

Thorn—Robust, Concurrent, Extensible Scripting on the JVM

Tobias Wrigstad Johan Östlund
Gregor Richards Jan Vitek

Purdue University

Bard Bloom John Field
Nathaniel Nystrom

IBM Research

Rok Strniša

University of Cambridge

Abstract

Scripting languages enjoy great popularity due 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 notoriously hard to evolve and compose, and often perform poorly at run-time. An additional weakness of most current scripting languages is lack of support for concurrency—though concurrency is required by more and more applications for scalability on parallel architectures, for handling concurrent real-world events, or for interacting with remote distributed services.

This paper reports on the design and the implementation status of Thorn, a novel programming language targeting the JVM. Our principal design 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 constraint 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.

1. Scripting: The State of The Art

Scripting languages are lightweight, dynamic programming languages designed to maximize 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 which is strictly not a scripting language 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” [8]. 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++ [33, 35].

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 operating systems or 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 support concurrency at all, do so through complex and fragile libraries and callback patterns. *E.g.*, AJAX style web applications [17], which typically combine browser- and server-side scripts written in different languages, are notoriously troublesome.

Currently, the weaknesses of scripting languages are largely dealt with by rewriting prototype scripts in less brittle 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 (*e.g.*, due to pointers and memory management) and increases the num-

ber 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 code can be encapsulated into *modules* whose behavior can be understood without requiring detailed knowledge of the code, and which can be composed with other modules to build reliable applications.

Thorn has been carefully designed to strike a balance between dynamicity and safety. It does not support the full range of dynamic features commonly found in scripting languages, but does include programming-in-the-large and concurrency constructs. The result is a language “safe by design”—dynamic enough to be suitable for rapid prototyping, and static enough to facilitate reasoning and static analyses. Thorn runs on the JVM, which allows it to execute on a wide range of operating systems and lets it use lower-level Java libraries for the Thorn runtime and system services.

Thorn has the following key features:

Scripting core: Thorn is dynamically-typed. It includes powerful aggregate data types (lists, records, and tables) and first-class functions.

Object-orientation: Thorn includes a simple class-based multiple inheritance object model. The “diamond problem” (multiple inheritance of stateful classes) is avoided by a simple static restriction.

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

Immutable data types Thorn has a number of features that encourage the use of immutable data and pure functions. Minimizing side-effects typically eases code composition and evolution, and facilitates code optimization.

Concurrency: Thorn supports concurrent programming with lightweight, isolated, single-threaded *components* that communicate through asynchronous message passing.

Modules: Thorn has an expressive module system that makes it easy to wrap scripts as reusable components.

Constraint annotations: Optional *constraint annotations* on declarations for variables, etc., permit static or dynamic program checking, and facilitate optimizations.

Extensibility: Thorn is built on top of an extensible compiler infrastructure written in Thorn itself. This infrastructure allows for language experimentation and development of domain specific variants.

It is equally 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—*e.g.*, dynamically adding fields/methods to objects or classes.

Unsafe methods: Thorn can access Java libraries through an interoperability API, but there is no access to the bare machine through unsafe methods, *e.g.*, as in C# [18].

Java-style interfaces: Their function is largely subsumed by multiple inheritance, modules, and abstract class members.

Java-style inner classes: These are complex and difficult to understand, and are largely subsumed by simpler constructs such as first-class functions (closures) in Thorn.

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. Such coercions are troublesome to explain and maintain.

Implementation Status and Availability. The language described in this paper has been implemented in a prototype compiler that targets the JVM. The compiler currently implements the scripting core of Thorn (with the exception of tables/queries), key features of the concurrency model, Java interoperability, and several classes of constraint annotations. While the current Thorn compiler was not designed with high performance as its primary goal, preliminary benchmarks show that it achieves execution speed comparable to Python. Thorn is developed as a joint project between IBM Research and Purdue. We are currently in the process of open sourcing the compiler and libraries.

Targeted Domains. Thorn is aimed at domains including client-server programming for mobile or web applications, embedded event-driven applications, networked sensors and actuators, and distributed web services. The relevant characteristics include a need for rapid prototyping, for concurrency (for interaction and event management), 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 the Thorn language for specialized domains, but allow more advanced optimizations and better error handling than would be possible using mechanisms such as macros or preprocessors.

Outline. The remainder of the paper is organized as follows: We first give an overview of Thorn’s core features (Section 2), starting with a tiny script and moving on to a much larger concurrent application. Next, we compare and contrast Thorn with other languages (Section 3). We describe Thorn’s core scripting data types and control structures (Section 4), and then Thorn’s concurrency model (Section 5). Thorn’s key programming-in-the-large features—modules and constraint annotations are covered next (Section 6), followed by details on the design and implementation status of Thorn’s JVM compiler (Section 7). We close by reviewing our contributions (Section 8).

To the reviewers: A paper on the design of a gradual type system for of Thorn has been submitted to OOPSLA [50], while there is some reuse of terminology, there is no overlap in contributions.

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:

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

By default, Thorn scripts intended for command line execution are run in an environment containing (among other things) bindings for `argv`—a list of command line arguments, `exec`—a predefined function that executes a command in the shell, and a `println` function with the expected behavior. The `exec` function returns the string generated by the `cat` command and the `split` method splits it into a list of lines. The `for` loop iterates over the strings in the list and prints any line containing an instance of the second command line argument.

2.1 A Thorn Program

Fig. 1 shows a more complex Thorn program: a match-making application. The program is organized as a central `DateService` process that interacts with client processes acting on behalf of customers seeking dates. In addition to the concurrent interaction between client processes and the `DateService` server, the server itself uses internal concurrency: lightweight worker processes are spawned off dynamically as necessary to handle client requests.

