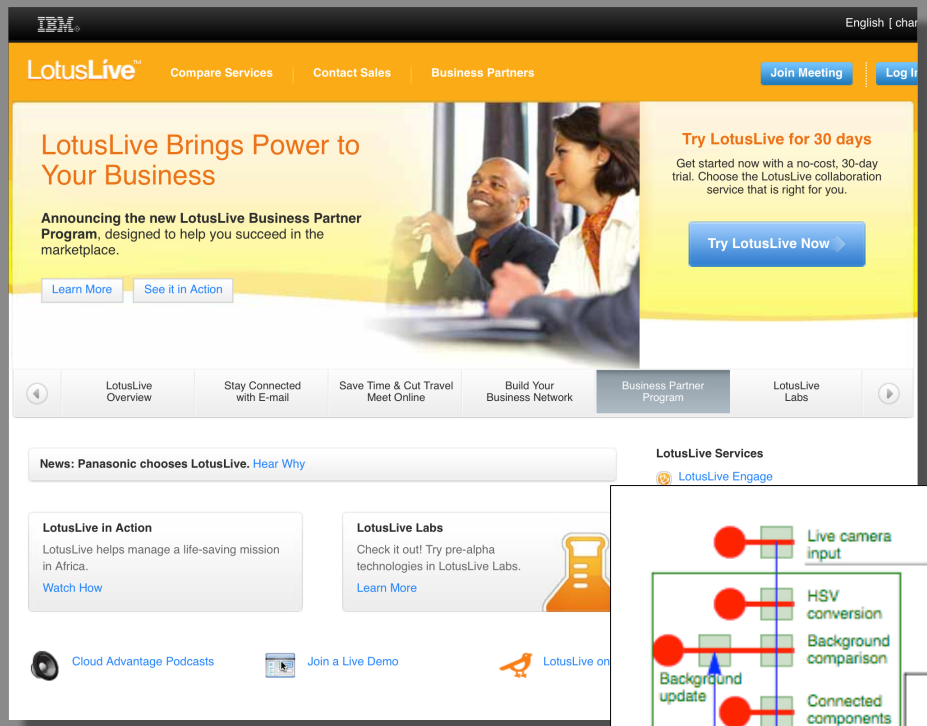
Purdue
Brian Burg
Peta Maj
Gregor Richards
Jan Vitek

