Proposed Changes to Thorn

Bard Bloom, Julian Dolby,
John Field, and Emina Torlak
IBM Research

Nate Nystrom
University of Lugano

Jan Vitek
Purdue University

August 10, 2010

This document collates recently proposed updates to the Thorn language, both semantic and syntactic. It summarizes the contents of the various discussions posted to the Thorn mailing list, from May through August 2010.

Contents

1 Changes to Syntax

1. Purity is currently marked by :**pure**, which was going to be an instance of some general syntax, but is not. Make it some other kind of modifier, like a prefix, if we keep it at all.

2. We could allow '**var** x = 4' as a synonym for '**var** x := 4'. It's not clear that this is a good idea, since using '=' for assignments may be confusing.

3. The ';' after '2' should be optional, but is not:

```
fun f(1) = 2;
| f(_) = 3;
```

4. **from**, **envelope**, and **prio** clauses on method and function definitions should be allowed in arbitrary order.

5. The constructor supercall should allow **new**@**super**(...) as well as **new**@A(...).

6. Remove **import own**.

7. Right now, public/private statements in a module are separate statements:

```
module Moo {
        var a; var b; var c;
        private a; private b;
}
```

   Having **private** be a modifier on declarations (and getting rid of **public**) would probably be better style. Failing which, at least allow: **private** a,b;

8. Currently a few choices of statement can be put in parens to make an expression: (**if** (true) 3; **else** 4;). Probably all of them should.

9. {: ... :} is pretty unsatisfying for something as fundamental as records. They are far more important than the two-char syntax indicates. Go for { x : 1, y : 2 } if possible. Another proposal is to use '**new** { x : 1; y : 2 }'.

10. Remove the '.nice(x)' pattern. The currently supported alternative 'it.nice ∼ +x' isn't much wordier, and is clearer and more general. Maybe also eliminate the '.nice()' pattern, which abbreviates 'it.nice ∼ !(null || false)'.

11. Change the syntax currently described as the nonterminal 'cases':

```
try {
        x := 1;
} catch {
        "Doom!" => {print("doom");}
        | "Mood?" => {}
}
```

    The '|' separating the two cases is too small visually to separate chunks of code. Allow something else as an option: perhaps '**case**'.

12. The syntax for table column declarations is awkward:

    ```
    t2 = table(x,y){var fg; val bg; zog : int;};
    ```

    The current syntax is intended to parallel var/val decls, which it does, but that's not so nice. It would be better to turn the semicolons into commas, and have the last one be optional:

    ```
    t2 = table(x,y){var fg, val bg, zog : int}
    ```

    Ditto for ords.

13. Replace the type constraint operation ':' with '::', which is probably essential to revising record syntax and perhaps conditional expressions.

14. Could we get away with using C-style conditional expressions, i.e. 'A ? B : C'? Originally we didn't do that because the ':' is being used for other things, but we may now have it clear enough.

    Failing which—could 'if A then B else C' be an expression with ANTLR? It didn't work in JavaCC, for LL reasons.

15. The % sign in queries is a problem. Can we do the following:

    Vars

    Most query controls don't need adaptation. The one that does is 'var'

    'var i := E %then1 F' → 'var i := E ,, F'

    (Note—consistent with the use of ,, in groups, later on.)

    'var i := E %then0 F' → 'var i := ,, E ,, F'

    This is not ideal.

    List Comprehension

    [Exp ...? | QueryControls]—that is, using the presence of a '|' early in the list to disambiguate from list constructors?

    Quantifiers

    Using **count**, **all**, **exists** instead of **%count**, **%all**, **%exists** is probably good enough. There is some issue about claiming a popular identifier like 'count' as a reserved word.

    First

    Use **first** instead of **%first**/**%find**. Use **find** as the statement-level variant (drop **first** as synonym). Use **else** for **%none**.

    After

    Use **after** instead of **%after**.

    Table Comprehension

    Can the '|' in the table constructor work here too? And the '=' in the key fields?

