ZACH DEVITO, NIELS JOUBERT

THE LISZT LANGUAGE SPECIFICATION

17 MAY 2010

This document describes the expected behavior of the currently implemented features of the Liszt language.

Contents

Liszt is a language for writing mesh-based programs for solving partial differential equations. Liszt code is a proper subset of the syntax and typing rules of the Scala programming language. Currently Liszt programs are written in Scala; a compiler-plugin to the Scala compiler translates the Scala code into an intermediate representation used by the Liszt compiler. This document defines the exact subset of Scala supported by Liszt and details the usage and semantics of Liszt-specific language features. The structure of this document mirrors that of the Scala language specification and defers to that document when Liszt's semantics are equivalent to Scala's.

Since Liszt is embedded in Scala we distinguish between Liszt code and normal Scala code using the annotation `@lisztcode`. This annotation must appear before an **`object`** definition in Scala. Code within the definition is then treated as Liszt code subject to the rules defined in this document rather than the default Scala semantics.

```
@lisztcode
object MyLisztCode {
    <liszt code>
}
```

1 Lexical Syntax

Liszt syntax is identical to Scala syntax. Liszt supports the same identifiers, newline characters, and literals as Scala. Literal expressions in Liszt are the same as those in Scala for all the types that Liszt supports.

2 Identifiers, Names and Scopes

Liszt identifiers and names have the same scoping and precedences rules as Scala's. Liszt's scoping rules are also borrowed from Scala. One minor difference is the treatment of functions: Liszt does not allow for first-class functions. Functions may only be declared at object scope. They cannot be assigned to another identifier or passed into/returned from functions. [1] Furthermore, Liszt functions cannot be recursive, unlike Scala's functions.

[1] This makes it easier to compile to architectures that do not support closures, or would require more sophisticated memory management to support it.

3 Types

Liszt extends a subset of Scala's type system with Liszt-specific features. Explicitly stating types is unnecessary for most values, since Liszt uses Scala's type inference algorithms. Liszt has three categories of types: value types, reference types, and mesh types. There are no user-defined types, thus no classes are supported.[2]

[2] Eventually we would like to add a simple class system to aid code organization

3.1 Value Types

Value types are passed and return by value through functions, and have no Liszt-specific restrictions. They include:

`Int` — Fixed-point numeric type, equivalent to Scala's Int.

`Float` — Floating-point numeric type, equivalent to Scala's Float. The precision (single or double) is configurable in the `liszt.cfg` file using the `-floating-point-type` flag.

`Double` — Floating-point numeric type, equivalent to Scala's Double. The precision (single or double) is configurable in the `liszt.cfg` file using the `-double-type` flag.

`String` — Equivalent to Scala's String type. Liszt does not support any string manipulation expressions, but strings can still be used for output and debugging purposes.

`Boolean` — Boolean type, equivalent to Scala's Boolean and used in conditional expressions.

In this document, we use the term `Numeric` to refer to `Integer`, `Float` and `Double`. In addition to these simple value types, Liszt includes two higher-kinded value types representing small dense vectors or matrices. These types take meta-integer type parameters to statically determine their size. Meta-integers are static integer literals, represented with underscores preceding their integer value: `_1, _2, _3, ..., _9`. Meta-integers may only be used as type parameters to Liszt types requiring meta-integers, or as arguments to extract a specific entry from a vector or matrix.

`Vec[N <: IntM,T]` — A dense vector type of length `N`, where `N` is a meta-integer. `T` is any Liszt value type.

`Mat[R <: IntM,C <: IntM,T]` – A dense matrix of dimension `R` rows by `C` columns, where `R` and `C` are meta-integers. `T` is any Liszt value type.

3.2 Reference Types

Reference types are passed to functions by reference. However, in constrast to Scala's reference types, Liszt reference types cannot be returned from functions. [3]

[3] This prevents objects from escaping their defining scope, allowing architectures without dynamic memory management like CUDA to avoid having to deal with heap-allocated memory.