Stockholm University
Johan Östlund
Tobias Wrigstad

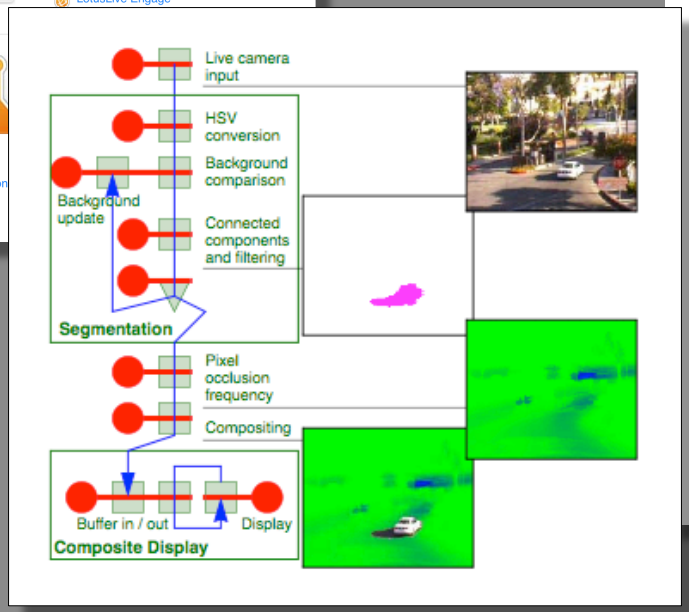
Texas/Lugano
Nate Nystrom

Cambridge
Rok Strniša

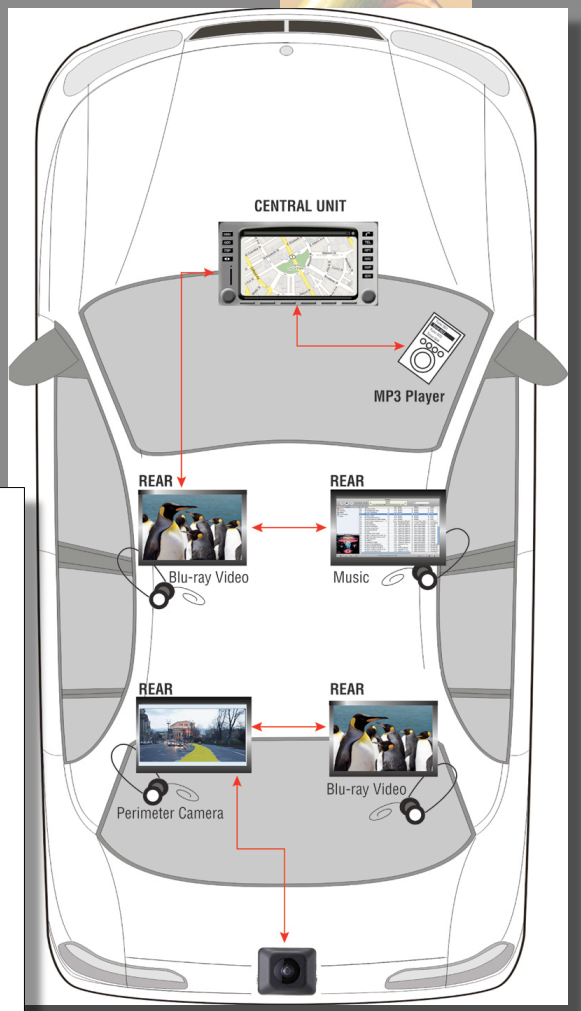
Do these apps have anything in common?



