Thorn—Robust, Concurrent, Extensible Scripting on the JVM

Bard Bloom[1], John Field[1], Nathaniel Nystrom[2*], Johan Östlund[3],
Gregor Richards[3], Rok Strniša[4], Jan Vitek[3], Tobias Wrigstad[5†]

[1] IBM Research [2] University of Texas at Arlington [3] Purdue University
[4] University of Cambridge [5] Stockholm University

Abstract

Scripting languages enjoy great popularity due to their support for rapid and exploratory development. They typically have lightweight syntax, weak data privacy, dynamic typing, powerful aggregate data types, and allow execution of the completed parts of incomplete programs. The price of these features comes later in the software life cycle. Scripts are hard to evolve and compose, and often slow. An additional weakness of most scripting languages is lack of support for concurrency—though concurrency is required for scalability and interacting with remote services. This paper reports on the design and implementation of Thorn, a novel programming language targeting the JVM. Our principal contributions are a careful selection of features that support the evolution of scripts into industrial grade programs—*e.g.*, an expressive module system, an optional type annotation facility for declarations, and support for concurrency based on message passing between lightweight, isolated processes. On the implementation side, Thorn has been designed to accommodate the evolution of the language itself through a compiler plugin mechanism and target the Java virtual machine.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed, and parallel languages; Object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; Modules, packages; Classes and objects; Data types and structures; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Design

Keywords Actors, Pattern matching, Scripting

* Work done while the author was affiliated with IBM Research.

† Work done while the author was affiliated with Purdue University.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

1. Introduction

Scripting languages are lightweight, dynamic programming languages designed to maximize short-term programmer productivity by offering lightweight syntax, weak data encapsulation, dynamic typing, powerful aggregate data types, and the ability to execute the completed parts of incomplete programs. Important modern scripting languages include Perl, Python, PHP, JavaScript, and Ruby, plus languages like Scheme that are not originally scripting languages but have been adapted for it. Many of these languages were originally developed for specialized domains (*e.g.*, web servers or clients), but are increasingly being used more broadly.

The rising popularity of scripting languages can be attributed to a number of key design choices. Scripting languages' pragmatic view of a program allows execution of completed sections of partially written programs. This facilitates an agile and iterative development style—"at every step of the way a working piece of software" [9]. Execution of partial programs allows instant unit-testing, interactive experimentation, and even demoing of software at all times. Powerful and flexible aggregate data types and dynamic typing allow interim solutions that can be revisited later, once the understanding of the system is deep enough to make a more permanent choice. Scripting languages focus on programmer productivity early in the software life cycle. For example, studies show a factor 3–60 reduced effort and 2–50 reduced code for Tcl over Java, C and C++ [38, 39]. However, when the exploratory phase is over and requirements have stabilized, scripting languages become less appealing. The compromises made to optimize development time make it harder to reason about correctness, harder to do semantic-preserving refactorings, and in many cases harder to optimize execution speed. Even though scripts are succinct, the lack of type information makes the code harder to navigate. An additional shortcoming of most scripting languages is lack of first-class support for concurrency. Concurrency is no longer just the province of specialized software such as high-performance scientific algorithms—it is ubiquitous, whether driven by the need to exploit multicore architectures or the need to interact asynchronously with services on the Internet. Current scripting languages, when they sup-

port concurrency at all, do so through complex and fragile libraries and callback patterns [20], which combine browser- and server-side scripts written in different languages and are notoriously troublesome.

Currently, the weaknesses of scripting languages are largely dealt with by rewriting scripts in either less brittle or more efficient languages. This is costly and often error-prone, due to semantic differences between the scripting language and the new target language. Sometimes parts of the program are reimplemented in C to optimize a particularly intensive computation. Bridging from a high-level scripting language to C introduces new opportunities for errors and increases the number of languages a programmer must know to maintain the system.

Design Principles. This paper reports on the design and the implementation status of Thorn, a novel programming language running on the Java Virtual Machine (JVM). Our principal design contributions are a careful selection of features that support the *evolution* of scripts into robust industrial grade programs, along with support for a simple but powerful concurrency model. By "robust", we mean that Thorn encourages certain good software engineering practices, by making them convenient. The most casual use of classes will result in strong encapsulation of instance fields. Pattern matching is convenient and pervasive—and provides some of the type information which is lost in most dynamically typed languages. Code can be encapsulated into modules. Components provide fault boundaries for distributed and concurrent computing.

Thorn has been designed to strike a balance between dynamicity and safety. It does not support the full range of dynamic features commonly found in scripting languages, though enough of them for rapid prototyping. It is static enough to facilitate reasoning and static analyses. Thorn runs on the JVM, allowing it to execute on a wide range of operating systems and letting it use lower-level Java libraries for the Thorn runtime and system services. Thorn has the following key features:

Scripty: Thorn is dynamically-typed with lightweight syntax and includes powerful aggregate data types and first-class functions.

Object-oriented: Thorn includes a simple class-based multiple inheritance object model that avoids the main complexities by simple static restrictions.

Patterns: Thorn supports an expressive form of pattern matching for both built-in types and objects.

Immutable data types: Thorn encourage the use of immutable data, which typically eases code composition and evolution.

Concurrency and distribution: Non-sequential programming is done by lightweight, isolated, single-threaded components that communicate by message passing.

Modules: An expressive module system makes it easy to wrap scripts as reusable components.

Constraints: Optional *constraint annotations* on declarations permit static or dynamic program checking, and facilitate optimizations.

Extensibility: An extensible compiler infrastructure written in Thorn itself allows for language experimentation and development of domain specific variants.

It is important to note what Thorn does *not* support. *Threads:* There is no shared memory concurrency in Thorn, hence data races are impossible and the usual problems associated with locks and other forms of shared memory synchronization are avoided. *Dynamic loading:* Thorn does not support dynamic loading of code. This facilitates static optimization, enhances security, and limits the propagation of failures. However, Thorn does support dynamic creation of components, which can be used in many applications that currently require dynamic loading. *Introspection:* Unlike many scripting languages, Thorn does not support aggressively introspective features. *Unsafe methods:* Thorn can access Java libraries through an interoperability API, but there is no access to the bare machine through unsafe methods as in C# [22]. *Java-style interfaces:* Their function is subsumed by multiple inheritance. *Java-style inner classes:* These are complex and difficult to understand, and are largely subsumed by simpler constructs such as first-class functions. *Implicit coercions:* Unlike some scripting languages, Thorn does not support implicit coercions, *e.g.*, interpreting a string `"17"` as an integer `17` in contexts requiring an integer.

Targeted Domains. Thorn is aimed at domains including client-server programming for mobile or web applications, embedded event-driven applications, and distributed web services. The relevant characteristics include a need for rapid prototyping, for concurrency and, as applications mature, for packaging scripts into modules for reuse, deployment, and static analysis. Thorn plugins provides language support for specialized syntax and common patterns, *e.g.*, web scripting or streaming. They make it possible to adapt Thorn to specialized domains and allow more advanced optimizations and better error handling than with macros.

Implementation Status and Availability. Thorn is joint project between IBM Research and Purdue.There are currently two implementations of Thorn on the JVM: a feature-complete reference implementation in the form of an interpreter, and a slightly restricted bytecode compiler. We are re-engineering the compiler to extend it to the full language. The only features discussed in this paper that are not fully supported in the interpreter (but available in the compiler) are general type constraints and language plugins. All of the examples in this paper have been tested on the reference implementation. More information is available from the Thorn web site at `http://www.thorn-lang.org`. An open source version of Thorn is expected in late 2009.

2. From Scripts to Programs

As an example of the succinctness that scripting requires, here is a simple variant of the Unix `grep` program written in Thorn. The call `argv()` returns the list of command-line arguments.

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

Subscripting uses parentheses and is 0-based, so `argv()(0)` is the first argument. The `file()` method produces a file object with the given name, and `contents()` gets the contents of the file as a string, and `split("\n")` breaks it into a list of lines. **`for`** iterates over the list, binding `l` to successive lines. The **`if`** tests whether the second command-line argument is in that line, and, if so, prints it.

2.1 Dining Philosophers

Figure 1 is a Thorn solution to the Dining Philosophers problem. We will cover of all of the Thorn constructs used in here in detail in subsequent sections; here are a few highlights.

Forks and philosophers are represented as *components*, analogous to threads or processes, which communicate by sending messages back and forth, and do not share any state. The `Fork` component describes a fork with two state variables, `holder` for the philosopher currently holding the fork, and `waiter` for the one waiting for it. These variables can be **`null`** if there is no such philosopher. The local function `taken()` describes what to do when philosopher `phil` picks up the fork: *viz.*, change the `holder` variable, and send the message `"taken"` to the philosopher. The main behavior of a `Fork` is given in its **`body`** section which loops indefinitely, waiting to receive one of three kinds of messages. Upon reception of the string `"die"` it breaks from the **`while`** loop, ending the **`body`** and thus the component. If a record with a field labeled `take` is received from any philosopher `phil`, it will respond appropriately to `phil`: either giving him the fork or making him wait. Finally, it responds to `"drop"` by having the holding philosopher put the fork down, and, if necessary, the waiting one pick it up.

In Thorn, blocks are enclosed in braces, as in C-derived languages. However, blocks begun with a keyword or identifier can optionally be closed by the same symbol immediately after the `}`, as in `}Fork` closing the component.