`Field[A <: MeshObj,T]` — A type representing a field defined over the mesh with support at elements of type `A`, where A is one of `Vertex`,`Edge`,`Face`, or `Cell`. `T` is the value stored in the field, and can be any value type. Fields must be declared at object scope and cannot be declared inside functions.

Though not fully implemented, `SparseVector` and `SparseMatrix` types will eventually be included as reference types

3.3 *Mesh Types*

The final Liszt types are those representing mesh elements and sets of mesh elements. All values of these types are constant, and are passed to functions by value. Variables assigned to mesh types must be `val` declarations, not `var` declarations. Hence it is illegal to reassign a mesh type. [4]

[4] This requirement makes it possible to always come up with a limited set of reachable topology given a particular piece of Liszt code, allowing for automatic domain decomposition

`Vertex` — A 0-dimension element, representing a single point on the mesh.

`Edge` — A 1-dimensional element, connecting two vertices.

`Face` — A 2-dimensional element (triangle, quadrilateral, etc.), composed of several edges.

`Cell` — A 3-dimensional element (tetrahedron, hexahedron, etc.), composed of several faces.

`Mesh` — An entire mesh. Currently the only build-in expression with type `Mesh` is `mesh`.

`Set[T <: MeshObj]` — A set of vertices, edges, faces, or cells.

In addition to these types, Liszt supports basic function types (eg. `(A, ... , B) => C`) and the `Unit` type. However, Liszt functions and `Unit` are not first class entities. Functions cannot be passed to or returned from functions, assigned to new identifiers, or declared inside other functions. `Unit` can only be used as the return type of a function that does not return a value.

4 *Basic Declarations and Definitions*

Liszt supports a subset of Scala's declarations:

```
val foo = <value type exp, reference type exp, mesh type exp>
var bar = <value type exp, reference type exp>
def baz(x : T1, ..., xn :TN) = <exp>
```

The user may also choose to annotate any of these declarations with result types. Functions must have exactly 1 argument list (unlike Scala where functions can have 0, or more than 1 argument lists). Liszt will ignore any modifiers on declarations. Type declarations `type T = Y` are not supported. [5] Arguments cannot be passed by name (`:=>` syntax).

[5] It would be nice to add support for these so we can assign aliases to vector types. eg: `type Float3 = Vec[_3,Float]`

5 *Classes and Objects*

Liszt does not yet support Scala's class system, though eventually a subset of it will be allowed. Currently code can be placed in multiple

Scala objects, where each object is annotated with the `@lisztcode` annotation.[6] Liszt code can call functions in other modules using standard Scala syntax:

[6] Until we implement some real class system, these objects allow the user to create modular code

```
@lisztcode
object Foo {
    def foo() {
        Baz.baz()
    }
}
@lisztcode
object Baz {
    def baz() {
        Print("This is baz.")
    }
}
```

Liszt objects containing a method called `main` taking zero arguments can be designated as the main class for a liszt program in the `liszt.cfg` file. This method will be invoked when the Liszt program is run. Variables and code written at object scope are run at Liszt startup. A module `A` relies on a module `B` iff A refers directly to an identifier in `B` or if `A` relies on a module `C` that relies on `B`. If `A` relies on `B` but `B` does not rely on `A`, then `B` will be initialized before `A`. In the case of a circular reliance between `A` and `B` the initialization order is undefined.

6 *Expressions*

The subset of Scala expressions that Liszt supports is listed here. When `<T>` is used below it means that an expression having type `T` is valid in that position.

6.1 *Literals*

All scala literal expressions for Liszt types are valid Liszt expressions for that type. Examples:

```
val a : Int = 1
val b : Int = 0x1F
val c : Float = 1.f
val d : Double = 1.0
```

6.2 *Unary Operators*

`-<Numeric>`,`-<Vec[N,T]>`,`-<Mat[R,C,T]>` — Numeric negation.

`~<Int>` — Binary not.

`!<Boolean>` — Logical not.

`<Vec[N,T]>.x` – extract the first element of a vector if `N >= _1`.

`<Vec[N,T]>.y` – extract the second element of a vector if `N >= _2`.

`<Vec[N,T]>.z` – extract the third element of a vector if `N >= _3`.