Thorn source code is organized into *modules*. Fig. 1 contains two modules: `dater` and `dateDemo`. Modules are simply collections of bindings of names to classes, component bodies, variables, and so on. By contrast, a running Thorn application is organized into collections of lightweight concurrent *components*. Components are *isolated* from one another:

updating the state of one component cannot affect the state of another. Components communicate with one another via message passing.

Thorn components are generally organized as collection of *communication declarations* which define code to be executed upon receipt of messages. The `DateService` component defines five communications. The **sync** communications return a value to the sender; the **async** communications do not. An expression of the form

```
dateServer<->register("Kim", profile)
```

sends a message to a component via a *component reference* (here, `dateServer`), then awaits a reply (in this case, a string). Alternatively, a message can be sent *asynchronously* like this

```
dateServer<--makeMeAMatch();
```

in which case the sender continues on without waiting for a reply. If a message is sent asynchronously to a synchronous communication, the reply is ignored. If a synchronous message is sent to an asynchronous communication, it could wait indefinitely; however, a timeout annotation can be used to prevent blocking indefinitely. It is important to emphasize that modules and components are orthogonal concepts. To see this, note that the `dater` module defines two classes of components: a named `DateService` component, and an anonymous worker component that is spawned by the function `spawnWorker`.

Along with component declarations, modules may define classes and functions and initialize variables. The `dater` module contains the definition of a `Profile` class, which is used to encapsulate personal client information. The `dateDemo` module imports the code encapsulated in the `dater` module and spawns an instance of the `dater` component defined there (bound to the variable `dateServer`). The `dateDemo` module also spawns an anonymous instance of a representative client, which interacts with the server. We will cover of all of the Thorn constructs used in Fig. 1 in detail in subsequent sections; here we note a few highlights.

The `=` operation introduces new immutable name-value bindings. The statement

```
dateServer = spawn dater.DateService;
```

spawns a new date service and names it `dateServer`. Mutable variables are defined by `var`, and updated by `:=`.

```
var maxPos := 0;
```

Most scripting languages allow the assignment operation to create new variables as well as modify extant ones, which is convenient but error-prone. Thorn’s approach is nearly as convenient and far safer. For extra safety, 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.

Methods in classes are declared using the `fun` keyword. Freestanding functions (closures) defined outside classes

```

module dater;

# Base class for date profiles
class Profile(music, food) {
  # default compatibility tester
  fun compatible?(otherProfile) = false;
}

# Defines a component "template", does not instantiate
component DateService {
  customers = table {
    key pos; # keys are always immutable
    val source, name, profile;
    var picked := [];
  }
  var maxPos := 0;

  sync register(name, profile) {
    maxPos += 1;
    # add new record to table; 'sender' is bound to
    # sending component
    customers @=
      {: pos:maxPos,source:sender,name,profile :};
    "You have been registered!";
  }

  async makeMeAMatch() {
    try {
      seeker = %find(r | for r <- customers,
                    if r.source == sender);
      spawnWorker(seeker, me);
    }
    catch {
      _ => sender<--notRegistered("Not found")
    }
  }

  async foundMatch(sourcePos, matchPos) {
    x = customers(sourcePos); # asked for the date
    y = customers(matchPos); # the one who we picked
    x.source<--gotDate("You've got a date!", y.name);
    y.source<--gotDate("Congrats!", x.name);
    customers(sourcePos).picked := y.pos::x.picked;
    customers(matchPos).picked := x.pos::y.picked;
  }

  async noMatch(sourcePos) {
    x = customers(sourcePos);
    x.source<--noDate("Sorry, no match for you.");
  }

  # look up record with key 'pos' in table
  sync recordFor(pos) = customers(pos);
}

# spawns anonymous worker component
fun spawnWorker(seeker, master): Private = spawn {
  body {
    var i := 0;
    while (true) {
      i += 1;
      r = master<->recordFor(i);
      # Have we come to the end of the table?
      if (r == null) break;
      # Pick out the fields we'll need
      {: pos, profile, picked :} = r;
      # Seekers don't want to date themselves or
      # date the same person twice
      if (pos != seeker.pos &&
          ! (seeker.pos in picked))
        if ( profile.compatible?(seeker.profile)
            && seeker.profile.compatible?(profile)) {
          master<--foundMatch(seeker.pos, pos);
          break;
        }
      }
      master<--noMatch(seeker.pos);
    }
  }
}

#-----#

module dateDemo;
import dater;

# spawn instance of date service
dateServer = spawn dater.DateService;

kimClient = spawn { # spawn instance of client
  profile = object extends dater.Profile(
    ["hip hop", "j-pop"],["thai", "sushi"])
  { fun compatible?(Profile(m,f)) =
    (music.intersect(m) @ food.intersect(f)).len() > 1;
  }
  var happiness := 0;

  async noDate(msg) = happiness -= 1;
  async gotDate(msg, name) = happiness += 1;

  body { # Sign up...
    response = (dateServer<->register("Kim", profile));
    # Try to get a date!
    dateServer<--makeMeAMatch();
    # Wait for an answer, which will come as a signal
    serve;
  }
}
}

```

Figure 1. A small, but complete Thorn application. Note that the *source code* is organized into *modules* (here, `dater` and `dateDemo`), while the *running application* is organized into collections of concurrent, isolated *components* (here, `dateServer`, `kimClient`, and the collection of anonymous components created by `spawnWorker`). Modules and components are orthogonal concepts.

also use **fun**. Functions and methods always return a value, which is the last expression computed within the function body.