    ```
    {table(x=i){xsq = i*i | for i <- 1..100}
    ```

    vs a table constructor:

    ```
    table(x){xsq}
    ```

    Sorting

    Use **sort** instead of **%sort**. Use **by** instead of %<. Use **yb** instead of %>. Or maybe **by decreasing** for a more cobolly effect?

    Group queries

    Use **count** instead of **%count** (a previously introduced keyword); 'E ...' instead of '**%list** E'; '... E' instead of

'**%rev** E'; and 'E ,, F ,, G' instead of '**%first** E **%then** F **%after** G'.

So, oldstyle:

```
%group(prime? = n.prime?) {
num = %count;
them = %list n;
meht = %rev n;
sum = %first n %then sum+n;
sumAsString = %first n %then
  sum+n %after sum.str;
| for n <- 2 .. 100 }
```

would be, newstyle:

```
group(prime? = n.prime?) {
num = count,
them = n ...,
meht = ... n,
sum = n ,, sum+n,
sumAsString = n ,, sum+n ,, sum.str
| for n <- 2..100 }
```

16. Timeout clauses in **recv** statements should have the same syntax as other clauses, mutatis mutandis.

    Currently:

    ```
    msg = (recv{
      x => x
      timeout(1000) { "timed_out";}
    });
    ```

    The timeout clause requires braces and no =>; it should look more like a regular clause.

17. We should allow

    ```
    def glurd() {frell();};
    ```

    that is, def's (and many other things) should be allowed to have trailing semicolons.

18. The **body** block in a **spawn** should just be the body of the spawn. Things like initially, reinit, before, after should just be methods of the component and not baked in. Spawn should be similar to **new** and just take a class name (a subclass of Component providing the appropriate methods) and arguments to pass to the constructor. **spawn** { ... } is equivalent to **spawn** <anon subclass of Component>. One should be able to do **new** { ... } to create an anonymous subclass of Any.

19. The body of a class declaration should allow expressions invoked before the constructor—like an initializer block in Java.

20. Replace the '...' patterns with the 'x::xs' syntax.

21. $(k) in patterns should be writable as $k.

22. Make \n a synonym for ';'. The semicolon should be used as a separator consistently.

23. Remove singleton objects. Should be using top-level module definitions instead.

24. Remove matches baked into function definitions. Explicitly write:

    ```
    fun foo(x) = match(x) { ... }
    ```

2 Changes to Serialization

2.1 Allowing transmission of arbitrary pure closures and objects

Proposal

We should allow transmission of arbitrary pure closures and arbitrary pure objects (including anonymous objects).

Rationale

This is needed to model queries against long-lived data (persistent or otherwise), code mobility in a cloud setting, allow encoding of futures, etc. Additional justifying comments below.

Details

Revised definition of purity:

- a variable or field is pure if it is immutable (non-val) and bound to a pure value
- a closure/method is pure if it contains no non-local references to impure variables/fields (an object's own fields are considered non-local)
- any javaly class or function that accesses site-specific (mutable or immutable) state is impure
- a class reference is pure if its constructor is pure
- a pure object has only pure fields and pure methods
- a pure class has only pure fields and pure methods
- tables, maps, and ords are impure
- files are impure
- all other built-in values (primitive types, records, and lists) are pure
- NB:
  - pure closures and methods can mutate values passed as arguments, call impure methods on objects passed as arguments, and access mutable local state
  - pure class references can be used to construct impure objects, and pure objects can be constructed via impure class references

Additional details:

- Do not require the 'pure' keyword (but allow it for documentation/optimization) for defining pure values. Instead, values are (conceptually) checked dynamically for purity. The dynamic check can be optimized in various ways, including adding an extra bit to make the check unit time.
- Closures/methods should (conceptually) be double-checked for purity on receipt, as well as on sending. Such checks can be implemented fairly cheaply when the sender and receiver are aware of each other. When they don't, something like a bytecode verifier will probably be needed.
- Classes defined in two components (even if derived from the same module) are incomparable. So, e.g., an object o of class A defined in component C1 will, when transmitted to component C2, fail to match a pattern in C2 which constrains o to have type A. "like A" tests should still work, however.
- Given the above, we should consider changing the semantics of ":A" tests, e.g., ":A" is an eager version of ":like A"—it checks for interface compatibility with A, but unlike ":like A", does so immediately. Or perhaps we should just get rid of ":A" altogether. Classes in thorn are mostly about code sharing/inheritance and encapsulation, not typing. So making the distinction between types and classes clearer is probably a good thing.
- Get rid of **import own**. It complicates the semantics and implementation in a number of

respects. No one has found a compelling need for it thus far. Components provide an alternative, and more thorn-idiomatic way of replicating "global" state, and objects are always available if you want "per instance-of-something" state.

– Eliminate the Identity predefined superclass in favor of GlobalIdentity. This ensures that object identity (if it exists at all—recall that it doesn't by default) can be safely preserved when objects are transmitted, and remains invariant when an object is passed from component to component. We can avoid generating a globally-unique identity when the object is known not to escape.

2.2 Allowing transmission of modules

Proposal

The concept is that you can send around a *module*. Transmitted modules are received as *seeds* rather than data. Seeds have very few operations on them. The most crucial operation is that they may be passed to new components when they are being instantiated, and the new component can import the module. (This is kindred to Erlang's mechanism for sending module names around.)

Rationale

Sending around arbitrary closures, and having them be receivable and applyable, carries a variety of security and implementation costs.

Details

We take advantage of the isolation of components here—untrustable transmitted code is executed in a *new* component, so its privileges and ability to mutate data are naturally somewhat curtailed.

We write **seed**(DB1) for the seedification of DB1, and **seed** M for a seed-valued formal parameter of a component. We may need to know which formals are which, for compilation.

A standard example of something we want to do is, send a query to a database and have it evaluated on the database.

We'll package that query in a module. Here's one, which gets the 'a' column from records a,b,c in the table "abc's" where c==31 and a<b:

```
module MyQuery {
  import AbstractDB.DB;
  fun query({: a,b,c:31 :} && a<b) = +a
    | query(_) = null;
}
```

The database server would look something like:

```
sync runQuery(Q : seed) {
  // Spawn a local component...
  comp = QueryProcessor(Q);
  answer = comp <-> doIt();
  answer;
  }
```

and QueryProcessor would be defined as something like:

```
component QueryProcessor(seed Q, db) {
  import Q.query;
  sync doIt() =
    %[ res | for (row <- db.iter()), if query(row) ~ +res];
  }component
```

db.iter() does lots of communication, sending rows of the database one at a time to the QueryProcessor. However, this is local communication, and hopefully sending the rows is a cheap pointer toss.

The "import Q.query;" line is where the magic happens in this proposal.

Oh, and to use the query processor, you'd have

```
import MyQuery;
database <-- runQuery( seed(MyQuery) );
```

The use of **seed** may not be necessary in all those places. The use of modules makes this example somewhat blurrier. It'll probably come out looking better when sending a class around, or several related things.

3 Database Integration

3.1 Version 1

Handling queries on remote tables is a prerequisite to backing Thorn tables by database tables.

Recall that the basic proposals for remote table access are (a) transmitting closures containing the desired query to the component containing the remote table (§2.1); (b) putting the query code in a module (§2.2), defining some kind of "module handle" datatype whichs points to the module thus defined, starting up a component on the same site as the remote table which runs the module pointed to by the handle, and which communicates via some kind of non-query API (e.g., one which conceptually copies the table, runs the query, and then updates the shared table) to the component containing the shared table. The plumbing connecting the query code to the table/row copying code in (b) could either be encapsulated in a library or baked into the language. The objection to (a) is that it requires defining some notion of pure closure, and can serve as a vector for security problems. The objection to (b) is that it seems kind of complicated and kludgy.