`<Vec[N,T]>.w` – extract the forth element of a vector if `N >= _4`. [7]

[7] For general matrix and vector element access, see Vector and Matrix Extraction

6.3 *Binary Operators*

If a binary operation takes two `Numeric` types and one is a `Int` while the other is a `Float` or `Double` the `Int` is promoted to the given floating point type, following Scala's type rules.

`<Numeric> + <Numeric>` — addition, if one argument is `Double`, promotes to type `Double`.

`<Vec[N,T]> + <Vec[N,T]>` — vector addition, defined if `T` is `Numeric`.

`<Mat[R,C,T]> + <Mat[R,C,T]>` — matrix addition, defined if `T` is `Numeric`.

`<Numeric> - <Numeric>` — subtraction, if one argument is `Double`, promotes to type `Double`.

`<Vec[N,T]> - <Vec[N,T]>` — vector subtraction, defined if `T` is `Numeric`.

`<Mat[R,C,T]> - <Mat[R,C,T]>` — matrix subtraction, defined if `T` is `Numeric`.

`<Numeric> * <Numeric>` — multiplication.

`<Numeric> / <Numeric>` — division.

`<Numeric> * <Vec[N,T]> , <Vec[N,T]> * <Numeric>` – scalar multiplication, defined if `T` is `Numeric`

`<Numeric> / <Vec[N,T]> , <Vec[N,T]> / <Numeric>` – scalar division, defined if `T` is `Numeric`

`<Numeric> * <Mat[R,C,T]> , <Mat[R,C,T]> * <Numeric>` – scalar multiplication, defined if `T` is `Numeric`

`<Numeric> / <Mat[R,C,T]> , <Mat[R,C,T]> / <Numeric>` – scalar division, defined if `T` is `Numeric`

`<Mat[R,C,T]> * <Vec[R,T]>` – matrix vector multiply, returns type `Vec[C,T]` if `T` is `Numeric`

`<Int> | <Int>` — binary or.

`<Int> & <Int>` — binary and.

`<Int> ^ <Int>` — binary xor.

`<Boolean> || <Boolean>` — logical or (not short circuiting[8])

`<Boolean> && <Boolean>` — logical and (not short circuiting)

`<value-`**`type`**`> == <value-`**`type`**`>` , `<mesh-`**`type`**`> == <mesh-`**`type`**`>` — Equality test returning `Boolean`

`<value-`**`type`**`> != <value-`**`type`**`>` , `<mesh-`**`type`**`> != <mesh-`**`type`**`>` — Inequality test, returns `Boolean`

`<Numeric> < <Numeric>` — less than.

`<Numeric> > <Numeric>` — greater than.

`<Numeric> <= <Numeric>` — less than or equal to.

`<Numeric> >= <Numeric>` — greater than or equal to.

`<Numeric> min <Numeric>` — Returns the less of the two operands.

`<Numeric> max <Numeric>` — Returns the greater of the two operands.

`<Vec[N,T]> min <Vec[N,T]>` — Maps the min operator across the vector, returns type `Vec[N,T]`

`<Vec[N,T]> max <Vec[N,T]>` — Maps the max operator across the vector, returns type `Vec[N,T]`

[8] The principle of least suprise says we should make these short circuit when we get a chance.

6.4 *Function Calls and Identifier Access*

`object_ident.ident` — access identifier `ident` from Liszt object `object_ident`. Identifier may be a variable or a function.

`<function-expression>(a1,...,an)` — function call of user-defined function.

6.5 *Field Access*

`field_identifier(<MeshObj>)` — Read the value of `field_identifier` at `<MeshObj>`.

`field_identifier(<MeshObj>) = <T>` — Write the value `<T>` to `field_identifier` at `<MeshObj>`.

6.6 *Dense Vectors and Matrices*

The following functions are used to construct dense matrices or vectors:

`def` `Vec(a1: T, ..., aN: T) : Vec[_N,T]` — Constructs a dense vector with entries corresponding to the given list of Numerics.