Once the definition of `Fork` is complete, the program spawns three instances of `Fork`, using a list comprehension expression `forks=%[...|for i<-0..2]`. The values in the list are component handles, immutable references to the components, which can be used to send messages to them. The list is bound to a variable, `forks`, whose scope includes the ensuing component declaration for `Phil`. One important property of the Thorn component model is that, semantically, no state is shared between component instances. As an optimization, the implementation may share immutable state; but this is invisible to Thorn. When a global constant, like `forks`, is used within a component definition, no state is shared; `forks` is treated as a parameter to the component, and, thus, passed by copy.

```
component Fork(n) {
  var holder := null;
  var waiting := null;
  fun taken(phil) {holder := phil; phil <<< "taken";}
  body {
    while (true) {
      receive{
         "die" => {break;}
       | {: take:_ :} from phil => {
           if (holder == null) taken(phil);
           else waiting := phil;
         }
       | "drop" from phil => {
           phil <<< "dropped";
           if (waiting != null) {
             taken(waiting);
             waiting := null;
           } else holder := null;
  } } } }
}Fork;

forks = %[ spawn Fork(i) | for i <- 0 .. 2];

component Phil(name, ln, rn, iter) {
  body {
    left = forks(ln);
    right = forks(rn);
    for(i <- 1 .. iter) {
                  # THINK: I think, therefore I am.
       left  <<< {: take:name :}; receive{"taken" => {}};
       right <<< {: take:name :}; receive{"taken" => {}};
                  # CRITICAL: I eat, therefore I am fed.
       right <<< "drop";  receive{"dropped" => {}};
       left  <<< "drop";  receive{"dropped" => {}};
  } }
}Phil;

phils = [
  spawn Phil("Kant", 0, 1, 10),
  spawn Phil("Hume", 1, 2, 12),
  spawn Phil("Marx", 0, 2, 8)];
  # A 3-way philosophical dinner now ensues.
```

Figure 1. Dining Philosophers in Thorn

The `Phil` component has several parameters: philosopher name, index of the left and right forks, and the number of times to think. A philosopher starts out by binding his left and right forks to two variables. The statement `left = forks(ln)` introduces an immutable binding to a fresh variable. Thorn distinguishes between mutable (**`var`**) and immutable (**`val`** or `=`) variables. The statement `holder := phil`, for example, assigns to an extant mutable variable; it cannot be used on an immutable one. As shadowing of variable names is forbidden, it is all but impossible to introduce a new variable when the intent was to update an extant one, or vice-versa. Most scripting languages allow the assignment operation to create new variables as well as modify them. Thorn's approach is nearly as convenient, and safer.

The philosopher iterates `iter` times, thinking and eating. After thinking, it picks up its left fork. This is done in two parts: `left <<< {: take:name :}`, which constructs a record with one field `take` bound to the philosopher's name, and sends it to the left fork. Then it waits for the fork to reply. The reply needs to be the string `"taken"`; other incoming messages (of which this example has none) are ignored. No particular action needs to be done when the message shows up, hence the empty block in the **`receive`**. The philosopher repeats the same send and receive with the right fork. After the critical section, the philosopher drops the forks in the opposite order, using the same mechanisms to communicate.

The example concludes with the spawning of three famous philosophers. Each is provided with its proper forks; Marx's forks are reversed to break the symmetry and allow the usual Dining Philosophers solution to work. When a philosopher is spawned, it starts executing its **`body`** clause, picking up forks and so on. The system runs philosophically from this point on.

2.2 A Server in Thorn

We now turn to a more complex example: a matchmaking application. The heart of the service is a date service component that interacts with a client components acting on behalf of customers seeking dates. The service advertises itself through a central registry, here presented for brevity in a minimalist form only capable of registering a single dating service. The code shown here could run within a single JVM as the previous example, but we have chosen to distribute it.

```
module dating {
  fun registry() = site("thorn://localhost:5555");

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

  class Profile(interests, limit) : pure {
    def compatible?(other) =
      other != null &&
      this.compat?(other) && other.compat?(this);
    def likes?(topic) =
      %exists(t==topic | for {: topic:t :} <- interests);
    def compat?(other : Profile) {
      matches =  %[ weight |
        for {:topic, weight:} <- interests,
        if other.likes?(topic)];
      sum(matches) > limit;
    }
  }
}dating

spawn Registry {
  var datesvc;
  sync dateServicePlz() = datesvc;
  sync registerDateSvc(datesvc') { datesvc := datesvc';  }
  body { while (true) serve; }
};
```

Figure 2. Dating module and registry in Thorn.

Every component (marked by the keyword **`spawn`**) runs in a different JVM. Component handles contains sufficient information to identify the node and port on which the component runs.

Thorn allows shared code to be organized in *modules*: bindings of names to components, classes, values, functions, and so on. Figure 2 defines the `dating` module, which exports the definition of the `registry` and `sum` functions, and the `Profile` class: the data structure that explains what a client of the dating service is looking for. Methods in classes are declared using the **`def`** keyword. Functions and methods always return a value, which is the last expression computed within the function body. `Profile` defines three methods; by convention method names ending in `?` return Boolean values. The `pure` annotation on `Profile` indicates that the class cannot refer to mutable or component-local state, and thus, its instances are suitable for transmission between components.

The `DateService` and `SampleClient` components, shown in Figure 3, both import the definitions in the module `dating`. In particular they share the `Profile` class. The keyword **`spawn`** is used here to create a singleton instance of the component definition that follows it, the component name is optional.

The `Registry` component is intentionally simplified. All it does is hold a component handle, the `DateService` component in this case, and provides an interface for accessing that handle.

Thorn enjoys two communication models. The Dining Philosophers used the send/receive model, which is suitable for intricate communication patterns. The dating service uses the **`serve`** model, more suitable for client/server and RPC-style coordination. Thorn components can have *communication declarations* which define code to be executed upon receipt of certain messages. The `DateService` component defines three. The **`sync`** communications (here `register` and `makeMeAMatch`) return a value to the sender, and, thus, are blocking. The **`async`** communications (here, `stop`) return nothing and are not blocking. An expression of the form `datesvc <-> register("Whale", whale)` sends a message to a component then awaits a reply (in this case, a string). Alternatively, a message can be sent asynchronously, as in `datesvc <-- stop()`, in which case the sender continues on without waiting for a reply. The difference between `c <<< v` and `c <-- m(s)` is that former tosses the value `v` into `c`'s mailbox with no further semantics implied, while the latter evaluates the body of `c`'s **`async`** named `m`.

Thorn's built-in types include lists, records, and a powerful associative *table* type. The expression `{: source:sender, name, profile, picked:[name] :}` in the `register` method is a record constructor. It yields a record with four fields (`source`, `name`, `profile`, `picked`). For convenience, an identifier occurring alone in a record constructor serves as both field name and value, so `name` abbreviates `name:name`. The expression `[name]` constructs a list with a single element.

```
spawn DateService {
  import dating.*;

  var done := false;
  val customers = table(name) {
      val source, profile; var picked;
  };

  async stop() { done := true; }

  sync register(name, profile) from sender {
    if (customers(name)==null)
      customers(name) :=
        {: source:sender, name, profile, picked:[name] :};
    null;
  }

  sync makeMeAMatch(name) {
    if (customers(name) ~ +r) findMatchFor(r);
    else {: failed: "Please register first!" :};
  }

  fun findMatchFor({: profile:pr, name:nm, picked :}) {
    first(for {: profile:pr', name:nm' :} <~ customers,
          if pr.compatible?(pr') && !(nm' in picked)) {
       customers(nm).picked := nm' :: picked;
       return {: date: nm' :};
      }
    else return {: failed: "No match":};
  }findMatchFor

  body {
    registry() <-> registerDateSvc(thisComp());
    while (!done) serve;
  }
}DateService;


spawn SampleClient {
  import dating.*;

  body {
    datesvc = registry() <-> dateServicePlz();
    porpoise = Profile([
      {: topic: "swimming",    weight: 15:},
      {: topic: "moon",        weight: 7:},
      {: topic: "watersports", weight: 10:}], 10);
    whale = Profile([
      {: topic: "swimming",    weight: 15:},
      {: topic: "singing",     weight: 10:},
      {: topic: "watersports", weight: 3:}], 5);

    # Register some clients...
    datesvc <-> register("Porpoise", porpoise);
    datesvc <-> register("Whale", whale);

    # Exercise the dating service.
    date =  datesvc <-> makeMeAMatch("Porpoise");
    #...a whale
    noDate = datesvc <-> makeMeAMatch("Porpoise");
    #...failure

  }
}SampleClient;
```

Figure 3. Dating Service program in Thorn.

Conceptually, a table such as `customers` in `DateService` is simply a collection of records, or *rows*, defined over a common set of field names, or *columns*. The subscripting expression `customers(name)` retrieves the row whose key equals `name`, or **`null`** if there is no such row. A table is created with an expression such as **`table`**`(name){`**`val`**` source, profile; `**`var`**` picked;}` which yields a new table mapping names to three values. Thorn has a rich collection of query expressions, *e.g.*, **`first`** which seeks a suitable set of values and binds them to variables if they exist, or executes an **`else`** clause if they do not.

The **`serve`** statement in the **`body`** of `DateService` and `registry` explicitly waits for a single incoming message: one of the component's **`sync`**s or **`async`**s. Incoming messages to a component are queued and their corresponding communication bodies are executed serially. Each component has only a single thread of control. Data races are impossible. They are replaced by message races, which are generally more benign: programmer error may result in operations happening in the wrong order, but not happening simultaneously.

3. Related Work

Designing Thorn, we carefully examined a variety of programming languages: Java, Scala, Perl, Python, Smalltalk, Lua, Ruby, JavaScript, Haskell, ML, Scheme, Lisp, Erlang, and others. Thorn takes some aspects of these languages and extends them with novel variations. The syntax of Thorn is influenced primarily by Scala [37] and Python [57]. Thorn supports a class-based object model, described in Section 4.1, similar to Java's or Scala's, but simpler to program. Thorn supports multiple inheritance with restrictions to avoid the diamond problem. Other recent languages [50, 42, 37, 3, 31] distinguish between different modes of inheritance for different kinds of entities (*e.g.*, interfaces, traits, mixins) to avoid these problems. Like Kava [7] and X10 [41], Thorn provides pure classes, which encourage a functional programming style while preserving the extensibility and flexibility of object-orientation. Thorn provides pattern matching features influenced by ML data types [29, 33], Scala extractors [17], active patterns [53], views [58], and more directly from our previous on Matchete [23].

Thorn includes associative data types and queries which are inspired by the design of SETL [43], SQL [13], LINQ [32], Haskell's list comprehensions [24], and CLU iterators [30].

Thorn's module system is based upon the Java Module System [51, 52]. Most scripting and concurrency-oriented programming languages have poor support for modularity, often lacking even *weak hierarchical visibility* (selective exporting combined with optional re-exporting) and support for versioning. For example, Ruby [54] only focuses on code reuse with its mixins [11]. Erlang's modules [1] support selective importing and exporting of functions, where

imported names are automatically merged into the local namespace, which leads to fragile module definitions. In Python [2], each source file creates its own namespace, which depends on its location in the file system. While selective importing is supported, the imported names can easily be merged with the local namespace, again leading to fragility. PLT Scheme [18] is a rare exception, defining an expressive module system that supports both weak hierarchical visibility, versioning, selective importing, and renaming. But, as with Python, each source file defines a module, and imported names are merged into the local namespace.