cloud-based web 2.0

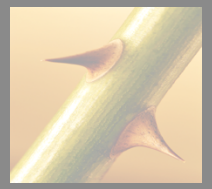


real-time data analysis



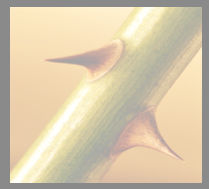
embedded network

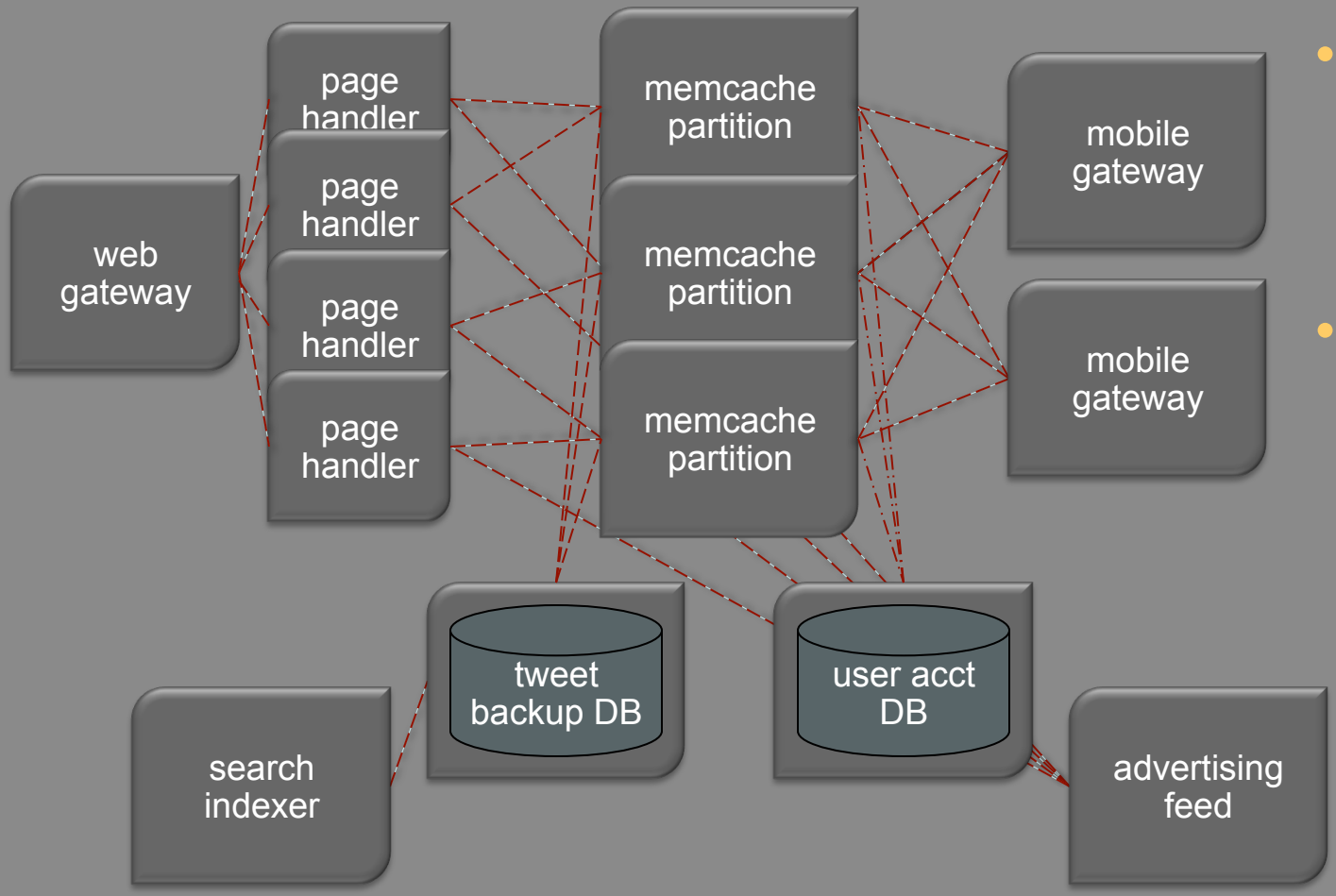
Yes



- Collection of distributed, concurrent components
- Components are loosely coupled by messages, persistent data
- Irregular concurrency, driven by real-world data (“reactive”)
- High data volumes
- Fault-tolerance important

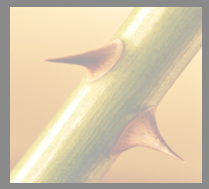
Example: Twitter





- each solid box is a logical process / event handler
- each dashed line is a message

Thorn goals

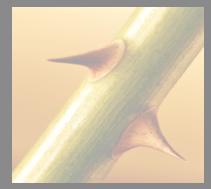


An open source, agile, high performance language for concurrent/distributed applications and reactive systems

Focus areas:

- *Concurrency:* common concurrency model for local and distributed computing
- *Code evolution:* language, runtime, tool support for transition from prototype scripts to robust apps
- *Efficient compilation:* for a dynamic language on a JVM
- *Cloud-level optimizations:* high-level optimizations in a distributed environment
- *Security:* end-to-end security in a distributed setting
- *Fault-tolerance:* provide features that help programmers write robust code in the presence of hardware/software faults

Features, present and absent



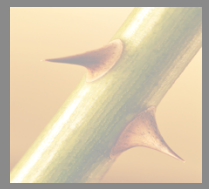
Features

- isolated, concurrent, communicating processes
- lightweight objects
- first-class functions
- explicit state...