`def` `Mat(a1: Vec[_M,T], ..., aN: Vec[_M,T])` — Constructs a dense matrix of NxM with rows corresponding to the given Vectors. Only defined if M is constant over all the parameters. eg: `Mat(Vec(1,1,1),Vec(1,1,1))` constructs a 2x3 matrix.

The following operations are used to extract specific elements from a matrix or vector. They require meta-integer literals `_0, _1, ..., _9`.

`vec_identifier(<meta-integer>)` — extracts a value from a vector.

`mat_identifier(<meta-integer>,<meta-integer>)` — extracts a value from a matrix.

The following operations are used to assign specific elements in a matrix or vector. They require meta-integer literals.

`vec_identifier(<meta-integer>) = <T>` — assigns value T to the vector.

`mat_identifier(<meta-integer>,<meta-integer>) = <T>` — assigned value T to the matrix.

6.7 Built-in Functions

The following functions are built-in to the language.

`Print(as : Any*) : Unit` — Output all arguments, newline terminated.

def `FieldWithConst[MO <: MeshObj, VT](s : VT) : Field[MO,VT]` — Create a new field over `MO` with value type `VT` and initial value `s`. This can only be called at object scope, not within any function (including main).

def `FieldWithLabel[MO <: MeshObj, VT](url : String) : Field[MO,VT]` — Create a new field over `MO` with value type `VT`. Load the initial values using a locator string `url`. The only currently supported locator is "position", which loads the positions of the vertices in the mesh (the field must have type `Field[Vertex,Vec[_3,Double]]`). This can only be called at object scope.

def `BoundarySet[MO <: MeshObj](name : String) : Set[MO]` — Load the set of mesh topology of the given identifier from the mesh file`name`. Currently all fluent-file boundary sets are immediately avaliable through this inteface. This can only be called at object scope.

`mesh` — A handle to the global mesh object.

def `size[MO <: MeshObj](s : Set[MO]) : Int` — Retrieve the size of a set.

def `cross[VT](a : Vec[_3,VT],b : Vec[_3,VT]) : Vec[_3,VT]` — Cross product

def `dot[N <: IntM, VT](a : Vec[N,VT],b : Vec[N,VT]) : VT` — Dot product

def `normalize[N <: IntM, VT](a : Vec[N,VT]) : Vec[N,VT]` — Return the normalized vector.

`ID[MO <: MeshObj](m : MO) : Int` — Return the unique ID of the mesh object as it was named in the input file.

`wall_time()` — Return the time is seconds since the program started. Low resolution accurate to 2ms.

`processor_time()` — Return a time in seconds according to the processor's tick count. High resolution accurate to 1ns, but will produce inconsistent results if the thread switches processors and is thus intended for accurate timing of short (<1s) events.

6.8 Mesh Topology Functions

The following functions manipulate the mesh topology.

def `vertices(e : Mesh) : Set[Vertex]` — all the vertices in the mesh.

def `vertices(e : Vertex) : Set[Vertex]` — all vertices sharing an edge with this vertex.

def `vertices(e : Edge) : Set[Vertex]` — the two vertices on either end of this edge.

def `vertices(e : Face) : Set[Vertex]` — all vertices on the edges of this face.

def `vertices(e : Cell) : Set[Vertex]` — all vertices on the edges of the faces of this cell.

def `cells(e : Mesh) : Set[Cell]` — all the cells in the mesh.

def `cells(e : Vertex) : Set[Cell]` — all cells containgtex vertex `e`.

def `cells(e : Edge) : Set[Cell]` — all cells containing edge `e`.

def `cells(e : Face) : Set[Cell]` — both cells containing face `e`.

def `cells(e : Cell) : Set[Cell]` — all cells that share a face with cell `e`.

def `edges(e : Mesh) : Set[Edge]` — all edges in the mesh.

def `edges(e : Vertex) : Set[Edge]` — all edges containing vertex `e`.

def `edges(e : Face) : Set[Edge]` — all edges in face `e`.

def `edges(e : Cell) : Set[Edge]` — all edges in cell `e`.