Inspired by Erlang [1], actors [6, 37, 47], and languages like Concurrent ML [40], Thorn supports concurrent and distributed programming through lightweight single-threaded components that communicate asynchronously through message passing. Thorn components are logically separate. This style of distributed programming improves system robustness by isolating failures. Clojure [15] supports asynchronous message passing through agents. Agents react to messages and can update a single memory location associated with the agent. Recent languages for high-performance computing such as X10 [41, 14] and Chapel [12] support partitioned global address spaces; these languages permit references to remote data, and hence do not readily support failure isolation.

A central idea in Thorn is that it supports the evolution of scripts into programs through optional annotations and compiler plugins. Pluggable and optional type systems were proposed by Bracha [10]. Compiler plugins use the type annotations to reject programs statically that might otherwise have dynamic type errors. By contrast, Thorn plugins can perform arbitrary transformations of the program. Java 5 [50] annotations adapt the pluggable type system idea to Java, which retains its original type system and run-time typing semantics. JavaCOP [5] is a pluggable type system framework for Java annotations in which annotations are defined in a meta language that allows type-checking rules to be specified declaratively. Thorn provides a richer annotation language than Java's, permitting arbitrary syntactic extensions that enable more natural domain-specific annotations rather than requiring that annotations fit into a narrow annotation sub-language. An instance of optional annotations in Thorn is its gradual typing system. This type system builds on recent work on gradual typing in dynamic languages [4, 44, 21, 45, 56, 55].

The Thorn compiler is built on an extensible compiler infrastructure, providing a compiler plugin model similar to X10's [35]. Plugins support both code analysis and code transformations. The design of the compiler itself is based on the design of the extensible compiler framework Polyglot [34]. The meta-programming features and syntax extension features of Thorn were influenced by Lisp [48] and Scheme macros [46].

Thorn is one of many dynamic languages and scripting languages implemented on the JVM [27, 25, 8, 15, 28]. We examined the implementation of several of these languages when designing Thorn to avoid some of the performance pitfalls. For example, the decision to disallow dynamic extension of objects was motivated by the requirement that objects be implemented efficiently. Unlike Groovy [8], for instance, we do not use reflection to implement method dispatch.

4. Thorn for Scripts

Thorn data can be classified into the categories listed in Figure 4. Every datum in Thorn, including atomic values such as integers, is an instance of a *primitive object*. Conceptually, a primitive object is a bundle containing some state (perhaps mutable) and a collection of methods (possibly empty). We will use the term *object* exclusively for referring to instances of classes, and data created using the **`object`** construct. Class and component *definitions* are not data in Thorn, though class and component *handles* are.

atomic value	boolean, numerics, immutable string
record	immutable set of named values
list	immutable vector of values
table	mutable associative aggregate
ord	mutable ordered collection
object	anonymous or class instance
closure	anonymous or named function
component handle	lightweight process reference

Figure 4. Thorn data categories.

4.1 Classes

Some scripting languages have a very slippery (or, if you prefer, flexible) concept of "class". Lua, and to some extent JavaScript, lets programmers define their own concept altogether. Ruby and Python are not quite this extreme, but object structure is determined at runtime. In such languages, it would be perfectly normal to have a class `Person`, one instance of `Person` that has a `name` field, and another that does not. If programmers take advantage of this feature, their code is likely to be difficult to understand or maintain. Thorn has a more concrete concept, hearkening to the statically-structured world of Java and C++. A class statically determines the structure of its instances: their fields, methods, superclasses, and so on. The syntax for classes is relatively lightweight.

Formal Class Parameters. Inspired by Scala, Thorn classes can be defined with formal parameters, which induce fields, constructors, and the extractors introduced in Section 4.2. For example,

```
class Point(x,y) {}
```

abbreviates the more Java-like

```
class Point {
   val x; val y;
   new Point(x',y') { x = x'; y = y'; }
}
```

Class formals passed as arguments to parent classes do not induce variables. So, the following declaration produces a subclass of `Point` with one new field `t`.

```
class TastyPoint(x,y,t) extends Point(x,y) {}
```

Methods. Methods are introduced by the keyword **`def`** and are called by the traditional `r.m(x,y)` syntax.[1] As in Scala, single-expression methods can be introduced with a compact syntax, and methods whose bodies are larger can use `{}`-blocks.

```
class Rectangle(xl, yl, xh, yh) {
  def well_formed?() = xl < xh && yl < yh;
  def area() {
   if (this.well_formed?()) {(xh-xl) * (yh-yl);}
   else throw "Degenerate rectangle";
  }
}
```

Instance Variables. The treatment of instance variables is a sweetened version of Smalltalk's. An object's instance variables are visible inside its definition; no other object's are. Thorn provides default getters and setters for all instance variables, unless the user has provided different ones. From outside, all access to fields is done by method call. The syntax `x.f` is simply an abbreviation for the nullary method call `x.f()`, and `x.f:=g;` is sugar for `x.'f:='(g);`.[2] The class `A` below, from outside, seems to have **`var`** fields including `b`, `c`, and `e`, and **`val`** field `d`. `a.b:=a.b+1` will increment the `b` field of `A` `a`. `a.c:=3` will work, but `a.c:=4` will not, as assignment to the `c` field is specialized to only work for primes. There is no instance variable named `e`, but `A`'s behave from the outside as if there were one: `a.e:=a.e+1;` does just what one would expect. Attempts to access the representation variable `secret` from outside the class will always throw exceptions.

```
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 e() = secret;
  def 'e:='(v) { secret := v; }
  def secret() {throw "Please don't";}
  def 'secret:='(x) {throw "Please don't";}
}
```

Many scripting languages have all fields *actually* public, which is delightfully convenient in the short term, but inhibits good software engineering practices like data hiding and enforcement of representation invariants in the long term. Our approach retains the convenience—by default all fields *appear* public—but, since the actual representation is always defended by an abstraction barrier, allows good practices as well.

Constructors. Constructor invocations look like function calls: `A()` creates a new instance of the class `A`. Constructors are defined with the **`new`** keyword. Within the constructor body (and only there), **`new`** may be called with arguments to evaluate the code of another constructor, rather the way that **`this`** can be called with arguments in Java. A cons cell class could be defined:

```
class Pair {
  val fst; val snd;
  new Pair(f,s) { fst = f; snd = s; }
  new Pair(f) { new(f,null); }
}
```

New pairs can be created by calls like `Pair(1,2)`, or, where `snd` is to be null, `Pair(3)`. The class `Pair` has two immutable **`val`** fields, which must be bound exactly once inside the constructor and cannot be rebound thereafter.

A pernicious source of errors in many languages is the ability of a constructor to leak references to the object being constructed before it is fully initialized, exposing state changes in immutable fields. Thorn disallows the use of **`this`** inside constructors, avoiding this problem. There are, of course, situations where one needs to refer to a newly-constructed object; *e.g.*, it is sometimes desirable to put every `Person` object into a list. Such bookkeeping can be put into the distinguished `init()` method, that, if present, is called after the constructor body but before the constructor returns, and, like any method, can refer to **`this`**.

Multiple Inheritance. Multiple inheritance, in its full generality, is extremely powerful but sometimes extremely confusing. Thorn supports a restricted form of multiple inheritance, designed to be reasonably straightforward to understand and to give most of the advantages of fuller forms of multiple inheritance with only a modest amount of extra work. The following is the defines a class `TastyPoint` as the composition of two classes, `Point` and `Flavor`.

```
class Flavor(fl) {}
class TastyPoint(x,y,fl)
  extends Point(x,y), Flavor(fl) {}
```

The first knot that Thorn cuts is method precedence. If `Point` and `Flavor` were expanded to provide a predicate `nice?()`,

[1] Unlike Java, the receiver `r` is required, even when it is **`this`**. Otherwise it would not be possible to distinguish between a call to a function `f` and a call to a method named `f`, or, more confusingly, a method named `f` that the class itself does not define but some subclass does.

[2] Nearly any string can be used as an identifier, by surrounding it in backquotes. This is used for methods with symbols as names, such as **`'f:='`** (field assignment) and + (addition operation).

but TastyPoint were not, what would tp.nice?() do? Some languages have elaborate precedence rules to control this situation. Thorn simply forbids it. If two parents of a class both have a method with the same name and arity, then the class must override that method. Often the code for this definition will involve a supercall to one or the other parent's method, using the **super** syntax to say which to call:

```
def nice?() =
  super@Point.nice?() && super@Flavor.nice?();
```

The second knot that Thorn cuts is multiple inheritance of mutable state. If class A has a single **var** field a, and B and C both extend A, and D extends B and C, then how many fields do instances of D have? In some cases, it is desirable for the a inherited from B to be the same as that from C; in other cases, they should be different. If they are the same, which constructor gets to initialize it? C++ has a subtle answer to these questions. Thorn has a simpler answer: multiple inheritance of mutable state is forbidden altogether. Similar restrictions forbid inheritance of immutable fields that could be fixed at different values depending on which inheritance path is taken.

Anonymous Objects. It is sometimes convenient to have anonymous objects: that is, to construct an object without bothering to code a whole class for it. Java uses anonymous objects extensively for callbacks, comparators, and other bits of code packaged to be first-class. They are less important in Thorn, since Thorn has closures and is not as fussy about types. Nonetheless, they can be convenient. They are built thus:

```
fun makeCounter() {
   var n := 0;
   object { def inc() {n += 1;} }
}
```

Each call to makeCounter() produces a separate counter object. Making n a field of the object, or returning a closure, would be more idiomatic Thorn.

Operator Overloading. Just like in Smalltalk, operators are implemented through method calls: 1+2 is syntactic sugar for 1.‘+‘(2). The usual flora of operators are allowed to be called without the dot syntax. Like in Ruby but unlike Smalltalk, operator expressions have their traditional precedence hierarchy. So, 1+3*2 is correctly treated as 1.‘+‘(3.‘*‘(2)) rather than Smalltalk's (1.‘+‘(3)).‘*‘(2). Since Thorn is untyped, operations on all objects (*e.g.*, user-defined Complex numbers) have the same precedence hierarchy.