Some thoughts: On reflection, (b) is not that different from the way real databases work: queries are run in distinct processes and (conceptually) access the shared relations one row at a time via something like a concurrent B-tree. Also, (b) has the advantage of allowing us to use the work-in-progress security model for Thorn to control access to distinct relations or even rows (since relation/row access would go through the lower level API). So (b) is certainly defensible semantically. Moreover, the principle of disallowing "injection" of new code into running components is semantically simple and justified pragmatically by the desire to statically compile modules (in addition to security concerns). The question is how to realize (b) in a way that allows access to remote tables with only slightly higher syntactic overhead than local tables.

Here's a slightly fuzzy idea of how this might work:

- Define a RemoteTable library (module)
- RemoteTable contains a function (say runQuery) which takes as arguments a closure containing the query, a component reference to the component containing the remote table, and a list of remote table names that will be accessed by the query.
- So, e.g., you might do:

```
import RemoteTable;
runQuery( fn (t1, t2) =
 %first{ p | for r1 <- t1, for r2 <- t2 },
 someRemoteComponent,
 ["realTable1", "realTable2"] ).
```

RemoteTable would do the work of (a) generating a module and module handle containing the query code, (b) spawning the module handle as a new component at someRemoteComponent, (c) copying the contents of each remote table accessed into the new component (conceptually), (d) running the query, (e) returing the result to the caller.

Issue 1: the names of the remote tables are strings and we don't allow reflection, so there is currently no way to map the table-names-as-strings to the corresponding table names in the remote component automatically. You could do this indirectly by defining a map from string names to table values on the remote site, or we could allow limited reflection. It would be nice.

Issue 2: you obviously don't want to implement this by literally copying the entire table to the query component. You could instead translate the query to row-at-a-time access to the table component, but this would require magic to dynamically deconstruct the query into appropriate row access messages. Alternatively, the implementation could use magic which directly accesses the table (with appropriate concurrency control), rather than copying it. But then, if we have to have pattern-specific code generation magic for remote tables, it would be nice for it to be an instance of more "general magic" which that can be applied to various code patterns.

Final thought: maybe the remote table pattern is sufficiently important that it should just be 100% baked into the language, i.e., a remote table is a hybrid of a local table and a component reference, in the sense that you can query it directly just like a local table, but has failure modes similar to normal communication. But then what happens if a query accesses multiple remote tables? Do you need distributed transactions? Things get messy quickly...

3.2 Version 2

Like the proposal above, the following is based on the module transmittal mechanism described in §2.2. We propose introducing the **query** keyword to help with forming table queries, in a way that lets us execute them easily on all remote tables (native-Thorn and database-backed alike).

To run a Thorn query on a (database) table, we would write

```
ABC = SqlDb("sql://there.ibm.com/db/abc");
q1 = query(({ a, b, c:31 } && a < b) = +a | (_) = null);
q2 = query("SELECT...")
ABC <-> run(q1);
```

Each **query** call generates a handle to a fresh module with two functions, 'query' and 'sql':

```
module autogen_q1 {
  import ...
  fun 'query'({ a, b, c:31 } && a < b) = +a | (_) = null;
  fun 'sql' = "...";
}
module autogen_q2 {
  import ...
  fun 'query'(...) = ...;
  fun 'sql' = "SELECT...";
}
```

The function 'query' contains the body as the query as given. The function 'sql' contains an SQL encoding of the query.

When the remote component (e.g., SqlDb) spawns a local component initialized with a query-generated module handle, that component calls the appropriate function, depending on whether the underlying table is a Thorn table or a database. A runtime error occurs if the underlying table is a database and the Thorn query sent by the client cannot be translated to SQL. Similarly, a runtime error occurs if the underlying table is a Thorn table and the SQL query sent by the client cannot be interpreted as a Thorn query.