def `edgesCCW(e : Face) : Set[Edge]` — all edges in face `e` oriented counter-clockwise around `e` when observed from cell `outside(e)`.

def `edgesCW(e : Face) : Set[Edge]` — all edges in face `e` oriented clockwise around `e` when observed from cell `outside(e)`.

def `faces(e : Mesh) : Set[Face]` — all faces in the mesh.

def `faces(e : Vertex) : Set[Face]` — all faces containing vertex `e`.

def `faces(e : Edge) : Set[Face]` — all faces containing edge `e`.

def `faces(e : Cell) : Set[Face]` — all faces in cell `e`.

def `facesCCW(e : Edge) : Set[Face]` — all faces `f` containing edge `e` oriented such that when viewed from vertex `head(e)` cell `outside(f)` is counter-clockwise from face `f`.

def `facesCW(e : Edge) : Set[Face]` — all faces `f` containing edge `e` oriented such that when viewed from vertex `head(e)` cell `outside(f)` is clockwise from face `f`.

def `head(e : Edge) : Vertex` — vertex that edge `e` points towards.

def `tail(e : Edge) : Vertex` — vertex from which edge `e` points away from.

def `outside(e : Face) : Cell` — cell on face `e`, dual of `head`.

def `inside(e : Face) : Cell` — cell on face `e`, dual of `tail`.

def `flip(e : Edge) : Edge` — Flip the direction of `e`. `head(e) == tail(flip(e))`.

def `flip(e : Face) : Face` — Flip the direction of `e`. `outside(e) == inside(flip(e))`

def `towards(e : Edge,v : Vertex) : Edge` — edge such that `head(e) == v`

def `towards(e : Face,c : Cell) : Face` — face such that `outside(e) == c`

6.9 Control Flow

Liszt supports the follow subset of Scala control-flow statements:

if`(<Boolean>) <T> [`**else** `<T>]` — if statement (can be used as an expression).

while`(<Boolean>) <T>` — while statement, execute until `<T>` is `false`.

return `<T>` — return from the current function. `T` must be the return type of the current function.

`{ <expressions> }` — Block, has the value of its last expression.

for`(x <- <Set[T]>) <stmt>` — for-comprehensions are only allowed for Liszt's set type. No guards or pattern matching are allowed. In Liszt, for-comprehensions do not impose an ordering on how two `<stmt>` blocks will be run in comparison to each other. `<stmt>` blocks may be run in parallel. For a particular set of mesh topology, `<stmt>` will be executed once for each member of the set. for-comprehensions impose additional constraints on the assignment and reduction statements allowed inside them.

6.10 Assignments and Reductions

Variables are allowed to be reassigned using the `=` operator, similar to how field values can be reassigned. We consider the following patterns to be *reductions* where `<op>` is a binary operator that takes two operands of type `T` and returns a type `T`:

```
v = v <op> <T>
v <op>= <T>
field_ident(x) = field_ident(x) op <T>
field_ident(x) op= <T>
```

Reductions always occur atomically, even when in for-comprehensions. However, additional constraints are placed on the objects. If a variable or a field is read inside the dynamic scope of a for-comprehension, it cannot also be written in that for-comprehension. If a variable or field is being written to using reduction operation `<op>` (`<op>` may be an assignment) in the dynamic scope of a for-comprehension, then it cannot be read in the dynamic scope of the for-comprehension, nor may another reduction of `<op2> != <op>` be used in the dynamic scope of the for-comprehension.

7 Implicit Parameters and Views

Liszt does not currently support user-defined implicit conversion and views.

8 Pattern Matching

Liszt does not currently support pattern matching expressions.

9 Top-Level Definitions

At the top-level you can use any standard scala imports and will need to import `Liszt.Language._` and `Liszt.MetaInteger._` for Liszt to see all of its types. Inside Liszt code, no import statements are allowed.

10 *XML Expressions and Patterns*

Liszt does not support XML expressions and patterns.

11 *User-Defined Annotations*

Liszt does not support user-defined annotations.

12 *Scala Standard Library*

Unless specifically mentioned as a valid expression, Liszt does not support arbitrary calls into the Scala Standard Library.