Thorn's built-in types include lists, records, and a powerful associative *table* type. The empty list is [], and `::` conses a value onto the head of a list. Records are constructed using expressions such as

```
{: pos:maxPos,source:sender,name,profile :}
```

In this case, `pos:maxPos` binds the variable `maxPos` to a record field named `pos`. When an identifier is referenced in a record constructor, the field name may be omitted (e.g., with `name` and `profile` above); this introduces a field with the same name as the identifier.

Conceptually, a table such as `customers` in Fig. 1 is simply a collection of records, or *rows* defined over a common set of field names, or *columns*. In tables like `customer` containing only a single **key** field, an expression of the form `customers(sourcePos)` retrieves the row whose key equals `sourcePos` (and raises an exception if there is no such row). Tables and lists both support a rich collection of *query* expressions, e.g., `%find`, which perform implicit iteration.

When a component is spawned, its default behavior is passive: it blocks awaiting an incoming message, executes the corresponding communication on receipt of the message, and repeats the process indefinitely. However, as in the `kimClient` component in Fig. 1, components may optionally specify a **body**, which is executed after a spawned component is initialized. The body may contain arbitrary Thorn code, but will usually include one or more **serve** statements, which have the effect of explicitly blocking to await an incoming message.

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.

The **Private** declaration on `spawnWorker` is an example of a *constraint annotation*. Constraint annotations are used for a variety of purposes and are discussed at greater length in Section 6.1; the **Private** annotation for `spawnWorker` indicates that the name is not visible outside the `dater` module in which it is declared.

3. Related Work

In the design of 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 the best aspects of these languages and extends them with novel variations on features such as lightweight concurrent components, optional constraint annotations, modules, and extensibility features.

The syntax of Thorn is influenced primarily by Scala [32] and Python [48]. Thorn supports a class-based object model, described in Section 4.1, similar to Java's or Scala's, but simpler to program. Like Kava [6] and X10 [37], Thorn provides value classes (Section 6.1.2), 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] and Scala extractors [14]. Thorn's pattern matching features are more directly derived from Matchete [19]. Thorn includes associative data types and queries which are inspired by the design of SETL [38], LINQ [28], Haskell's list comprehensions [22], and CLU iterators [27].

Thorn's module system is based upon the Java Module System [45, 46]. 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 [47] only focuses on code reuse with its mixins [10]. 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 [15] 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 [5, 32, 41], and languages like Concurrent ML [36], 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. Recent languages for high-performance computing such as X10 [37, 12] and Chapel [11] 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 [9]. The idea is to completely separate the run-time semantics from the type system. Type annotations, implemented in compiler plugins, serve only to reject programs statically that might otherwise have dynamic type errors. By contrast, Thorn plugins can perform arbitrary transformations of the program. Java5 [44] annotations adapt the pluggable type system idea to Java, which retains its original type system and run-time typing semantics. JavaCOP [4]

<i>atomic value</i>	boolean, numerics, immutable string
<i>record</i>	immutable set of named values
<i>list</i>	immutable vector of values
<i>table</i>	mutable associative aggregate
<i>object</i>	anonymous or class instance
<i>closure</i>	anonymous or named function
<i>component reference</i>	lightweight process reference

Figure 2. Thorn data categories.

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.

Thorn is built on an extensible compiler infrastructure, providing a compiler plugin model similar to X10’s [31]. 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 [30]. The meta-programming features and syntax extension features of Thorn were influenced by Lisp [42] and Scheme macros [40].

Thorn is one of many dynamic languages and scripting languages implemented on the JVM [25, 23, 7, 13, 26]. 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 [7], 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 Fig. 2. 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, or data created using the **object** construct. Class and component definitions (do not confuse the latter with component *references*) are *not* data in Thorn.

4.1 Classes

Some scripting languages have a very slippery (or, if you prefer, flexible) concept of “class”. Lua [20] 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 very much, their code is likely to be troublesome to understand or maintain. If they do not take

advantage of it, it is of no value. So, 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. In the scripting style, the syntax for classes is relatively lightweight. Members are private by default but can be made public by annotations.

Methods. Methods are introduced by the **fun** or **method** keywords and are called by the traditional `receiver.m(x,y)` syntax. As in Scala, single-expression methods can be introduced with a compact syntax, and methods whose bodies are larger can use `{}`-blocks.

```
class Rectangle(x1, y1, xh, yh) {
  fun well_formed?() = x1 < xh && y1 < yh;
  fun area() {
    if (this.well_formed?()) {(xh-x1) * (yh-y1)};
    else throw DegenerateRectangleException();
  }
}
```

Constructors Constructors look like function calls. They are defined with the **new** keyword. Within the constructor body (only), **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) }
}
```

And new pairs can be created by `Pair(1,2)`, or `Pair(3)`, where the latter has `snd=null`. Note that a `Pair` has two immutable (**val**) fields, which are bound in the constructor body. **val** fields must be bound exactly once inside a constructor, and cannot be assigned or bound otherwise.

A pernicious source of errors in Java is the ability of a constructor to leak references to the object being constructed, before it is finished. Other procedures or even other threads can stumble on uninitialized fields. Thorn largely eliminates this issue by disallowing the use of **this** inside constructors: they cannot refer to the object being constructed explicitly, and thus cannot pass it around. 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. `init`, like any method, can refer to **this**.

Defaults. 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 don't induce variables.