- ...but many functional features
- powerful aggregate datatypes
- expressive pattern matching
- dynamic typing
- lightweight module system
- Java interoperability; JVM impl.
- gradual typing system (experimental)

Non-features

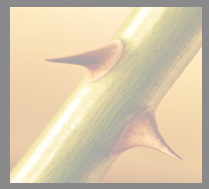
- changing fields/methods of objects on the fly
- introspection/reflection
- serialization of mutable objects/ references or unknown classes
- dynamic code loading

Status



- Open source: http://www.thorn-lang.org
- Interpreter for full language
- JVM compiler for language core
  - performance comparable to Python (with limited optimizations
  - currently being re-engineered
- Initial experience
  - web apps, concurrent kernels, compiler, ...
  - in progress: revisions to syntax, etc. based on experience
- Prototype of (optional) type annotation system

Trivial Thorn script



```
for (l <- argv()(0).file().contents().split("\n"))
  if (l.contains?(argv()(1))) println(l);
```

access command-line args

file i/o methods

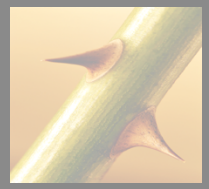
split string into list

iterate over elements of a list

no explicit decl needed for var

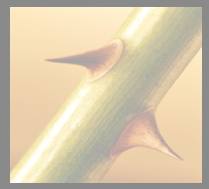
usual library functions on lists

Concurrency in Thorn: a MMORPG*



- Adverbial ping-pong
- Two players
- Play by describing how you hit the ball
- Distributed
- Each player runs exactly the same code

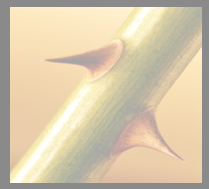
*minimalist multiplayer online role-playing game