4.2 Pattern Matching

Thorn provides a powerful set of facilities for matching patterns and extracting data from objects and built-in data structures. Patterns provide a good way to specify the expected structure of values, and to disassemble them and use their parts. This mitigates some of the disadvantages of having an untyped language. Indeed, it provides expressive power beyond most type systems: pattern matching can be used to check that “this list contains three elements” just as easily as “this list contains only integers”. It also provides a degree of convenience that programmers will appreciate. Several constructs in Thorn do pattern matching; for this section we only use exp ∼ pat which returns true if exp's value matches pat, and false otherwise.

Most built-in types provide patterns that parallel their constructors. For example, as an expression, 1 is an integer; as a pattern, 1 matches precisely the integer 1. As an expression [h, t...] produces a list whose head is h and whose tail is t, a synonym for h @ t. As a pattern, it matches such a list; *e.g.*, it would match the list [1,2,3,4] binding h to 1 and t to [2,3,4]. Some built-in pattern constructs do not correspond to Thorn types: +p matches a non-null value that matches p. $(e) matches the value of the expression e, and (e)? matches if the boolean expression e evaluates to true. Type tests are available in patterns as well: [x:int, p:Person] matches a two-element list containing an integer followed by a Person.

The conjunctive pattern p && q matches a value that matches both p and q. For example,

```
r && {: source:$(sender) :}
```

matches a record whose source field is equal to sender—it may have an arbitrary set of other fields as well—and binds that record to r, thus behaving like ML's **as**. However, && is more powerful than **as**. The pattern

```
[_..., 1, _...] && [_..., 2, _...]
```

matches a list containing both 1 and 2 in either order. x && (x>0)? matches a positive number and binds it to x. This trick eliminates the need for side conditions in pattern expressions, and allows finer control of when pattern matching is stopped than side conditions do: the match lst∼[x:int && (x != 0)?, $(32 **div** x)] is true on [4,8] and false (rather than dividing by zero) on [0,0].

Thorn also has disjunctive and negative patterns. p||q succeeds if either p or q succeeds, and !p succeeds iff p fails. For example, [(3||!(_:int))...] matches lists whose integral elements are all 3, such as [3, **true**]. Note that this could not be expressed as nicely as a Boolean combination of non-Boolean patterns. These patterns produce no externally-visible bindings.

It is often desirable to look for something, and return it if it was found, or a note that it was not found otherwise. Thorn's idiom for this uses the +e operation, and its inverse the +p pattern. +e (pronounced “positively e”) packages the value of expression e in a way guaranteed to be non-null. +p matches a subject that is non-null, and, when + is inverted, matches p. *E.g.*, +1∼+x is true and binds x to 1, and +**null**∼+y is true and binds y to **null**, but **null**∼+z is false. This pair of operations gives a convenient idiom for func-

tions that search for something and return what they find. For instance, `assoc(x,lst)` searches a list of 2-element lists `lst` for a list whose first element is `x`, and returns the second element—packaged with `+` to distinguish the case where the second element is **`null`** from the case where `x` was not found.

```
fun assoc(x, [])                 = null;
  | assoc(x, [[$(x), y], _...]) = +y;
  | assoc(x, [_, tail...])      = assoc(x,tail);
```

Then, `assoc(a,b)`∼`+c` will succeed and bind the value `b` associates with `a` to `c`, or fail if `a` is not found.

The functionality of `+e` could be achieved by boxing `e`, and having `+p` unbox it. However, this operation is quite common in Thorn, and we do not want to pay for all the boxes. So, Thorn avoids constructing new data structures in nearly all cases. `+x == x` for nearly all Thorn values. Obviously, +**`null`** cannot be **`null`**, so it is an otherwise undistinguished constant not used for anything else—as are ++**`null`** and so on, though they rarely arise in programs.

Patterns may be nested arbitrarily. Bindings produced by matching are available to the right of the binding site: `[x,!$(x)...]` matches a list of size two or more, whose later elements are not equal to the first element. Quite intricate structures can be described quite succinctly in this way, and, if one is careful, they can even be understood with sufficient study.

The formal parameters of classes define *extractors*, inverse to their default constructors. So, for the class definition **`class`** `Point(x,y){}`, we can use `Point` as an extractor. Matching `Point(1,2)` against pattern `Point(1,z)` succeeds and binds `z` to `2`.

Matching Control Structures. Matching is used in a number of contexts. The **`match`** construct matches a subject against a sequence of patterns, evaluating a clause for the first that matches. One way to write the function to sum a list is:

```
fun sum(lst) {
  match (lst) {
     []       => { 0; }
   | [h,t...] => { h + sum(t); }
} }
```

Functions and methods can match on their arguments. Another way to write `sum` is:

```
fun sum([])       = 0;
  | sum([h,t...]) = h + sum(t);
```

The binding operation `=` allows patterns, not just simple variables, on the left. For example, `[h,t...]=lst` binds `h` and `t` in the following code if `lst` is a nonempty list, and throws an exception if `lst` is anything else. This is useful for *destructuring* in the Lisp sense: when one is certain what some structure is, and wishes to take it apart and use the pieces. Iteration over a list includes destructuring. So, to loop over a list of two-element pairs, one might use **`for`**`([fst, snd]<-pairs)`.

When one is less certain, the ∼ operation is used. `s`∼`p` matches subject `s` against pattern `p`, returning true or false. It also introduces the bindings inspired by `p` into code that the match *guards*, *i.e.*, that is obviously evaluated iff the match succeeds. A conjunction of patterns in the test of an **`if`** introduce bindings into the then-clause. The function `zip`, turning two lists into a list of pairs (`zip([1,2,3],[11,22])` `=[ [1,11], [2,22] ]`) can be written:

```
fun zip(a, b) {
  if (a ∼ [ha, ta...] && b ∼ [hb, tb...]) {
    # ha, ta, hb, tb are bound here
    [ [ha, hb],  zip(ta,tb)... ];
  } else {
    []; # They're not bound here.
  }
}
```

This behavior combined with `+` gives the Thorn searching idiom. To do something with the value associated with `a` in `B`, as in the previous section:

```
if ( assoc(a,B) ∼ +c ) doSomething(c);
else dealWithMissing(a);
```

Similarly, matches in **`while`** loops produce bindings in the loop body.

```
p = Person(); thingsHappen(p);
while (p.spouse ∼ +q) {
  # 'q' is bound to p's spouse here
  otherThingsHappen(p,q);
}
# Now 'p' is not married and 'q' is unbound.
```

Dually, **`until`** loops can produce bindings *after* the loop. The plot of many romance novels can be formalized as:

```
p = Person();
do { seekSpouse(p) } until (p.spouse ∼ +q);
# 'q' is bound to p's spouse here.
```

4.3 Built-in Data Types

Thorn enjoys a selection of built-in data types, with their constructors, pattern matches, and methods. Strings are ordinary immutable Unicode strings. As in some other scripting languages, the `$` character interpolates values into strings: `x = "John"; "Dear $x"` evaluates to `"Dear John"`.

Ranges are finite lists of consecutive integers: `1..4` has elements 1,2,3,4. Ranges are, of course, implemented efficiently. Ranges are a useful utility class: *e.g.*, **`for`**`(i<-1..4)` is an ordinary loop over integers; `lst(1..4)` is a slice of the list `lst`; and `n` **`mod`** `2..5` is the number between 2 and 5 congruent to `n` mod 4.

Lists are immutable ordered collections. The term `[a,b,c]` constructs or matches a fixed-size list. A postfix ellipsis `...` in a list expression indicates a sublist to be appended or matched. So, the standard map function could be defined as:

```
fun lstmap(f, []) = [];
  | lstmap(f, [x, y...]) = [f(x), lstmap(f,y)...];
```

Lists can be subscripted: `lst(0)` gets the first element, `lst(-1)` the last element, and `lst(1,-1)` is the tail of `lst`.

Records are finite collections of immutable named fields. The syntax `{: a:1, b:2 :}` is used for record constructors and patterns. A lone identifier `a` abbreviates `a:a` in a record. The selector `r.a` gets single fields from records; pattern matching can get several fields out at once. Both field selector and record pattern notation get fields out of *objects* as well, albeit mediated by methods. This allows script-writers to start out using records, and, if more exotic behavior is necessary, upgrade to objects, and the record-based code will generally continue to work.

4.4 Tables

The **table** type is a generalized map, or variation on a theme of a database table. Here's the `customers` table from Figure 3, which stores, for each customer, a name, profile, and other information:

```
customers = table(name) {
   val source, profile; var picked;
}
```

Tables have a statically-determined set of *columns* describing the information they hold (`name`, `source`, etc.), and a dynamically-determined set of *rows* expressing the information currently in the table. The columns are named by Thorn identifiers; the table can be regarded as a set of records whose fields are the columns of the table.

One or more columns, the *key(s)*, look like formal parameters to the table. The vector of keys determines a unique row of the table, and can be used as a subscript to get a row of the table as a record. So `customers(n)` looks up the customer whose key (`name` field) has value `n`. The following destructuring assignment lets us get the corresponding `profile` and `picked` fields (note that the pattern matching semantics of destructuring assignment allows us to pick out just a subset of the row fields if we wish):

```
{: profile, picked :} = customers(n);
```

Rows can be inserted by a method call:

```
customers.ins({: name, profile, source, picked:[] :});
```

or, using the key:

```
customers(name):={: profile, source, picked:[] :};
```

or deleted by key.

```
customers("Whale") := null;
```

This is semantically sound because the row of values being inserted with the previous operation is never null. In a table, **var** fields can be updated without requiring the whole row to be replaced:

```
customers(n1).picked := y.n2 :: picked;
```

Tables work nicely when one wishes to associate several pieces of information with one or more keys. The common case of maps or dictionaries, associating *one* datum to a single key, appears in many scripting languages. Thorn has syntactic sugar to let tables serve as maps. Programs using maps instead of tables frequently evolve to have several parallel maps, having in effect a poor man's table structure. So, Thorn maps are also tables; upgrading from one column to several neither requires introducing extra data structures nor breaks extant code using it as a map. A single non-key column of a table may be singled out by the **map** keyword.

```
m = table(k){map var v; val a; };
```

Subscripting such a table with *brackets* gets the **map** field. The usual subscripting with parentheses still works, and still gets the whole row:

```
m(1) := {: v:2, a:3 :};   #(*)
# m[1] == 2
# m(1) == {: k:1, v:2, a:3 :}   (*)
m[1] := 22;
# m(1) == {: k:1, v:22, a:3 :}  (*)
```

Adding a new column `b` to `m` would require just the code which inherently mentions all columns, `(*)`'ed above. The other lines, which work with just the map field, would not need to be changed.