```
class FlavoredPoint(x,y,f1) extends Point(x,y) {}
```

produces a subclass of `Point` with one new field `f1`.

Multiple Inheritance. Multiple inheritance, in its full and grand generality, can be extremely powerful but extremely confusing. Thorn supports a simple form of multiple inheritance, designed to be reasonably straightforward to understand and to give some of the simpler advantages of multiple inheritance with only a modest amount of extra work. Thorn's multiple inheritance provides the power of traits. If Thorn were typed, it would provide the power of interfaces as well; but, with optional types, there is not much point to interfaces.

```
class Flavored(f1) {}
class FlavoredPoint(x,y,f1)
  extends Point(x,y), Flavored(f1) {}
```

The first knot that Thorn cuts is method precedence. If class `C` inherits from `A` and `B`, and `A` and `B` each provide a method `m()` but `C` does not, what is `C().m()` supposed to do? Should it be `A`'s `m()`, or `B`'s? Some languages have elaborate precedence rules to control this situation. Thorn simply forbids it. If two parents of a class `C` both define `m()`, then `C` must also define `m()`. Often the code for this definition will involve a supercall to one or the other parent's method, using the `super@A` syntax to say which to call:

```
class C extends A, B {
  fun m() = super@A.m() + super@B.m();
}
```

The second knot that Thorn cuts is multiple inheritance of mutable state. If class `D` has a `var` field `dd`, and `E` and `F` both extend `D`, and `G` extends `E` and `F`, then what sort of `dd` field do instances of `G` have? In some cases, it is desirable for the `dd` inherited from `E` to be the same as that from `F`; in other cases, they should be different. C++, for one example, has a subtle answer to this question. 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 fussing about constructing 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 can be built thus:

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

Each call to `makeCounter()` produces a separate counter object.

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 explain what structures are expected, and to disassemble them and use their parts. This recovers a certain amount of the protection that was lost by having an untyped language—and other protections that few type systems provide: pattern matching, unlike many type systems, can check “this list contains three elements” just as easily as “this list contains only integers”. It also provides a degree of convenience that any programmer will appreciate.

Most built-in types provide patterns that parallel their constructors. 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`. 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 types: `+p` matches a non-null entity that matches `p`. `$(e)` matches the value of the expression `e`, and `(e)?` matches if the boolean expression `e` evaluates to true.

The conjunctive pattern `p && q` matches an entity that matches both `p` and `q`. For example, `x && [h, t...]` matches a nonempty list, binding the whole list to `x`, the head to `h`, and the tail to `t`, thus behaving like ML's `as`. `&&` is more powerful than `as`, however;

```
[..., 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 eliminates the need for side conditions in pattern expressions.

Thorn also has disjunctive patterns: `P || Q` succeeds if either `P` or `Q` succeeds. For sanity, this is only allowed if `P` and `Q` provide the same bindings. Negative patterns are available as well: `!P` succeeds iff `P` fails; negative patterns provide no bindings whatever.

Extractors Classes can define *extractors*, pattern components that succeed or fail, and produce data on success. The

skeletal `Person` class defines an extractor that (a) tells if a person is married, and (b) if so, tells who the spouse is.

```
class Person {
  var spouse := null;
  pat married(to) {
    if (spouse == null) fail;
    else { to = spouse; succeed }
  }}
```

succeed and **fail** cause the match to succeed or fail. The formal parameters to `married` are *output* parameters, dual to the input parameters used elsewhere in Thorn. Their values must be bound on the paths to **succeed**, as with `to = spouse`; in this example. An attempt to match a person `p` against the pattern `married(q)` (with `q` a fresh variable) succeeds and binds `q` to the spouse, or fails. Side effects are not formally forbidden in **pats**, but they are a particularly bad idea there.

A particularly useful form of extractor is the *type test*, which Thorn produces by default for all classes. Matching against `Person()` checks whether an object is a person. A class with a default constructor also has a default extractor that is the inverse of the constructor:

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

`Point` can be used as an extractor. Matching `Point(1,2)` against pattern `Point(1,yy)` succeeds and binds `yy` to 2.

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

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(L) {
  match (L) {
    [] => {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...] = L;
```

binds `h` and `t` in the following code if `L` is a nonempty list, and throws an exception if `L` 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.

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 is 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.
    [];} }
```

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

```
p = Person();
thingsHappen(p);
while (p ~ married(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 ~ married(q));
# 'q' is bound to p's spouse here.
```

4.3 Built-in Data Types

Thorn enjoys a selection of fairly conventional built-in data types, with their constructors, pattern matches, and suitable 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, `L(1..4)` is a slice of the list `L`, and `n mod 1..4` is the number congruent to `n mod 4` between 1 and 4.

Lists are immutable ordered collections. `[a,b,c]` constructs or matches a fixed-size list. A postfix ellipsis `...`

in a list constructor or pattern indicates a sublist to be appended or matched. So, the standard map function for lists could be defined as:

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

Lists can be subscripted: $L(0)$ gets the first element, $L(-1)$ the last element, $L(3..5)$ gets $[L(3), L(4), L(5)]$, and $L(1, -1)$ is the tail of L . Many standard patterns for manipulating lists are available as *queries*; see Section 4.5.