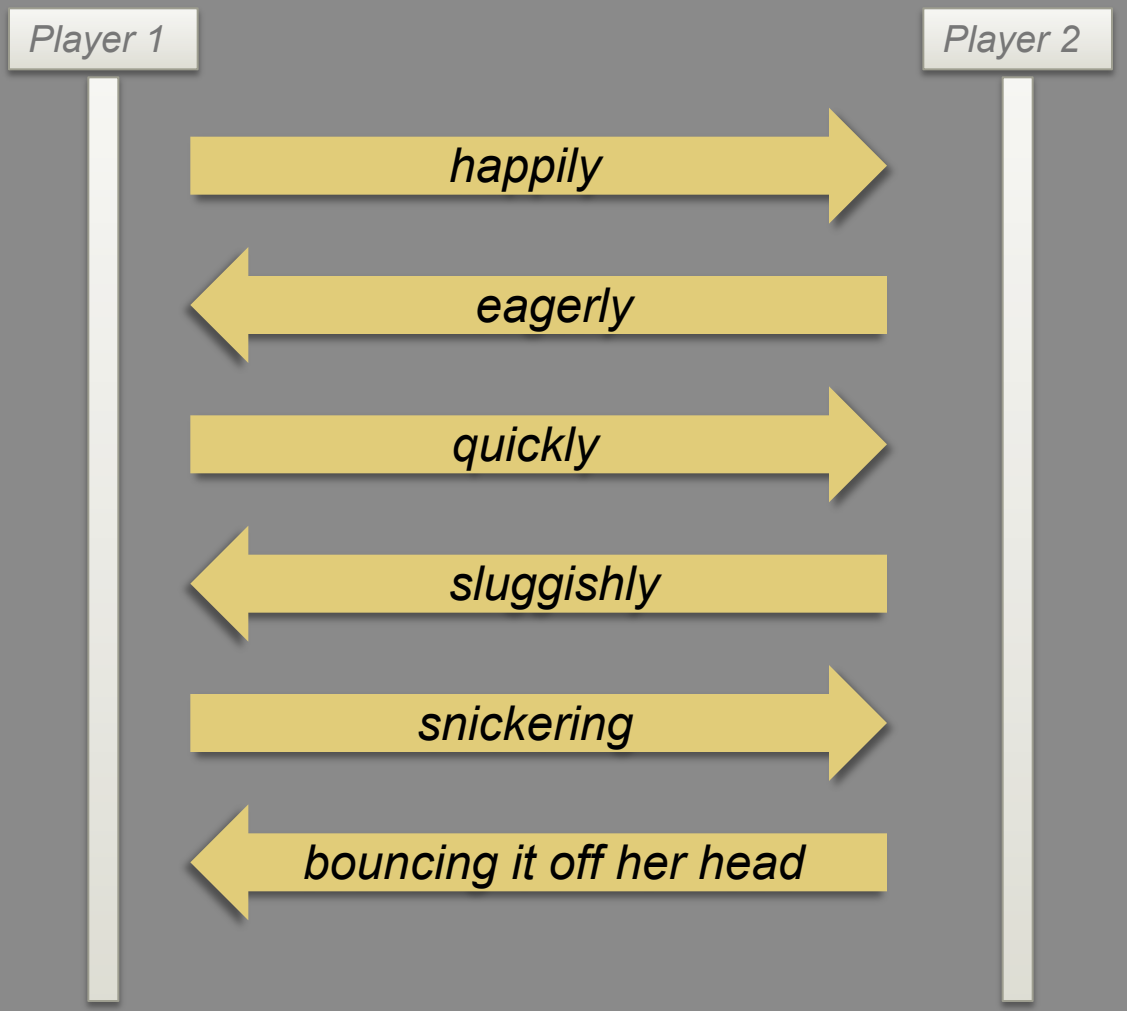


MMORPG

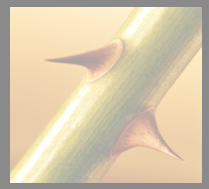
DEMO

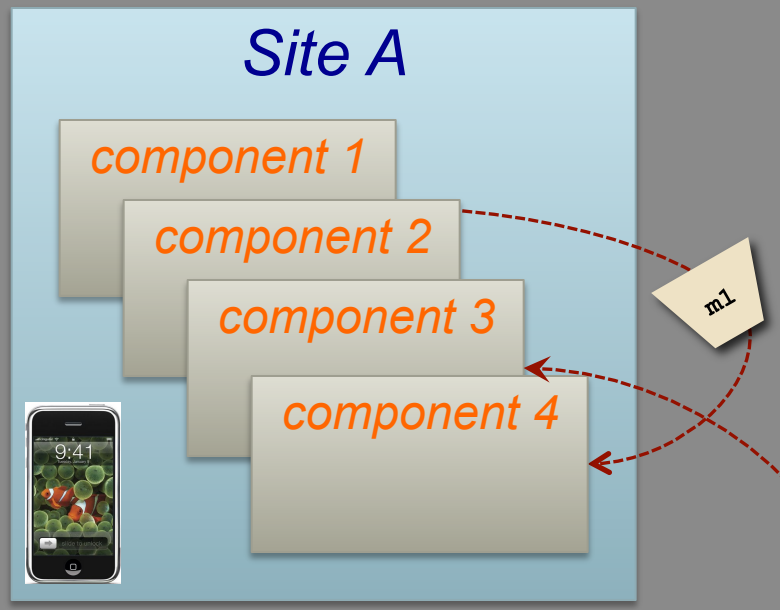
MMORPG message flow





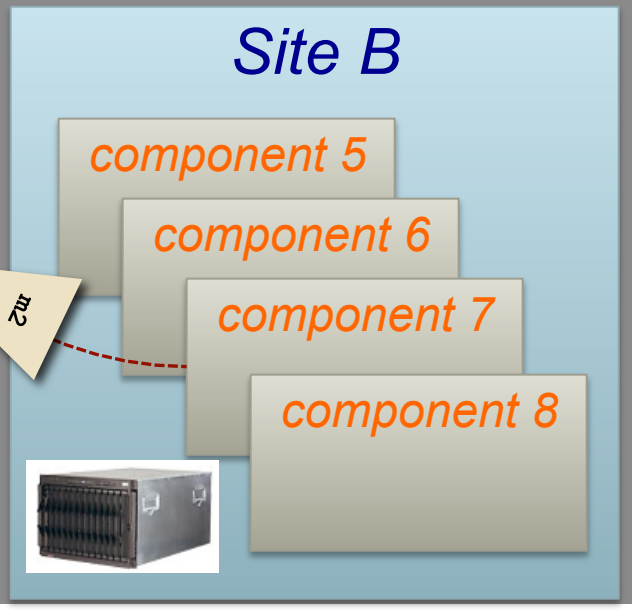
Thorn app: birdseye view





Components are Thorn processes

- components can spawn other components (at the same site)
- processes communicate by message passing
- intra- and inter-site messaging *works the same way*



Sites model physical application distribution

- one JVM per site
- I/O and other resources managed by sites
- failures managed by sites