The construct **map()** constructs a table with key `k` and a **map var** field `v`. (`k` and `v` are hardwired into this construct.) This, plus the subscripting-with-`[]` syntax, provides the familiar maps and dictionaries of many scripting languages as a special case of a more powerful table construct. Furthermore, at need, **map()** can be expanded into a table declaration with more columns.

4.5 Ords

Thorn lists are immutable, which is the right behavior for most situations. However, it is sometimes useful to have mutable ordered structures. Thorn provides the **ord**, short for "ordered table". Like tables, ords have many fields, one of which can be designated by **map** to be of particular interest. Like tables, **ord()** produces an ord of a fixed simple structure, which is often a good place to start.

```
primes = ord();
for (i <- 2 .. 12)
  if (i.prime?) primes @= i; #add a value to end
```

`primes @= i` adds a new row to the ord, with the map field set to `i` and all other fields (if any) **null**. As with tables, adding a new field to an **ord** is straightforward, and obviates the need for constructing parallel lists of related data. A row can be added as follows:

```
emails = ord{ var name, address };
for ( {: name, email :} <- dataset)
  emails.add( {: address:email, name :} );
```

Ords allow integer subscripts, and can be mutated via them:

```
emails(0) := null; # delete first entry
emails(2).name := "Kim"; # modify field of second
```

4.6 Queries

Thorn has a constellation of *queries*, which encapsulate a variety of common patterns of iteration, decision, selection, and grouping over collections. They are inspired by list comprehensions, higher-order libraries, and database queries. All of these could be done by the loops and conditionals you have written ten thousand times. Encapsulating them as queries makes them more convenient and less error-prone, and may introduce optimization opportunities.

The queries all use the same set of *controls*, which determine how iteration will proceed and what values are bound to what. Controls inspire iteration (**for**), or filter it (**if**, **while**), or manipulate bound values (**var**, **val**). We start with list comprehensions. To compute the list of squares of primes up to 100:

```
%[ n*n | for n <- 1..100, if prime?(n)]
```

The % symbol is used to distinguish queries; operations with [] produce lists, {} tables, and () arbitrary data. The **while** query control stops iteration altogether when a test becomes false, and **val** allows simple bindings. A crude primality test can be defined as:

```
fun prime?(n) = %some( n mod k == 0 |
                for k <- 2 .. n-1, while k*k <= n)
```

The **%sort** query returns a list sorted on some keys. %< keys are sorted in ascending order; %> in descending. To sort some people by last name, and, for those sharing a last name, by first name:

```
%sort[ p %< p.lastname %< p.firstname
     | for p <- some_people]
```

The **var** query control allows pseudo-imperative accumulation of partial results, like reduce in Common Lisp or fold_left in ML. **var** sum:=0 **%then** sum+n means that sum is zero before the first iteration, and sum+n on each successive iteration. The **%after** query returns a value from after the last iteration. (Iteration-bound variables like n are not available in the result, but **var** variables are.) The sum of a list can be computed as:

```
%after(sum|for n<-L, var sum:=0 %then sum+n)
```

Some operations do not make sense without at least one iteration; *e.g.*, computing the maximum of a list. **%then1** handles this case; on the first iteration, m is bound to the first value of n, and on later ones, to max(m,n):

```
%after(m | for n <- L, var m := n %then1 max(m,n))
```

Dually, the **%first** query returns a value from the *first* iteration. This is useful in searching, and so it can also be written **%find**. It throws an exception if no value is found, or, with an additional clause, can be given a value to return instead. To find the first record in custs whose src field is snd, one can use:

```
row = %find(r|for r <- custs, if r.src == snd);
```

This query has a statement form, written **first** or **find**:

```
find(for r <- custs, if r.src == snd) {
  println("Found $r");
}
else { println("Not found");  }
```

This kind of search happens quite frequently, and it would be a shame if we had a powerful pattern matching idiom but couldn't search a list for an element matching a pattern. **for** has two alternatives. The form we have seen before is **for** [fst,snd]<-pairs, which will throw an exception if any element of the list doesn't match the pattern [fst,snd]. The searching form uses <∼ rather than <-, and simply skips non-matching elements. So finding the first record in custs with src field equal to snd can be written:

```
row = %find(r | for r && {:src:$(snd):} <∼ custs);
```

Boolean quantifiers **%every**, **%some** and integer quantifier **%count** can detect whether a predicate is always true, sometimes true, or count how often it is true. To tell how many friendships there are between A and B:

```
%count(a.likes?(b) | for a <- A, for b <- B)
```

To produce a table giving several pieces of information about each of several things, use **%table**. The syntax echoes **table**, except that all fields must be initialized. The following produces a table of elementary arithmetic operations:

```
%table(x = x, y = y){
  sum = x+y; prod = x*y; diff = x-y;
| for x <- 1 .. 100, for y <- 1 ... 100}
```

As database programmers discovered long ago, it is often useful to aggregate information: GROUP BY in SQL; **%group** in Thorn. A characteristic use of this is to split a list based on whether or not the elements satisfy a predicate. The following code computes the lists of primes and composites up to 1000 in a single pass, and their number and their root-mean-square, declaratively. The common special cases **%count** and **%list** X evaluate to the number of items in the group and the list of values that X takes. **%first** F **%then** T **%after** A allows a fairly general accumulation of values, F for the first one, and T for later ones, and then uses A (if supplied) to provide the final result. In this case, rms is used as an accumulator to calculate the sum of squares in the **%first** and **%then** clauses, and converted to the actual RMS by the **%after**.

```
prime_or_not = %group(prime=n.prime?()) {
   map numbers = %list n;
   number = %count;
   rms = %first n*n %then rms+n*n
         %after sqrt(rms / n);
   | for n <- 1 .. 1000;};
primes = prime_or_not[true];
rmsComposites = prime_or_not(false).rms;
```

4.7 Equality and Identity

The intent of *equality* is that `a==b` means that `a` and `b` have the same state at the moment. The == operator is provided automatically for built-in data types and for *pure* classes (see Section 5.4) to check structural equality, and is simply not defined by default on other classes (but can be provided by users). Object *identity* is a different matter, and, unlike Java, for example, Thorn tries to avoid confusing identity and equality. Objects can get a VM-local identity by extending the system class `Identity`. `Identity` defines a unique identity for all objects that extend it. The value of the identity field is unspecified. However, `Identity` implements the method `a.same?(b)` that returns true iff `a` and `b` are the same object. For distributed applications, the class `GlobalIdentity` gives an object a globally unique identifier. The reasons for splitting global and local identity into separate concerns are purely pragmatical. *Guaranteeing* globally unique ids is tricky and may degrade performance. Most objects will only need local identity, and the refactoring step for a class that needs to evolve into a distributed system is a simple one.

5. Concurrency in Thorn

Scripting languages are heavily used for running web sites and similar Internet programs. Quite large businesses and organizations implement massive amounts of web functionality in Ruby, Python, or Perl, none of which was designed with web programming as a primary initial goal. Thorn focuses on distributed and concurrent computing.

In the distributed setting in particular, entities running at the same time are, potentially, physically and logically quite separate. They can, potentially, fail independently: it is quite normal for two processes to have a conversation and one of them to fall silent for seconds, or for eternity.

5.1 Components

Conceptually, each Thorn process—called a *component* to avoid confusion with, say, Java threads or operating system processes—has a single strand of control, and a private data store isolated from all others. Objects are not shared, which eliminates the need for locking locally and cache-updating algorithms globally. Processes communicate exclusively by passing messages. The values communicated must be immutable; they can be, *e.g.*, strings, lists, records, and objects of certain classes which are immutable by construction.

If objects of user-defined class are transmitted, the receiver must have that class available. Values constructed entirely from built-in types (records, lists, tables, strings) are universal, and will be understood by any Thorn component. Indeed, they can often be understood outside of Thorn. They correspond roughly to the values that can be transmitted via the JSON protocol [26], and should, provide a convenient way to communicate with any JSON-compliant program, Thorn or not.

The **component** construct defines a kind of component, rather the way **class** defines a kind of object. Components can have local **var** or **val** data, as well as functions (**fun**) and communication operations (**sync** and **async**). They have a **body** section, giving code that they are to run when started. For example, a parallel Life program with many worker threads could have the general structure:

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

A component can be spawned, alarmingly enough, by the **spawn** command:

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

spawn returns a handle to the component, which can be used to communicate with it. Since many components are one-offs, **spawn** can take a component body as well, and indeed this is how it is most often used:

```
c = spawn {
  var region;
  async workOn(r) {region := r;}
  ...# as above
};
```

5.2 High-level Communication

Thorn provides two communication models. The high-level model provides named communication, which may require an answer (*synchronous*, keyword **sync**) or not (*asynchronous*, keyword **async**). The syntax for high-level communication parallels that of methods, as shown in Figure 5.

```
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; }
}
```

Figure 5. High-level Communications.

However, communication differs from method calls in several important respects—the first one being that sending a message can in general have higher latency than performing a method call, and, indeed, may never return at all even when it ought to—so the syntax makes it very clear which one you are doing. Asyncs use `comp <-- m(x)`, and syncs use `comp <-> m(x)`, or, if they are to tolerate the failure of the peer to answer quickly,

```
comp <-> m(x) timeout(n) { dealWithIt(); }
```

The arrows are intended to suggest both the duration and direction of the communication.

The receiver has control over when it accepts communication. The **serve** statement causes a component to wait for a *single* high-level communication event. **serve** has an optional `timeout(n){...}` clause in cases where waiting indefinitely for a message to come in is undesirable. A typical reactive component will have a skeleton of the following form:

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

The high-priority `quit()` allows the outside to stop the component any time it is between doing real things. It is *not* an interrupt. **serve** simply looks for high-priority communications before lower-priority ones.