Records are finite collections of immutable named fields. The syntax `{: a:1, b:2 :}` is used for record constructors and patterns. A lone identifier abbreviates `a:a` in a record.

```
fun rect {: x, y :} = {: x, y :};
  | rect {: r, theta :} =
    {: x: r * cos(theta), y: r*sin(theta) :};
# Extract and bind rect. coords:
{: x:x0, y:y0 :} = rect(p0);
```

The selector notation `r.a` also gets fields from records. Both field selector and record pattern notation get fields out of *objects* as well. 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. Here's a table of customers, storing, for each one, a customer number, name, profile, and other information.

```
customers = table {
  key custNo;
  val source, name, profile;
  var picked;
}
```

Tables have a statically-determined set of *columns* describing the information they hold (`custNo`, `source`, `name`, 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 are marked **key**. 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. `customers(n)` looks up customer number `n`. Destructuring assignment lets us get their name, profile, and picked bound to those names:

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

Rows can be added by the `@=` operation:

```
customers @= {: name, profile,
  custNo : n+1, source, picked:[] :};
```

or, in a more subscripting syntax using the key:

```
customers(n+1) := {: name, profile,
  source, picked:[] :};
```

or deleted by key:

```
customers \= {: custNo: 28 :};
```

or have **var** fields updated without requiring the whole row to be replaced:

```
customers(sPos).picked := [y.custNo, picked...];
```

Tables are objects, though without fields per se. Tables may be given methods. For example, it is often useful to give sequential numbers to rows of a table as they are added:

```
t = table {
  key pos;
  val source, name;
  fun add(source, name) {
    # compute new index (a bit clumsily)
    pos = max(0 :: %[r.pos | r <- this]) + 1;
    this @= {: pos, source, name :};
  } };
t.add(s1, "Kim")
```

adds Kim to the table at a position after all current table members.

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 (usually) a single key, appears in many scripting languages. With only a single datum, map syntax in Lua [21] or Python [48] is less wordy than the full table syntax from Thorn. So, Thorn has syntactic sugar to let tables serve as maps. Programs using maps instead of tables frequently evolve to have several parallel maps (or a map containing records or objects), 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{ key 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 :}; #(*)
assert(m[1] == 2);
assert(m(1) == {: k:1, v:2, a:3 :}); #(*)
m[1] := 22;
assert(m(1) == {: k:1, v:22, a:3 :}); #(*)
```

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

The confectionary construct `map{a => b, c => d}` constructs a table with a **key** field `k` and a **map var** field `v`, and populates it with two rows initially. This provides the familiar maps and dictionaries of many scripting languages as a special case of a more powerful table construct.

4.5 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.

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, as from SETL [38] or Python. (The % symbol is used to distinguish queries; operations with [] produce lists, {} tables, and () arbitrary data.) To compute the list of squares of primes up to 100:

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

The **while** query control stops iteration altogether, 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 **%last** query returns a value from the last iteration. The sum of a list can be computed as:

```
%last( sum | for n <- L, var sum := 0 %then sum+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 source field is **sender**, one can use:

```
% find(r | for r <- custs, if r.source == sender);
```

The trichotomy query **%()** provides a declarative way to find the one desired element of a collection, handling the cases of “not found” and “too many found” if desired. This is particularly useful in scripts that have to deal with English or other natural languages with a negative/singular/plural distinction.

```
%("The desired one is $x."
  %none "There are no desired ones.")
```

```
%many "There are several, and they are %(...x)."
| for x <- them, if desired?(x) )
```

In the first clause, returned if there is precisely one desired value, `x` is the value of the iteration variable `x` for that value. In the **%none** clause, evaluated if there are no iterations (*viz.* `them` has no `desired?` elements), `x` is conceptually meaningless and thus out of scope. In the **%many** clause, evaluated if there are several desired values, there is no single value of `x` to use, and so `x` is meaningless; but `...x` is bound to the *list* of `x`'s from the (two or more) iterations.

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. Suppose that `corpus` is a list of records summarizing information about some emails. The following computes a table of senders and their email addresses; note the pattern matching in **for**.

```
%table{
  key email_addr = email_addr;
  val sender_name = sender_name;
  | for { : email_addr, sender_name : } <- corpus}
```

(There may be several emails with the same email address; they are consolidated into a single entry in the table, and it is an error if their `sender_names` differ.)

As database programmers discovered long ago, it is often useful to aggregate information: `GROUP BY` in SQL; **%group** in Thorn. As in the **%many** clause, `...n` is the *list* of values of `n` that fall into the group. A common use of this is to split a list based on whether or not the elements satisfy a predicate. To compute the lists of primes and composites up to 1000 in a single pass (and, for amusement or reference, their squares), declaratively:

```
prime_or_not = %group{
  key prime = prime?(n);
  map val numbers = ...n;
  val squares = ...(n*n);
  | for n <- 1 .. 1000;};
primes = prime_or_not[true];
composites = prime_or_not[false];
```

4.6 Equality and Identity

The intent of *equality* is that `a == b` means that `a` and `b` have the same state at the moment. Equality is provided automatically for built-in data types and for *value* classes annotated with the **Value** constraint (see Section 6.1.2); this operator tests for structural equality and may not be overridden. The default implementation of `==` in non-value classes throws an exception, but it can be overridden with an appropriate user-defined notion of equality if desired.

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 identity cannot be accessed directly. However, `Identity` implements the method `a.same?(b)` that returns true if `a` and `b` are the same object.

The class `GlobalIdentity` is a subclass of `Identity` and gives an object a globally unique identifier relying on what such service is available in the underlying operating system. If no such service is available, Thorn provides a simple default implementation with a weak uniqueness guarantee. 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 goal. Thorn focuses on *distributed* and *concurrent* computing. That is, 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. Processes need not be able to share memory references, file systems, processor types, or even time zones.

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 can be nearly anything (we discuss the exceptions below)—but if the content is a mutable entity, the entity is deep-copied. (This includes many objects, tables, closures referring to mutable state, and so on.) The receiver must be content with working with a facsimile, a snapshot of the mutable object at the instant it was sent.