MMORPG code

```
// MMORPG code for both player

spawn {

  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.";
      }
    }

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

    if (start) play("serve");

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

spawn an isolated *component* (process)

mutable component-scoped variable

immutable component-scoped variable

convert URI into component ref

function decl

receive messages matching *pattern*

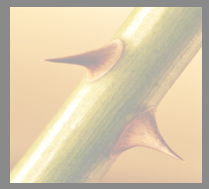
send a message (any immutable datum)

pattern variable (with type constraint)

constant pattern

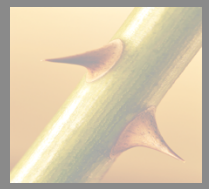
interpolate data into string

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)
- Assume concurrency is ubiquitous
  - this affects *every* aspect of the language design
- Adopt best ideas from scripting world...
  - dynamic typing, powerful aggregates, ...
- ...but seduce programmers to good software engineering
  - powerful constructs that provide immediate value
  - optional features for robustness
  - encourage use of functional features when appropriate
  - no reflective or self-modifying constructs
- Syntax follows semantics
  - more consequential ops have heavier syntax

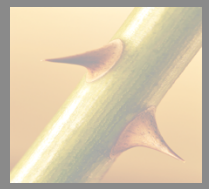
Why the trend toward *dynamic* languages?



- Programming is not the art of implementing a spec, it's the art of *refining* a (usually informal) design
- Want to *defer* non-critical decisions while exploring design space
- *Test* consequences of decisions by running some code
- In the real world, design space typically explored bidirectionally
  - top-down refinement of code architecture, global invariants, shared types
  - bottom-up testing of concrete cases
- *Bugs* are ever-present, but should not manifest themselves so early that they get in the way of refinement process

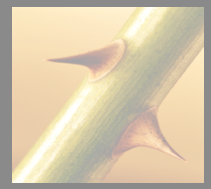
Forcing programmers to document design decisions too early can inhibit productivity

Scripting + concurrency: ? …or… !



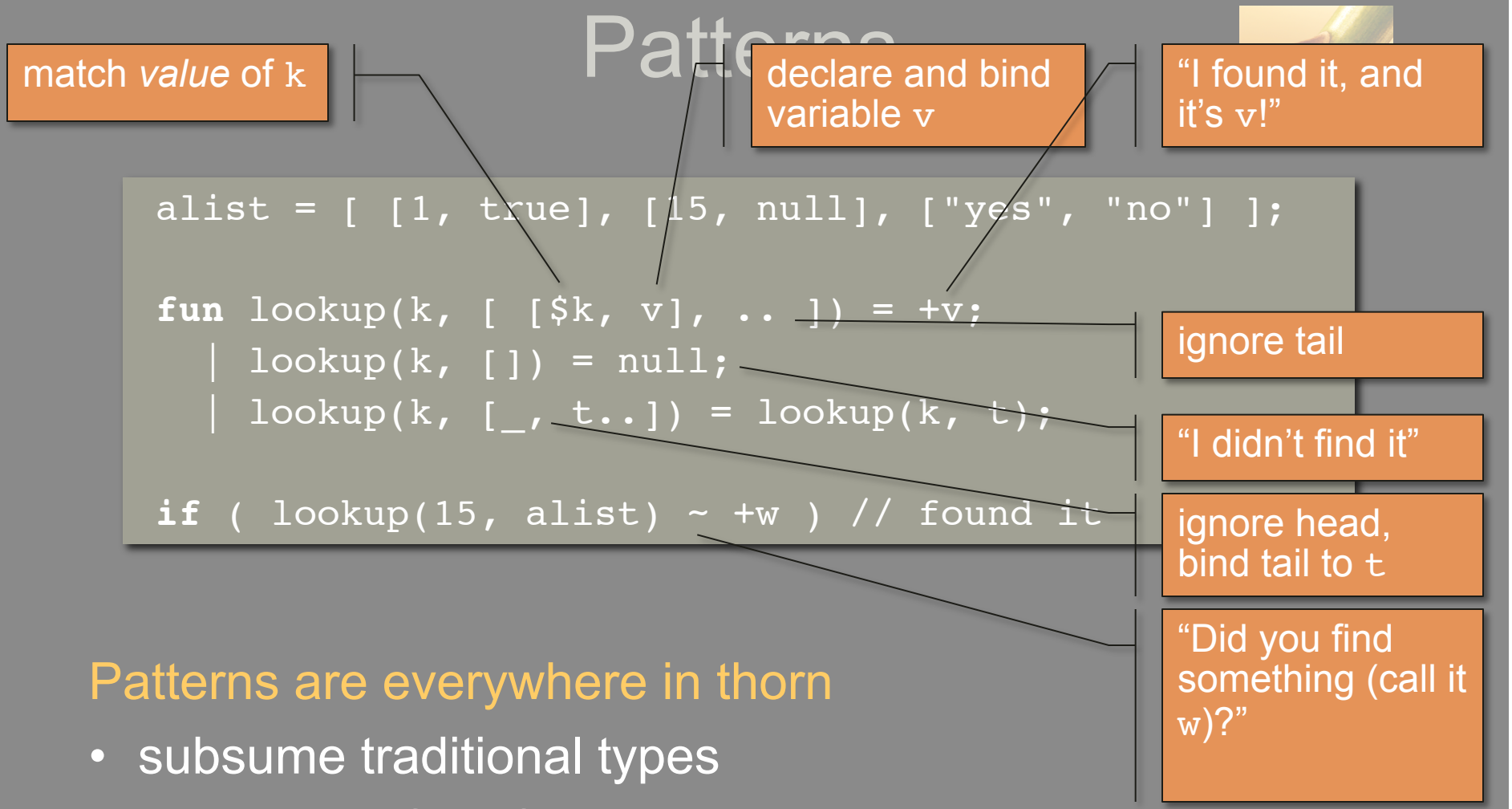
- Scripts already handle concurrency (but not especially well)
- Dynamic typing allows code for distributed components to evolve independently…code can bend without breaking
- Rich collection of built-in datatypes allows components with minimal advance knowledge of one another's information schemas to 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 Robustness features



- No reflection, eval, dynamic code loading (à la Java)
  - alternatives for most scenarios
- Ubiquitous patterns
  - for documentation
  - to generate efficient code
- Powerful aggregates
  - allow semantics-aware optimizations
- Easy upgrade path from simple scripts to reusable code
  - e.g., simple records → encapsulated classes
- Channel-style concurrency
  - to document protocols
- Modules
  - easy to wrap scripts, hide names
- Experimental gradual typing system

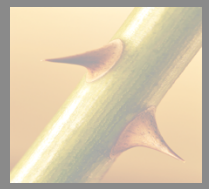
Patterns



Patterns are everywhere in thorn

- subsume traditional types
- provide useful information on intent to compiler
- can be weakened/strengthened as needed

Exposing data: records



- Immutable name-value bindings