Q1: Can the component such as SqlDb contain more than one table? If so, how does one specify which table(s) the query should be applied to? Where within the auto-generated module are we importing remote table names (and is this necessary)?

Q2: If there is only one table per component, how does one express a join operation on two tables in two different components?

Q3: How do we modify remote tables (i.e., how do we add/delete rows/columns, change entries, etc.)?

4 Resolved Syntax Changes

Legend: ✓ marks adopted changes, × marks rejected changes, and ? marks proposals that need further discussion.

✓ 1. Purity is marked by the prefix modifier **pure**.

× 2. Do not allow '=' to be used for assignments.

✓ 3. Semicolons should be treated as (optional) separators rather than terminators.

✓ 4. **checked**, **from**, **envelope**, and **prio** clauses on method and function definitions are allowed in arbitrary order.

✓ 5. Adopt the proposed change but use the keyword **super** instead of **new**. That is, the constructor supercall should allow **super**(...) as well as **super**@A(...).

✓ 6. Remove **import own** in its current form. However, we may re-introduce the functionality of **import own** when we revise the module system.

✓ 7. **private** must be a modifier on a member declaration. A member may not be marked as **private** in a separate statement, after it has been declared. We must allow **public** to be used in a separate statement as a way to re-export specific members of imported modules.

✓ 8. Every Thorn statement should be usable as an expression. Braces and parentheses can be used interchangeably to delimit expressions and eliminate ambiguity. Braces introduce a new name scope; parentheses don't.

✓ 9. Record syntax is now

```
<x = 1, y = 2>
```

✓10. Remove the '.nice(x)' pattern.

×11. Allow the following alternative syntax for cases in all matching contexts (e.g., match statement, receive statement, etc.):

```
try {
        x := 1;
} catch {
        case "Doom!" => {print("doom");}
        case "Mood?" => {}
}
```

✓12. Replace semicolons with commas and braces with angle brackets in table, ord and map declarations. The keyword **val** should be disallowed (it is the default), and the last comma should be optional. For example:

```
t2 = table(x,y)<var fg, bg, zog : int>
```

×13. Retain ':' for the type constraint operation.

×14. Since if-statements can be used as expressions (see Resolved Changes 8), we don't need C-style conditional expressions.

?15. Eliminate % as a prefix to query keywords. Adopt the current proposal except in the following cases:

Vars
: Query control in 'var' needs adaptation, but the current proposal to use double commas (instead of %then1 and %then0) does not look nice.

Quantifiers
: Consider using **cnt** or # to replace %**count**. Rethink the use of braces vs. parentheses vs. angle brackets to ensure consistency with other constructs.

Table Comprehension
: Clarify the current proposal.

Sorting
: Use **inc** or **increasing** instead of %<. Use **dec** or **decreasing** instead of %>.

Group queries
: Adopt the proposal except for the use of double commas ',,' to replace %**then** and %**after**.

✓16. Timeout clauses in **recv** statements should have the same syntax as other clauses, e.g.:

```
msg = (recv{
 x => { x; }
 | timeout 1000 => { "timed_out"; }
});
```

✓17. See Resolved Changes 3.

?18. Parts of this proposal may be applicable if we allow spawning of arbitrary code.

×19. Do not allow initializer blocks in classes.

×20. Retain the functionality of the '...' patterns since they are strictly more expressive than 'x::xs'. However, replace the '...' syntax for these patterns with double dots ('..').

✓21. $(k) in patterns should be writable as $k, where k is a simple identifier (alphanumeric characters only). $(e), where e is an expression, should be allowed in both strings and patterns.

✓22. If \n is allowed as a synonym for the semicolon, we need conventions to deal with incomplete expressions broken up over multiple lines. For example

```
e1
+e2
```

One solution is to use a continuation character (e.g., '\') to break up expressions over multiple lines. For consistency, the same solution should be used for strings that are broken over multiple lines.

×23. Not applicable to the Fisher version of Thorn.

×24. Keep the matching in function definitions.