Subtleties arise when sending objects around, especially when sending a `Person` object to a component that has a different `Person` class than the sender, or no `Person` class at all. Such cases can be detected by sending a hash of the `Person` class. Note that entities 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 proto-

col [24], and should, provide a convenient way to communicate with any JSON-compliant program, Thorn or not.

Component and Spawn. 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. They have a `body` section, giving code that they are to run.

```
component Lifer {
  var universe := # ...
  body {
    # code to run Conway's Life here.
  }
}
```

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

```
c = spawn Lifer;
```

`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:

```
c = spawn { var universe :=
            # as above
          };
```

High-level Communication. Thorn provides two communication models. The high-level model provides named forms of 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 3. (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—so the syntax makes it very clear which one you are doing.) Method invocation uses the familiar `ob.m(x)` syntax. Asyncs use `comp <-- m(x)`, and syncs use `comp <-> m(x)`. The arrows are intended to suggest both the length and directionality of the communication.

The receiver has detailed 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) {deal();}` clause in cases where waiting indefinitely for communication is undesirable. A typical reactive component will have a skeleton of the following form. 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.)

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

```

proc {
  sync findIt(aKey) {
    logger <- someoneSought(sender, aKey);
    # ... code to look it up ...
    return theAnswer; }
  ...
logger = proc {}
  var log := [];
  async someoneSought(who, what) {
    # do not answer, just cons onto log.
    log ::= { : who, what : };
  } }

```

Figure 3. High-level Communications

Low-level Communication. The low-level communication model allows sending unadorned values from component to component. `comp.send(v)` sends value `v` to component `comp`. `v` simply goes into `comp`'s mailbox: a list of values that have come to `comp`, but that `comp` has not yet dealt with.

`comp` can retrieve values from its mailbox by the **receive** statement, a variation on Erlang's **receive** scans through the mailbox, looking for the first message of the highest priority that matches one of a set of patterns. When it finds one, it executes the corresponding arm, 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 satisfying a predicate; 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: true : } prio 1 => {return;}
| { : please: "post", data: x : } => {do_post(x);}
| { : please: "scan", want: pred : } => {do_scan(pred);}
| timeout(10000) => {bored := true;}
}

```

Confined Values and Messages Consider the following Thorn fragment:

```

var x := 0;
class Questionable { x++; };
comp = spawn {...};
comp<-Questionable();

```

This code is problematic, since the class `Questionable` has a reference to mutable global state, which is by definition inaccessible on the receiving end. To prevent unexpected behavior in such cases, we forbid sending data containing functional values (i.e., objects, closures, and tables) as messages unless they satisfy the following *confinement* conditions: (1) atomic data (integers, strings, floats, booleans) and component references are confined; (2) records, lists, and tables are confined if they contain only confined data; (3) a variable or class field is confined if it is immutable (i.e., not declared using **var**) and is bound to a confined datum; (4) a method is confined if all the variables or fields it accesses

and which are defined outside its scope are confined; (5) a closure is confined if all the variables or fields that it accesses and which are defined outside its scope are confined; (6) an object or table is confined if all the methods it can access are confined. Note that a confined method is free to access and update mutable fields of the class that defines it.

In addition to checking the confinement property 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 confinement 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.

The confinement property is checked dynamically before a datum is sent, but the check can be optimized in various ways. Section 6.1.2 shows how simple constraint annotations can be used to identify *values*, which satisfy more stringent requirements than confined data and allow message passing without copying when the sending and receiving components are in the same address space.

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.

6.1 Constraint Annotations for Robust Programming

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. The following sections describe access modifiers, value classes and Thorn's "like types", which all rely on the annotation system.

6.1.1 Access Modifiers

The intent of Thorn's access modifiers is to provide the most useful access for each sort of thing. So, methods of objects are public by default, as are the fields of records. However, it is generally wise to hide fields of objects. Access restrictions cannot be enforced solely at compile time:

```

class A { var f := 1; fun m(x) = x.f; }
a = A();
class B { var f := 2;}

```

For example, `a.m(a)` and `a.m({ f:3 :})` obey the access restrictions, but `a.m(B());` does not.

The **Public** annotation makes fields public:

```

class Point(x: Public, y: Public) { ... }

```

In sufficiently typed contexts (e.g., if `x` were declared to be an `A`), the compiler can tell which field accesses are acceptable. In untyped contexts, the determination must be made at runtime. The compiler imposes the additional restriction that

only private fields of **this** are accessible, not those of other objects, even of the same class. This allows an efficient compilation strategy.

6.1.2 Values and Immutability

To promote a less brittle programming style, to enable optimizations, and to facilitate safe sharing of data among Thorn components, Thorn lists, records, and strings are immutable. To extend the benefits of immutability to objects and closures, it is useful to define the notions of *value*, *constant* and *pure methods/closure* as follows: (1) atomic data (integers, strings, floats, booleans and component references are values; (2) records and lists are values if they contain only values. (3) a variable or class field is a *constant* if it is immutable (i.e., not declared using **var**) and is bound to a value; (4) a method is *pure* if all the fields it accesses are constants, and all the variables it accesses that are defined outside its scope are constants; (5) a closure is pure if all the variables or fields that it accesses and which are defined outside its scope are values. (6) an object is a value if all of its fields are constants and all of the methods it can access are pure. Note that the conditions enumerated above strengthen the confinement conditions discussed in Section 5.

To facilitate enforcement of these constraints, we provide **Value** and **Pure** constraint annotations for fields, variables, and classes (in the first case) and methods and closures (in the second case). Checking these annotations in a statically typed language is straightforward. However, in an untyped language, it is generally impossible to tell statically whether a method is pure and thus should only be called with value arguments, or whether an expression used to instantiate a **val** field returns a value or not. Our solution is to generate two method entries for every pure method, e.g., `foo` and `pure$foo`, and translate every method call inside a pure method at compile-time from `x.bar()` to `x.pure$bar`. Thus, only pure methods will be called from within a pure method, and passing in an argument that does not have a pure `bar` will throw a run-time `Impure` exception. Field accesses in Thorn are implemented through method calls, so the same technique works to purify them as well.

6.1.3 Types In Thorn

In Thorn, the type of variables, fields and parameters defaults to **dyn**, the dynamic type. Uses of untyped variables are unconstrained at compile-time, and checks that a method call is actually understood by a receiver are delayed until the method call. If the method is not understood, an exception is thrown.

Thorn supports gradual typing [39] using *concrete* types and *like* types. Class declarations introduce new types that can be used to annotate variables, parameters, method returns, and so forth. In a completely untyped setting, class types simply describe the methods understood by the class and its superclasses. Whatever types the programmer supplies (e.g., types of methods and formals) are included. A

simple type inference engine deduces types of local variables from the information thus provided, and rejects the code if it can be proven never to work. In lightly-typed programs, such proofs are rare; the main use of type inference is optimization.

Class types are called *concrete types* in Thorn. Since they are nominal, dynamic tests for class membership are quick. Most importantly, basic data types like integers and floats can be represented in terms of Java primitives, which improve performance considerably. Correct programs will call concretely typed functions from untyped code; the compiler inserts run-time checks as necessary.

To allow more flexible typing than is possible using concrete types, we provide *like* types, a variant of structural typing. A companion paper [50] describes this notion and its application to optimization in detail; here we review a few key attributes. *like* types can be constructed from any class type, e.g., **like Profile**. A variable typed **like Profile** will be type checked against the protocol of `Profile` but no guarantees are made as to how the variable may be bound at run-time. In contrast to gradual typing systems with structural subtyping (e.g., [39]), run-time checks that a value in a variable of type **like T** conforms to `T` are done for each individual method call, right before the call, much like operations on **dyn** typed variables. As a consequence—and unlike structural typing systems—conformance is only required for the parts of the protocol actually used. We show in [50] that an optimizing compiler for Thorn which takes advantage of type annotations can yield performance for the Fannkuch benchmark in Figure 5 that is twice that of Python 2.5, even though Thorn arrays are immutable.

6.2 Modules

Overview 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 [34]. 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 roughly based on the upcoming Java Module System [45, 46]. In line with our previous work [43], 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. Albeit, the Thorn module system remains simple and non-intrusive. Thorn’s module aliasing and module-prefixed references remove the need for renaming of imported names; and, the module system supports features such as per-import repository override, module name aliasing, and module-level generics (not described here).

```

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

S.C: Public;       # re-exported as M.C
# N.A: Public;     # ERROR: M.A already exported

```

Figure 4. Thorn Modules: A simple example.

Non-intrusiveness All the above power is provided in a non-intrusive manner. For example, Thorn source files in a single directory automatically belong to the same (unnamed) module definition. Consequently, a source file does not represent a semantic boundary. When an application grows, module membership can be specified either by declaring membership on top of the source file (as shown in Figure 4), or through a module file (by explicitly listing file names by URI). When a module definition is compiled, it is automatically placed into a filesystem-based repository, making it available for importing. Alternatively, the location of an imported module can be specified explicitly with a URL.

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 4 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 4, *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 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. The statement `import N;` imports the component-wide instance of *N*, while `import own O;` creates a non-shared instance for the current context. 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 executing in parallel [43].

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 4. 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.

Compilation The module system supports incremental, cut-off compilation similar to the Java’s compilation of source files. Separate compilation is currently supported only through explicit specification of interfaces of imports in the form of module definition stubs. This could be improved upon by either adding explicit module interfaces (and a form of subtyping), or by having compositional constraints [3].

7. JVM Implementation

Method Dispatch Thorn sports several features that, at face value, are incompatible with the Java object model and the JVM. The perhaps 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 take the same solution as JRuby [23] and Jython [25]. The Thorn implementation generates dispatch interfaces. Each method gives rise to an interface. For example the method `fun foo(x)` will generate an interface `Ifoo_1` declaring a unary 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 not strictly necessary, however it allows for better 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. 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.) 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. A field inherited from a superclass actually becomes a field of the child class. Another issue concerning fields is that since Thorn is dynamically typed we cannot statically reject programs that assign to fields declared `val`. So, we do it dynamically, by having the setter of a `val` field throw an exception.

7.1 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 datatypes are translated into primitive operations on unboxed primitives for speed. Such optimisation is only possible locally and as soon as values move in and out of containers, are passed to and from methods, etc., the primitives must be boxed into proper Thorn objects.

A companion paper on *like* types [50] reports on a type-based optimizing compiler of Thorn that allows methods on several primitive data types to be called with primitive values and arrays of primitive values without unboxing, which gives a huge performance boost in code that largely operates on basic data types.

7.2 Implementation of 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 instance of a class inside a 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 [41] that rewrites bytecode into CPS to allow preempting a component schedule another in the same thread. To minimise 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. So, a Thorn program can spawn a very large number of concurrent components without undue overhead.

7.3 Performance

Although the current prototype of Thorn was not designed with performance as a primary goal, its performance is comparable to Python. Several benchmarks from The Computer Language Benchmark Game were translated to Thorn and their runtime compared to the existing benchmarks running

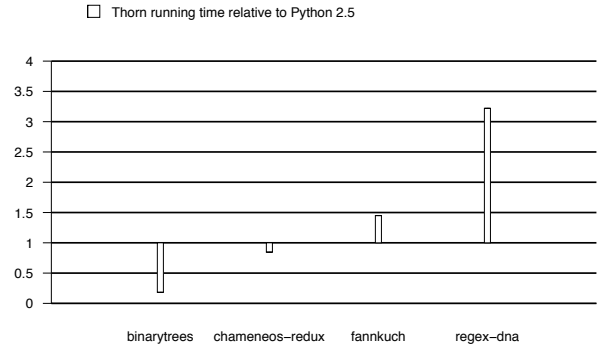


Figure 5. Performance of Thorn in relation to Python 2.5.

on Python 2.5. On half of the benchmarks tested, Thorn is faster.

Several conclusions can be drawn from these results. Thorn’s support for the basic features of object orientation, 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.

7.4 Extensibility Features

A key design goal of Thorn 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.

7.4.1 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 load 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 also hook into the existing passes such as name resolution. This core language AST is compiled to Java bytecode and evaluated.

7.4.2 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 [16]. 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 [49], 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 interpreter itself to support new AST nodes; rather, these are translated into core language nodes for the interpreter subsequently to evaluate.

7.4.3 Plugins for extensions: Assertions

Figure 6 shows the complete code for a plugin that adds an `assert` construct like Java's to Thorn. The plugin is implemented 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.

7.5 Java Integration

Thorn programs run on a Java virtual machine and 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.

8. 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