```
r = { a:1, b:2, c:[17, 18] }
```

- Access via selectors

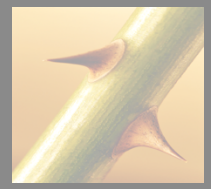
```
r.b == 2
```

- Access via pattern matching

```
if (r ~ { a:1, c }) println(c);
```

  - partial match works
  - `c` alone abbreviates `c:c`

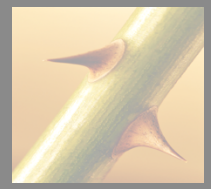
Encapsulating data: classes



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

- class parameters `(text, user, n)` give:
  - instance variables of those names
  - constructor
  - pattern match
  - getters (and setters if mutable)
  - pure means "immutable" and "transmissible"
- multiple inheritance

Records to objects



- Prototype with records

```
r = { a:1, b:2 }
```

- Upgrade later to classes

```
class Abc(a,b) { def aplusb() = a + b };
  ...
r = Abc(1, 2);
```

- And things still work
  - access via selectors

```
r.b == 2
```

  - access via pattern matching

```
if (r ~ { b }) println(b);
```

- Plus, you get method calls

```
r.aplusb() == 3
```

Tables

key field

value fields

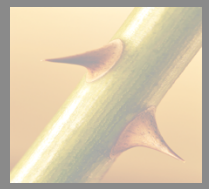
`var`: conveniently update one field "in place"

```
chirps = table(num){chirp, var plus, var minus};
  ...
chirps(n) := { chirp:c, plus:p, minus:m }
```

update row with key `n` (other ops check if row already exists)

- Tables are high power maps/dictionaries
- Each row of a table is a record
- Always mutable: can add/delete rows
- Adding a new column is easy; no need for objects or parallel tables
- Variants: *ordered* (extensible arrays), *map*-style
- Wide selection of *queries*

Tables and queries



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

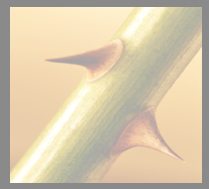
```
th -f dice.th -- 30 2 6

     2 *
     3
     4 ***
     5 ****
     6 ******
     7 ****
     8 ******
     9 ***
    10 **
    11 *
    12
```

Dice co

non-trivial pattern

list query

constructing a table using a query

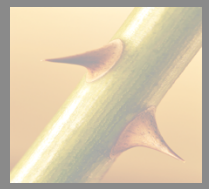


```
[.int(nRolls), .int(nDice), .int(nSides) ] = argv();

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

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

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

explicit loop over a range

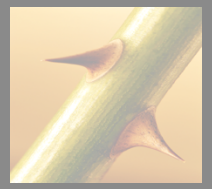
There's a lot going on here....