Often it is desirable for a client to send a synchronous request to a server (`response=server<->command()`), and for the server to call upon a worker component to actually do the computation. With the model as presented so far, this does not work properly: the server would need to have code:

```
sync command() {
  srvResponse = worker <-> subcommand();
  srvResponse;
}
```

and, by that code, the server would need to wait for the worker to respond. Thorn thus allows *split sync*, allowing the server thread to pass responsibility for answering the request to the worker. The server code is written:

```
sync command() envelope e {
  worker <-- subcommand(e);
  throw splitSync();
}
```

where `splitSync()` is a library function understood by the server's **serve** command as an instruction not to send a response to `command`. The **envelope** `e` clause captures the message together with its metadata in variable `e`; the worker will need the entire envelope to respond.

The worker's part of the split sync is:

```
async subcommand(e) {
  workerResponse = ... # compute it
  syncReply(e, workerResponse);
}
```

`syncReply` is a library function that sends `workerResponse` as reply to the message with envelope `e`. This is transparent to the client. The client uses the same synchronous call, `server<->command()`, regardless of whether the server's evaluation of `command` is ordinary or split.

5.3 Low-level Communication

The low-level communication model allows sending unadorned values from component to component. The statement `c<<<v` sends value `v` to component `c`. The value simply goes into `c`'s mailbox, a list in `c`'s underlying data structures holding messages which `c`'s code has not yet actively seen. A component can retrieve values from its mailbox by the **receive** statement, a variation on Erlang's, which scans through the mailbox, looking for the highest priority message that matches one of a set of patterns. When it finds one, it executes the corresponding code block, much like a **match**. For example, the following looks for an emergency stop message anywhere in the mailbox, and stops if it finds one. If there is none, it looks for a message asking it to post some data, or scan for data with some parameter `p`; their priorities default to zero. If none of those three is present, and none arrives within ten seconds, it marks itself bored instead.

```
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;}
}
```

5.4 Pure Values and Messages

Allowing the communication of complex values between components and computers raises some troublesome issues. Consider the following Thorn fragment:

```
var x := 0;
class Counter { def incr() { x++; } }
c <<< Counter();
```

This code is problematic, since the class `Counter` has a reference to mutable sender-local state, which is inaccessible on the receiving end. It could be cloned, but that introduces a number of complexities. Cloning is not sensible for all objects, arguably including `Counter`: if the intent of a single `Counter` is to maintain a universal count, a clone will not suffice. Even when cloning *is* sensible, there are subtleties: *e.g.*, sending a single `Counter` to `c` twice should, arguably, result in `c` getting two references to a single clone. The situation is not much better in the following case:

```
module M {
  n = 10000.rand();
  class RandRef { def mn() = n; }
}
c <<< RandHolder();
```

While `M.n` is immutable, the sender's and receiver's values will likely differ. One can make a good case that `r.mn()` should return either the sender's or the receiver's value.

Thorn avoids all such troublesome issues, in the same way that it avoids subtleties of multiple inheritance. We restrict communication to *pure* values, which cannot exhibit them. Atomic constant data types (*e.g.*, numbers and strings) are pure. Lists and records whose fields are all pure are pure. Tables are *never* pure; they are inherently mutable. Classes may be annotated as `pure`, which makes their instances pure. Dynamic and static checks ensure that instances of pure classes cannot refer to external variables or anything impure, and cannot have mutable state.

Pure classes have constant data. However, they can be *given* mutable data to operate on, and they can use local variables. They can thus formalize a wide variety of computations, albeit with the mutability supplied by the outside or localized within a method call. The following bit of highly imperative code adds up many values of `f(i)`; when it is applied, it inserts all the values of `i` to the ord `accum`, as well as returning the sum of their squares.

```
class C: pure {
  def sum3np1(f, n) {
    var sum := 0; var i := n;
    while (i != 1) {
      sum += f(i);
      if (i mod 2 == 0) i := i div 2;
      else              i := 3*i+1;
    }
    sum;
  }
}C
# ...
accum = ord();
fun fAdd(i) { accum @= i; i*i; }
c = C();
someone <<< c;
c.sum3np1(fAdd, 100);
```

In addition to checking the purity of data that are to be sent as messages, any variable or field references within a component body and which are defined outside its scope must be checked for purity before the component is spawned. Like message data, all such captured references must be (effectively) copied after the component is spawned to ensure isolation from the spawning component.

6. Thorn for Programs

Thorn is intended to support writing large programs, initially as untyped prototype scripts which can then be packaged into reusable modules or incrementally hardened by addition of annotations. Thorn's syntax supports adding arbitrary *constraint annotations* to declarations. As with Java annotations, these are intended to be used by compilers and tools to facilitate static analysis and optimization. We have explored a similar system in previous work [36], and plan to add it to Thorn in the future.

6.1 Types and Like Types

Dynamic typing, sometimes called duck-typing, is flexible and allows code to operate on objects as long as the necessary behavior is implemented and returns compatible results. In Thorn, the type **dyn** (for dynamic) is assumed as default (and never written explicitly). At the other extreme, Thorn supports *concrete types*, as used in statically typed programming languages. A variable of a concrete type `T` is guaranteed to refer to a value of that type (or a subtype). Concrete types in Thorn use nominal subtyping. When the programmer declares a variable as `int`, the Thorn compiler can use a 32-bit word and integer JVM bytecodes rather than a boxed integer and method calls, and similarly for other primitive types. While concrete types help with performance and correctness, they introduce restrictions on how software can be used and make rapid development more difficult; scripting languages do not favor them.

As an intermediate step between the two, we propose *like types*, getting some of the safety of concrete types while retaining the flexibility of dynamic types. Concrete types for `var x:T` or `fun f(x:T)` are used in two main places. At a method call `x.m()`, a static type check ensures that `x` actually has an `m` method. At a binding or assignment, like `x := y;` or `f(y)`, a static type check can ensure that `y`'s value makes sense to assign to `x`, can reject it entirely, or can inspire a dynamic check. Like types, `var x: like T` or `fun f(x:like T)`, give the expressive power of concrete type checks on method calls, but do not constrain binding or assignment. They do require runtime checks and thus may cause programs to fail with runtime type errors: sometimes fewer and never more than dynamic types do.

Like types provide three specific advantages over dynamic types.

1. **Documentation**: `fun f(p: like Point)` explains to humans how its argument can be used.
2. **Code Understanding:** Inside `f`, a programming environment could provide code completion on `p`'s methods.
3. **Error Detection:** Some errors can be detected at compile time. *E.g.*, typing `p.mvoe(q)` rather than `p.move(q)` will be caught, if `Point` has a `move` but not a `mvoe` method.

Like types provide nearly all the advantages, and nearly all the other flaws, of dynamic types.

A variable declared to be of type `like T` is constrained by the compiler to have methods invoked on it according to the interface of `T`, but at runtime may refer to any value. For

example, assuming a class Point with x(), y() and move(), like types allow us to type the parameter to the move() method thus:

```
class Point(var x, var y) {
 def x() : int = x;
 def y() : int = y;
 def move(p: like Point) {
   x := p.x();
   y := p.y();
   # p.hog();    would raise compile time error
 }
}
```

Declaring variable p to be **like** a Point makes the compiler check all method calls on that variable against the interface of Point. Thus, p.hog() would be statically rejected since there is no method hog in Point.

As assignments and bindings to like-typed variables are unconstrained, Thorn must check at runtime that the methods called are actually present. p.move(**true**); would pass the static check—because there *is* no static check on binding of like types, even the binding of actuals to formals in a method call. The call p.x() would fail for p==**true**, though our error message could be somewhat more detailed than simply "method not understood".

Like-typed formals, like dynamically-typed formals, are only constrained to have the methods that are actually used. Consider the following Thorn singleton object:

```
origin = object {
  def x(): int = 0;
  def y(): int = 0;
}
```

origin has no move() method. It implements the *relevant* parts of Point's protocol for move() to run successfully—x() and y(). So p.move(origin) works fine, just as with dynamic typing. It would not pass the static type check if move() had used concrete rather than like types.

The method move() even works with an untyped Pair

```
class Pair(x,y) {
  def x() = x;
  def y() = y;
}
```

In this case, the run-time return value of x() and y() would be type-tested against their explicit concrete type int, so p.move(Pair(1,2)) works, but not p.move(Pair("f", "b")). If x() and y() in Point were typed **like** int, checking the return type would not be necessary as assigning to a like type always succeeds.

An important difference between like types and gradual typing systems like that of [44], is that code completely annotated with like types can go wrong due to a run-time type error. On the other hand, a code completely annotated with concrete types will not go wrong.

6.2 Modules

A language should provide some form of information hiding, to encapsulate the design decisions that are likely to change. This minimizes redundancy, and maximizes opportunities for reuse. Unfortunately, most object-oriented languages use classes as their main encapsulation mechanism, which does not scale well. Thorn's module system, on the other hand, provides a way to encapsulate, package, distribute, deploy, and link large bodies of code. The core of the Thorn's module system is inspired by the upcoming Java Module System [51, 52]. In line with our previous work [49], we: (i) use a more intuitive and expressive name resolution, where modules become robust against interface evolution of the modules they import; and (ii) allow users to control the sharing and isolation of module instances.

Namespace control & robustness. To promote rapid prototyping, all module members are exported by default. Members can be hidden by declaring them private, and members imported from other modules can be re-exported by declaring them public. This localizes the influence of a single module, which is essential for scalability. Figure 6 shows a module M, which imports modules N under its own name N, and O under the name S. M also defines a class A and a value n, both of which are exported by default, and a variable x, which is hidden from module's clients. M also re-exports S.C, but cannot re-export N.A.

The names and aliases of members must be distinct from each other, as must exported and re-exported names; therefore, re-exporting N.A from M would clash with M.A. These two conditions guarantee that any non-fully-qualified name can be disambiguated, and that a fully-qualified name (*e.g.*, N.A) is never ambiguous.

The name resolution algorithm first checks within the module's own namespace, and only then in the exported namespaces of imported modules. For example, in Figure 6, A resolves to a local definition, even though N exports A. Suppose that M defines B; then, if N and/or S started exporting B, B would still resolve to the local definition—this is a useful form of robustness. Now suppose that M does not define B, and that B is only exported by N; then, if S starts exporting B, too, a compile-time ambiguity error occurs—a way to

```
module M {          # this module is named M
 import N;          # shared instance of N
 import own S = O; # own instance of O as S

 class A extends B {}  # B must come from M xor N xor S
 val n = A();       # A is M.A
 var x: private = N.A();  # x not exported

 public S.C;        # re-exported as M.C
 #public  N.A;     # ERROR: M.A already exported
}
```

Figure 6. Thorn Modules: A simple example.

protect against such breakage is to use fully-qualified names (or their aliases) for names that resolve within imports.