```

object AssertPlugin extends Plugin {
  fun 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;
  }

  fun goals(unit) = [AssertGoal(unit)];
}

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

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

```

Figure 6. An assert plugin. It adds the statements `assert <exp>` and `assert <exp> : "Message"` to Thorn.

avoids problems generally associated with shared memory and lock-based synchronization. Most of Thorn's features are not altogether 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] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In *Proceedings of POPL*, volume 40(1) of *ACM SIGPLAN Notices*, pages 26–37. ACM Press, January 2005.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] Kenneth Barclay and John Savage, editors. *Groovy Programming*. Morgan Kaufmann, December 2006.
- [8] Kent Beck and et al. Principles behind the agile manifesto, 2007. <http://agilemanifesto.org/principles.html>.
- [9] Gilad Bracha. Pluggable type systems. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, October 2004.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] Clojure. <http://clojure.org/>.
- [14] 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.
- [15] Matthew Flatt and Robert Bruce Findler. PLT Scheme Guide – Modules. <http://docs.plt-scheme.org/guide/modules.html>, 2009. Version 4.1.5.1.
- [16] 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.
- [17] Jesse James Garrett. Ajax: A new approach to web applications, February 2005. <http://www.adaptivepath.com/ideas/essays/archives/000385.php>.
- [18] A. Hejlsberg, S. Wiltamuth, and P. Golde. C# language specification. 2003.
- [19] 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.

- [20] Roberto Ierusalimsky. *Programming in Lua, Second Edition*. Lua.org, 2 edition, March 2006.
- [21] Roberto Ierusalimsky, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of Lua. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26, 2007.
- [22] Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003.
- [23] Java powered Ruby implementation. <http://jruby.codehaus.org/>.
- [24] Introducing JSON. <http://www.json.org/>.
- [25] The Jython Project. <http://www.jython.org/>.
- [26] The Kawa language framework. <http://www.gnu.org/software/kawa>.
- [27] Barbara Liskov and John V. Guttag. *Abstraction and Specification in Program Development*. MIT Press/McGraw-Hill, 1986.
- [28] 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.
- [29] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
- [30] 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.
- [31] Nathaniel Nystrom and Vijay Saraswat. An annotation and compiler plugin system for X10. Technical Report RC24198, IBM T.J. Watson Research Center, 2007.
- [32] 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.
- [33] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.
- [34] David L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12), December 1972.
- [35] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, 2000.
- [36] 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.
- [37] Vijay Saraswat et al. The X10 language specification. Technical report, IBM T.J. Watson Research Center, 2008.
- [38] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets; an introduction to SETL*. Springer-Verlag, New York, 1986.
- [39] 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.
- [40] 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.
- [41] 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.
- [42] 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.
- [43] Rok Strmiša. Fixing the Java Module System, in Theory and in Practice. In *Proceedings of FTJJP*, pages 88–99. Radboud University, July 2008.
- [44] Sun. *Core Java J2SE 5.0*. Sun Microsystems Inc., <http://java.sun.com/j2se/1.5.0/>, 2005.
- [45] Sun Microsystems, Inc. JSR-277: Java™ Module System. <http://jcp.org/en/jsr/detail?id=277>, October 2006. Early Draft.
- [46] Sun Microsystems, Inc. JSR-294: Improved Modularity Support in the Java™ Programming Language. <http://jcp.org/en/jsr/detail?id=294>, 2007.
- [47] David Thomas and Andrew Hunt. *Programming Ruby: the pragmatic programmer's guide*. The Pragmatic Programmers, LLC., Raleigh, NC, USA, 2 edition, August 2005.
- [48] Guido van Rossum and Fred L. Drake Jr., editors. *The Python Language Reference Manual (version 2.5)*. Network Theory Ltd, 2006.
- [49] 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.
- [50] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebesne, Johan Östlund, and Jan Vitek. Integration of Typed and Untyped Code in Thorn. Submitted to OOPSLA 2009.