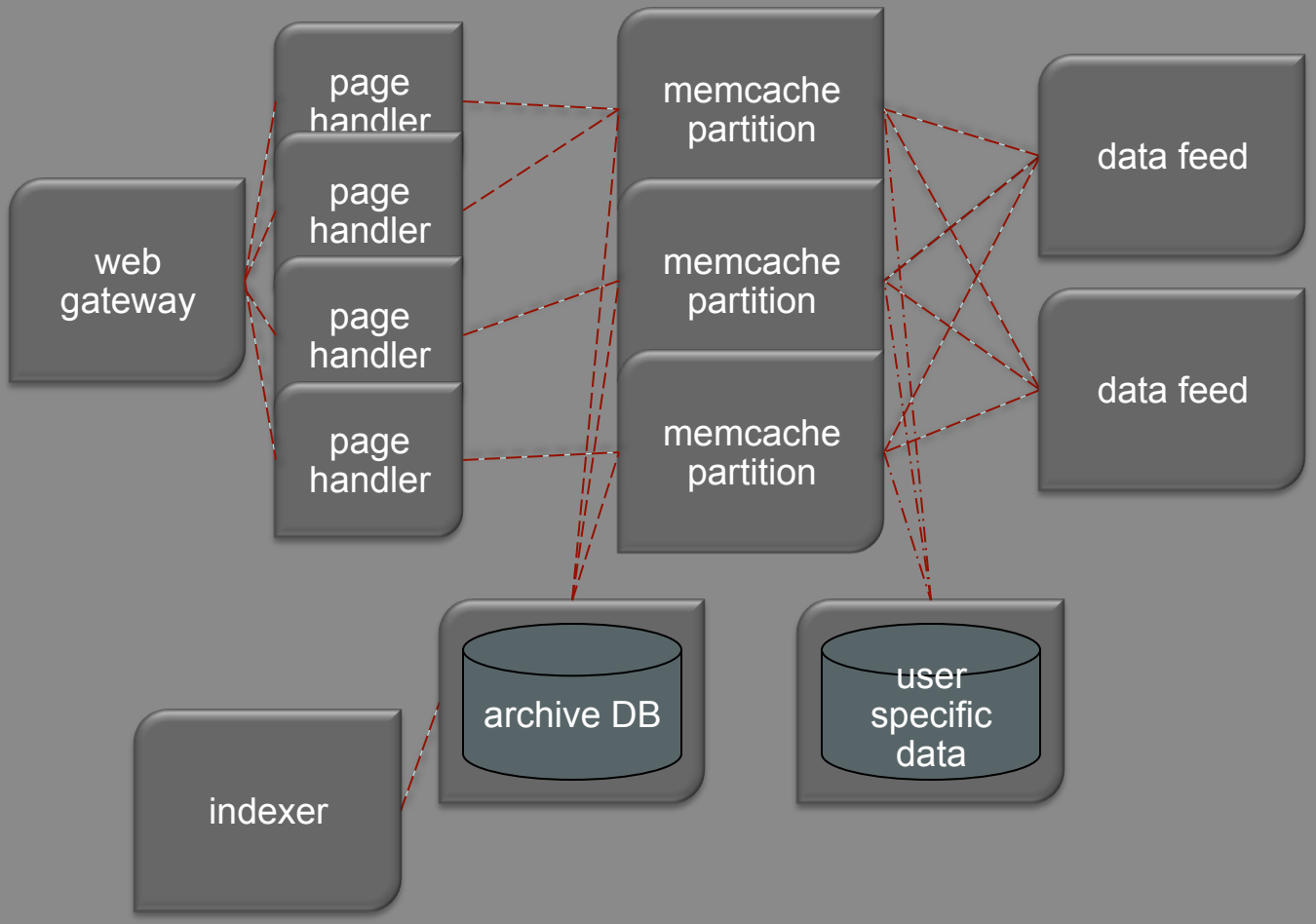
Potential relevance to scientific community



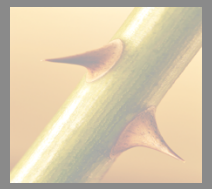
- Substrate for building scalable, domain-specific libraries
  - auto-scaling on cloud platforms
  - adaptive algorithms which require dynamic process creation
  - federated query on multiple data sources
  - take advantage of fault tolerance substrate (e.g., for generalizations of Hadoop)
- Orchestrating wide area computations
  - access to multiple remote data repositories
  - efficient serialization
  - near-real time data analysis of remote feeds
  - coordinating work of loosely coupled research groups
- Security
  - provenance tracking
  - access control
- Robustness
  - patterns, modules, tables/queries, ...
- JVM substrate
  - access to Java libraries
  - portable

Real-time data analysis: not that different from Twitter?



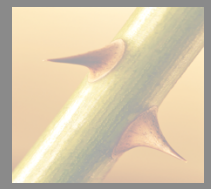


Research challenges



- Greater synergy between programming models and large scale systems (data stores, streaming, messaging, caching systems)
  - languages can help to *compose* functionality more effectively
- "Compiling in the large"
  - optimizing networking, data access, process placement, network caching
  - more critical to large system performance than optimizing registers, instructions
- Managing failures
  - how much to expose to application programmers, how much to hide?
  - what are failures consequences when systems are _composed_?
- Harnessing distributed compute and data resources
  - explicit control of resources vs. resource management by "Cloud OS"?
- How to build high-level abstractions on lower-level distributed systems?
- Encapsulating existing systems, code without introducing fragility
- What are the right types, annotations for *large scale* composition and specialized domains?

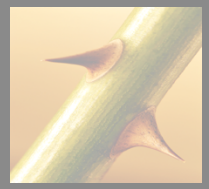
Cloud optimization challenges



- Simple data splitting:
  - split components whose communications access disjoint data
- Replicate stateless components
  - can arbitrarily replication components where state not accessed across multiple communications
- Sharding
  - split components with table state into disjoint key spaces
- Batch→Stream
  - replace sequence of bulk data transformations with parallel per-item transformations
- Generalized map-reduce
  - identify parallelizable queries, break into pipelines
- Caching
  - introduce intermediate components that store the results of computations

- NB: *These optimizations are much easier to do when the source language understands processes and associative datatypes*

More information



- http://www.thorn-lang.org
  - download interpreter
  - links to papers
  - online demo
- Additional collaborators welcome!