Sharing vs. isolation. In a single Thorn runtime, we can have multiple module instances of a single module definition. There can be a single shared module instance for each component, and as many non-shared instances as required. (Nothing, of course, is shared *between* components.) The statement `import N;` imports the component-wide instance of `N`, while `import own O;` creates a non-shared instance. This approach allows the developer to share module instances when required, and to have multiple clients that rely on conflicting invariants of a single module definition coexisting within a component [49].

In the following example, `A` and `B` refer to the shared instance of `NN`, and thus `A.n` and `B.n` are the same variable. `C` and `D` are distinct non-shared instances, and hence `C.n` and `D.n` are different variables from each other and from `A.n`:

```
module NN { var n := 0; }
import A = NN;      # shared instance
import B = NN;      # shared instance
import own C = NN; # non-shared #1
import own D = NN; # non-shared #2
A.n := 1;
B.n := 2;
C.n := 3;
D.n := 4;
# A.n == B.n == 2
# C.n == 3 && D.n == 4
```

Module instance state. Thorn *classes* cannot define static state. However, Thorn *modules* can define their own state, *e.g.*, `M.n` and `M.x` in Figure 6. Classes within `M` can use module-level variables in the ways that Java classes use static state. Each module instance has its own state; this is the main differentiator between module instances. Classes from different modules instances are incompatible because their invariants rely on different state.

7. Extensibility

A key design goal of the Thorn compiler is to support language evolution by allowing the syntax and semantics to be extended through plugins into the runtime system. The architecture facilitates construction of domain-specific languages based on Thorn. Unlike Java annotations, syntactic extension in Thorn is not limited to annotations on class or method declarations; arbitrary syntactic extension is permitted. This allows developers the freedom to support natural domain-specific extensions of the core language without having to wedge these extensions into a restricted syntax. Plugins provide a semantics for new syntax by translation into the core language. Semantic extensions can support additional static analyses (*e.g.*, taint checking) or code transformations.

7.1 Plugins

The runtime system provides a number of hooks for extension. Plugins may add new abstract syntax tree (AST) nodes and new passes over the AST. Plugins may also extend the parser and other passes over the AST to support new syntax or new semantics. Passes added by the plugin may perform static analysis or code transformations, including translating extended abstract syntax into the base Thorn language before evaluation. When the Thorn runtime starts up the initial component, it loads the *bootstrap class*, which can loads one or more plugins. Plugins can be installed by initializing the runtime with an overridden version of the bootstrap class. The loaded plugins are composed and then applied to the code in the component. Other components may be spawned with different sets of plugins. After plugins are loaded for a given component, parser extensions are composed into a single parser for Thorn code loaded into the component. Once an (extended) AST is constructed, plugins run their analysis and transformation passes on the AST, ending with a base language AST. Plugins add new passes to the compiler by registering a set of *goals* with the compiler's pass scheduler. A goal specifies prerequisite goals and a pass to run to satisfy the goal. The scheduler ensures that a goal's prerequisites are satisfied before the goal's pass is run. Plugins also hook into the existing passes such as name resolution. This core language AST is compiled to Java bytecode and evaluated or compiled to disk for future execution.

7.2 Syntax extensions

The base Thorn compiler provides a parser that plugins can extend to implement syntax extensions. Parser extensions are specified using a domain-specific extension of Thorn, itself implemented as a plugin, that supports parsing expression grammars (PEG). Parsing is performed by a packrat parser [19]. Plugins export productions and semantic actions for constructing ASTs. When the plugin is loaded, these new productions are composed with the current parser to create a new extended parser. The parser plugin translates the PEG grammar specification into a Thorn class that implements the parser. Since packrat parsers are scannerless, plugins can define their own tokens. Thorn's packrat parser supports left recursive rules using Warth et al.'s algorithm [59], overcoming one of the limitations of PEG grammars and simplifying development of parser extensions. Plugins may freely define new AST node classes and have the parser generate them. AST classes implement a visitor interface to interoperate with compiler passes. Plugins currently cannot extend the compiler itself to support new AST nodes; rather, these are translated into core language nodes for the compiler subsequently to evaluate or are compiled to disk.

7.3 The Assert Plugin

Figure 7 shows the complete code for a plugin that adds an `assert` construct to Thorn. The plugin is implemented

```
object AssertPlugin extends Plugin {
  def parser(_next) = AssertGrammar(_next);

  class Assert(e,m) extends Exp { }

  class AssertGrammar(next) extends
      Delegate(next), thorn.grammar.GrammarUtil {
    rule Exp = 'assert' Exp ':' String
               { Assert(Exp, String) }
             / 'assert' Exp { Assert(Exp) }
             / next.Exp;
    rule 'assert' = "assert" Spacing? { "assert" };
    rule Keywords = "assert" / next.Keywords;
  }

  def goals(unit) = [AssertGoal(unit)];
```

```
class AssertGoal(unit) extends
    Goal("Assert"), plugin.DesugaringGoal {
  def prereqs() = [];
  def run() =
    unit.setAst(unit.ast().accept(AssertDesugarer()));
}

class AssertDesugarer() extends Visitor {
  def visit(n) {
    match (n.rewriteChildren(this)) {
        Assert(e,m) => @'(unless(@,(e)) throw m)
      | m => m
    }
  }
}
} # end of object AssertPlugin
```

Figure 7. An assert plugin. It adds the statements **assert** <exp> and **assert** <exp> : "Message" to Thorn.

as a singleton object called `AssertPlugin`, extending the `Plugin` class. The plugin's functionality is provided by several nested classes. The `AssertGrammar` class defines the syntax of the new construct. The `parser` method of the `Plugin` class is overridden to return the new grammar definition. The new grammar will be composed with the existing grammar. By overriding the rules of the existing grammar the plugin can redefine the syntax, much like ordinary overriding of methods when subclassing. Normally an overriding rule will add a delegation call to the overridden rule `next.Exp` as its last case, which calls the `Exp` rule of the preceding plugin. The plugin also overrides the `Keywords` rule to add the `assert` keyword. The `Assert` class implements the new AST node that is returned from the semantic actions. The AST node inherits methods for interacting with visitors.

A plugin extends the compilation process by defining goals that, in turn, run passes over the AST. The goal may have prerequisite goals required to be met before processing the plugin's goal. In this example the plugin has no opinion on the order and just returns an empty list from its `prereqs` method. The `AssertGoal` class creates an `AssertDesugarer` visitor and applies the visitor to the AST. The desugarer pattern matches on the method argument. If the argument is an `Assert` then the assertion expression `e` and the associated message `m` are extracted and used to construct a semantically equivalent **`unless`** expression. The new expression throws an exception, carrying the message, if `e` does not hold. The `@'` and `@,` expressions are meta-operators (also implemented by a plugin) used to generate the AST for the **`unless`**, a feature influenced by Scheme.

8. JVM Implementation

Thorn's interpreter, like many interpreters written in Java, is not particularly fast. The performance of the compiler is closer to what we would expect from a production language. This section describes some of our choices in implementing the compiler.

Method Dispatch. Thorn sports several features that, at face value, are incompatible with the Java object model and the JVM. The most striking difference between Java and Thorn is that Thorn is dynamically typed while the JVM requires type information to do method lookup and dispatch. We use the same solution as JRuby [25] and Jython [27], generating a *dispatch interface* for each method that is implemented by all classes that has a method with a matching signature. For example the method **`def`** `foo(x)` will generate an interface `Ifoo_1` declaring a method `IObject foo(IObject)`. At the call site it is determined, with an `instanceof` instruction, that the receiver actually implements the called method before calling the method with an `invokeinterface` instruction. The `instanceof` is used for error reports when a message is not understood by the receiver.

Multiple Inheritance. Thorn's multiple inheritance conflicts with Java's model of single inheritance plus interfaces. As a consequence, Thorn inheritance cannot be implemented as Java inheritance and Thorn method calls cannot simply be implemented as JVM method calls; the JVM has the wrong notion of supercall, and does not know which of several parent JVM classes contains the code of an inherited method. Instead, every Thorn method in every Thorn class gives rise to an instance method in every Thorn class that inherits it, and a static method in the defining class in the JVM. The instance method simply calls the static method and returns the result. The static method contains the actual compiled body of the Thorn method. (We use a single static method and an extra method call to avoid massive duplication of code, and to enable separate compilation.) As mentioned earlier, Thorn forbids dispatch ambiguities, so there is no need to search for the right method at runtime.

Field Access. Field access is done via Java method call. Fields inherited from Thorn superclasses are redeclared in the implementations of child classes (which do not inherit from the implementations of superclasses). Furthermore,

due to dynamic typing we cannot statically reject programs that reassign **`val`** fields. Our simple solution is to have the setters of **`val`** fields throw an exception.

Static analysis. Thorn employs a simple local type inference to optimize the bytecode instructions generated for operations on primitive data types. Operations on variables of primitive data types, and arrays of integers and floats, are translated into primitive operations on unboxed primitives for speed. Such optimization is only possible locally.

Components. The Thorn compiler provides process isolation by providing separate copies of global values for each component. The scheduler is written in Java and uses a pool of worker threads that calls the run method of each component with the next message in the message queue. The **`spawn`** keyword creates a new component and registers the component with a worker thread in the scheduler. Components will currently not be preempted, although we have discussed merging Thorn with a tool such as Kilim [47] that rewrites bytecode into CPS to allow preempting a component schedule another in the same thread. To minimize context switching, a component lives inside a specific worker-thread. The scheduler however implements a work-stealing algorithm which may cause components to be moved from an overloaded worker thread to an idle thread. Consequently, unless components go into infinite loops, a Thorn program can spawn a very large number of concurrent components without undue overhead.

Java Integration. Compiled Thorn programs can create and use Java objects and invoke Java methods. This functionality is vital for interacting with the system. Java objects and classes are exposed to Thorn through the wrapper classes `JavaObject` and `JavaClass`. Thorn objects passed to and from Java code are automatically wrapped and unwrapped. Java's strings and primitive types are mapped to their Thorn counterparts rather than being wrapped. Fields are accessed through `get` and `set` methods of `JavaObject`, which take the name of the field. Methods are accessed through the `invoke` method of `JavaObject`. There are two versions of `invoke`. The first version takes the method name as a string, a list of actual arguments and a list of types, represented by instances of `JavaClass`. The type list is used to resolve which method to invoke if the method is overloaded. An exception is thrown if the method cannot be resolved. The other version of `invoke` elides the type list and instead uses the run-time types of the actual arguments to disambiguate the method.

Performance. Although the prototype Thorn bytecode compiler was not designed with performance as a primary goal, its performance is comparable to Python and, with the aid of optional type information, runs several times faster. We translated several benchmarks from The Computer Language Benchmark Game [16] into Thorn and timed their runtime and compared the result existing benchmarks running on Python 2.5.1 and Ruby 1.8.6. The result is shown in Figure 8.

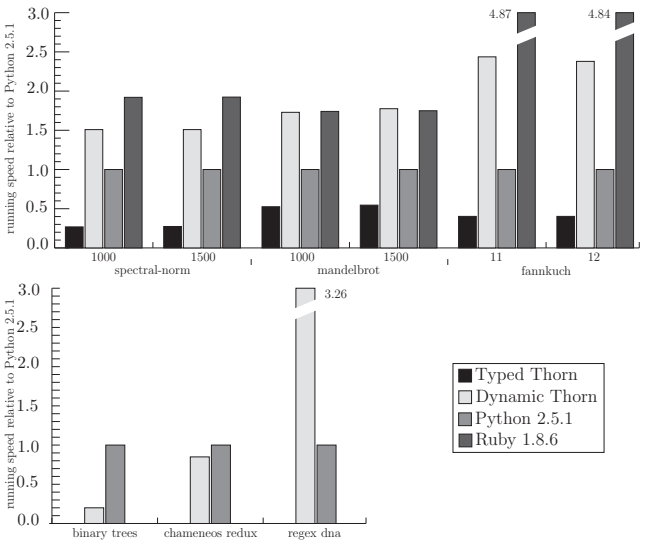

Figure 8. Performance of Thorn (bytecode compiler) in relation to Python 2.5.1.

Several conclusions can be drawn from this experiment. Thorn's support for the basic features of object-orientation (such as fast method dispatch), which are heavily stressed by the implementation of the binary-trees and chameneos-redux benchmarks, is quite efficient. With several benchmarks, including Regex-dna, the speed of underlying libraries (here, Java's built-in regular expression facilities) not written in the host language is the dominant influence on running time, rather than the actual language implementation's. For primitive data types, wrapping of integers etc. in objects degrade performance, especially in long-running computation-intensive benchmarks like Fannkuch, unless a modicum of type information is added, which causes a significant speed-up: Typed Thorn runs spectral-norm, mandelbrot and fannkuch between 2x and 4x faster than Python and about 3x and 6x faster than dynamic Thorn. The Ruby implementation is the slowest by far and is outperformed by a factor 7x to 12x by Typed Thorn.

9. Conclusions

We have presented the design and implementation of Thorn, a new object-oriented language that supports the evolution of scripts into concurrent applications by striking a balance between flexibility and robustness. We have also shown that even without extensive optimizations, a prototype compiler for a significant subset of the language, built using an extensible plugin architecture, achieves competitive performance on a Java Virtual Machine. The dynamically-typed core of Thorn is designed to enable rapid and exploratory programming by dint of its succinct syntax and the presence of flexible aggregate data types such as tables. On the other hand,

classes, an optional type system, and an expressive module system provide the support needed for programming in the large. While Thorn is an imperative language, we encourage side-effect free programming to decrease program fragility by including immutable built-in types and value classes. Further, by isolating components, the Thorn concurrency model avoids problems generally associated with shared memory and lock-based synchronization. Few of Thorn's features are wholly novel in isolation. Our principal contribution is to combine these features in a balanced way to allow programmers to prototype applications using simple scripts, then modularize and annotate these scripts so that they can be composed into reliable applications.

References

[1] Erlang Reference Manual. `http://erlang.org/doc/`, 2008. Version 5.6.5.

[2] The Python Tutorial – Modules. `http://docs.python.org/3.0/tutorial/modules.html`, 2009. Version 3.0.1.

[3] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, Jr., and Sam Tobin-Hochstadt. The Fortress language specification, version 1.0, March 2008.

[4] Christopher Anderson and Sophia Drossopoulou. Babyj: from object based to class based programming via types. *Electr. Notes Theor. Comput. Sci.*, 82(7), 2003.

[5] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, October 2006.

[6] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 22, pages 430–440, December 1987.

[7] David F. Bacon. Kava: a Java dialect with a uniform object model for lightweight classes. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, pages 68–77, 2001.

[8] Kenneth Barclay and John Savage, editors. *Groovy Programming*. Morgan Kaufmann, December 2006.

[9] Kent Beck and et al. Principles behind the agile manifesto, 2007. `http://agilemanifesto.org/principles.html`.

[10] Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.

[11] Gilad Bracha and William R. Cook. Mixin-based Inheritance. In *Proceedings of OOPSLA*, volume 25(10) of *ACM SIGPLAN Notices*, pages 303–311. ACM Press, October 1990.

[12] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, August 2007.

[13] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264, New York, NY, USA, 1974. ACM.

[14] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, 2005.

[15] Clojure. `http://clojure.org/`.

[16] The Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/`.

[17] Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 273–298. Springer Verlag, 2007.

[18] Matthew Flatt and Robert Bruce Findler. PLT Scheme Guide – Modules. `http://docs.plt-scheme.org/guide/modules.html`, 2009. Version 4.1.5.1.

[19] Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '04)*, January 2004.

[20] Jesse James Garrett. Ajax: A new approach to web applications, February 2005. `http://www.adaptivepath.com/ideas/essays/archives/000385.php`.

[21] Niklaus Haldiman, Marcus Denker, and Oscar Nierstrasz. Practical, pluggable types for a dynamic language. *Computer Languages, Systems & Structures, ESUG 2007 International Conference on Dynamic Languages (ESUG/ICDL 2007)*, 35(1):48–62, April 2009.

[22] A. Hejlsberg, S. Wiltamuth, and P. Golde. C# language specification. 2003.

[23] Martin Hirzel, Nathaniel Nystrom, Bard Bloom, and Jan Vitek. Matchete: Paths through the pattern matching jungle. In *Practical Aspects of Declarative Languages (PADL 2008)*, pages 150–166, January 2008.

[24] Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.

[25] Java powered Ruby implementation. `http://jruby.codehaus.org/`.

[26] Introducing JSON. `http://www.json.org/`.

[27] The Jython Project. `http://www.jython.org/`.

[28] The Kawa language framework. `http://www.gnu.org/software/kawa`.

[29] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rmy, and Jrme Vouillon. The objective caml system, release 3.11. Documentation and user's manual, 2004. http://caml.inria.fr/pub/docs/manual-ocaml/.

[30] Barbara Liskov and John V. Guttag. *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, 1986.

[31] Donna Malayeri and Jonathan Aldrich. CZ: Multiple inheritance without diamonds. In *OOPSLA '09: Proceedings of the 24th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, October 2009.

[32] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706, 2006.

[33] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[34] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *12th International Conference on Compiler Construction (CC 2003)*, number 2622 in Lecture Notes in Computer Science, pages 128–152, Warsaw, Poland, April 2003. Springer-Verlag.

[35] Nathaniel Nystrom and Vijay Saraswat. An annotation and compiler plugin system for X10. Technical Report RC24198, IBM T.J. Watson Research Center, 2007.

[36] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *Proceedings of the 2008 ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, October 2008.

[37] Martin Odersky, Philippe Altherr, Vincent Cremet, Iulian Dragos Gilles Dubochet, Burak Emir, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Lex Spoon, and Matthias Zenger. An overview of the Scala programming language, second edition. Technical report, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[38] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer, 31(3):23 30*, 1998.

[39] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, 2000.

[40] J. H. Reppy and Y. Xiao. Specialization of CML Message-passing Primitives. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*, pages 315–326, 2007.

[41] Vijay Saraswat et al. The X10 language specification. Technical report, IBM T.J. Watson Research Center, 2008.

[42] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behaviour. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274, 2003.

[43] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag, New York, 1986.

[44] Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP 2007—Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 2–27. Springer Berlin / Heidelberg, 2007.

[45] Jeremy G. Siek. Gradual typing for functional languages. In *In Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[46] M. Sperber, R.K. Dybvig, M. Flatt, A. Van Straaten, R. Kelsey, W. Clinger, and J. Rees. Revised 6 report on the algorithmic language Scheme, 2007.

[47] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP 2008—Object-Oriented Programming*, volume 5142/2008 of *LNCS*, pages 104–128. Springer Berlin / Heidelberg, 2008.

[48] Guy L. Steele and Richard P. Gabriel. The evolution of Lisp. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 231–270, New York, NY, USA, 1993. ACM.

[49] Rok Strniša. Fixing the Java Module System, in Theory and in Practice. In *Proceedings of FTfJP*, pages 88–99. Radboud University, July 2008.

[50] Sun. *Core Java J2SE 5.0*. Sun Microsystems Inc., http://java.sun.com/j2se/1.5.0/, 2005.

[51] Sun Microsystems, Inc. JSR-277: Java™ Module System. `http://jcp.org/en/jsr/detail?id=277`, October 2006. Early Draft.

[52] Sun Microsystems, Inc. JSR-294: Improved Modularity Support in the Java™ Programming Language. `http://jcp.org/en/jsr/detail?id=294`, 2007.

[53] Don Syme. An upcoming experimental feature: Active patterns in F#, August 2006. `http://blogs.msdn.com/dsyme/archive/2006/08/16/activepatterns.aspx`.

[54] David Thomas and Andrew Hunt. *Programming Ruby: the pragmatic programmer's guide*. The Pragmatic Programmers, LLC., Raleigh, NC, USA, 2 edition, August 2005.

[55] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 964–974, 2006.

[56] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 395–406, 2008.

[57] Guido van Rossum and Fred L. Drake Jr., editors. *The Python Language Reference Manual (version 2.5)*. Network Theory Ltd, 2006.

[58] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, January 1987.

[59] Alessandro Warth, James R. Douglass, and Todd Millstein. Packrat parsers can support left recursion. In *Workshop on Partial Evaluation and Program Manipulation (PEPM '08)*, January 2008.