The Tyr Programming Language

Timm Felden

November 2, 2017

Abstract

This document defines Tyr, a research language for type-oriented programming. Type-oriented programming is a paradigm that extends on object-oriented programming. In type-oriented languages, types are first order values like integers and objects. An existing but primitive form of type orientation is the Java reflection API.

Acknowledgements

For Pony! Inherits from Ada, C++, Java, Scala, SKilL.

Contents

Part I
Core Language

To do (1)

1 Introduction

Type-oriented programming (TOP) is a paradigm that states that types are objects. In consequence, it is possible to perform calculations on types like any other calculation. As it is true for objects in object-oriented programming (OOP), types can be copied and may have mutable state. The mutable state of a type can be bounded by static knowledge in the same way as pointer can be restricted to point to objects of a certain type. As such, TOP implies OOP.

Tyr is a programming language created to explore this idea in practice. Tyr as a language is a descendant of C++ and Scala. In order to examine consequences of TOP for resource management, Tyr features manual memory management.

Type-level functions are descendants of C++ templates and Ada generics.

1.1 Milestone 1

The first milestone is the ability to proove the expression:

```
HList(Type) :: LiteralInt :: LiteralString <: HList(Type) :: int :: string
```

In consequence, the validity of method placements should be easily provable. The typing of funcitons should be something along the lines of

```
functionPointer <: pointer
function(T- : Type)(args: HList(Type), r : T) <: functionPointer
```

Note to self: The minimal type theory for this step is $<:, \tau, \forall^{+-}, \top, \bot$. The τ is not strictly required, but it would be pointless to start without type orientation. Top and bottom follow from variant inheritance. Lambdas are not required ever, and are unlikely to appear ever, as they require garbage collection for closures.

Note to self: Ultimately, there will also be a type Function that holds a vtable and, hence, is an Object. This is similar to Int/int, Long/long, etc.

1.2 On OOP and TOP

TOP has no point without OOP for the following reason:

```
type A;
type B;

val x = new (if(phi) A else B)();
```

What could be the type of x if not $A \sqcap B$?

1.3 Guiding Principle in the Design of Tyr

This section will explain several principles influencing the language design of Tyr.

Do not define wrong properties. For instance, literals have a value and, hence, have a single type. As for all types, the type of a literal must be a subtype of whatever the value of the literal is assigned to. Likewise, the `null` pointer is an instance of each pointer type. Therefore, there must be a bottom type of the pointer sublatice with `null` as its single instance.

Flexibility of a language is the source of good library design. It is perfectly sane to write a type, that enriches strings with a postfix operator allowing their execution as a process in a terminal. Also, as a more common example, collections should allow adding objects via +.

It is not the duty of a language to ensure sanity or beauty of source code. If a programmer calls his variable `` `\n` `` he is likely an idiot. On the other hand, he could be writing a test for a source code manipulation program.

There is no point in preventing idiotic behaviour at the cost of flexibility. Also, there is no objective measure of idiotic behaviour. Hence, it should not be prevented by a programming language.

No real language is type safe. It is the purpose of type systems to show obvious problems and to give the programmer an understanding of program behaviour. Hence, a programmer may choose to violate the type system arbitrarily.

For instance, if a language allows programers to use the POSIX mmap related funcitons, he can violate basically any guarantee given by a language.

Weak typing is for weak programmers. A type system is used to ensure that values have the properties a programer wants to make use of. There is no point in deferring this check to runtime other than making weak programmers think their program is not as bad as it is.

Pure languages are for poor programmers. Writing programs is about modifying the state of a computer. The state of a computer is unrelated to mathematics in general.

2 Syntax

The syntax of Tyr is inspired by Scala and SKilL.

2.1 Keywords, Operators and Identifiers

Keywords in Tyr are:

do def else for if return val var while with

Operator Keywords are:

: <: :=

Keywords can in no context be used as identifiers or operators. They can be made available by using literal identifiers.

⟨*int*⟩ ::= ('0'-'9')+

⟨*hex*⟩ ::= ('0'-'0'|'A'-'F'|'a'-'f')+

⟨*Integer*⟩ ::= '-'? ⟨*int*⟩ ('i' ⟨*int*⟩?)?

⟨*HexInteger*⟩ ::= `0x` ⟨*hex*⟩ ('i' ⟨*int*⟩?)?

⟨*long*⟩ ::= '-'? ⟨*int*⟩ 'L'

⟨*Float*⟩ ::= '-'? ⟨*int*⟩? '.' ⟨*int*⟩ (('e'|'E') '-'? ⟨*int*⟩)? ('f' ⟨*int*⟩?)?

⟨*string*⟩ ::= '"' ~['"']* '"'

⟨*Identifier*⟩ ::= '`' ~['`']+ '`'

⟨*Operator*⟩ ::= ???

Figure 1: Literals

Operators and identifiers are can name function declarations that are treated diffirently when it comes to their invocation. There are no other differences. Literal identifiers containing only operator symbols can be matched by equivalent operator invocations in an expression.

Note that identifiers like `type` or `class` can be used as identifiers because they can only appear in situations where other identifiers are illegal.

2.2 Literals

2.3 Grammar

2.3.1 Top Level Structure

To do (2)

2.3.2 Members

2.3.3 Expressions

- es muss simpleExpressions geben, die für typen und einfache ausdrücke verwendet wird - es muss literalExpressions geben, die ausdrücke ohne berechnungen enthalten, wie etwa Typnamen; literalExpressions enthalten keine operatoren, sind also vor dem OperatorBinding-Schritt auswertbar - statements sind expressions, aber keine simple-Expressions - blöcke sind in diesem sinne wie statements - d.h. wird eine expression erwartet kann man einen block als (...) unterbringen, nicht aber als ....

2.4 Examples

```
def foreach (p : BlockParameter(Any), b : Block(void)) {
  while(move()) {
        p = get();
        b();
```

⟨*file*⟩ ::= ⟨*scopeImport*⟩* ⟨*typeDeclaration*⟩+

⟨*scopeImport*⟩ ::= `with` ⟨*simpleExpression*⟩ ';'?

⟨*typeDeclaration*⟩ ::= `type` ⟨*staticTypeDeclaration*⟩
| `class` ⟨*classDeclaration*⟩
| `interface` ⟨*interfaceDeclaration*⟩
| `property` ⟨*propertyDeclaration*⟩

⟨*staticTypeDeclaration*⟩ ::= ⟨*Identifier*⟩ ⟨*typeExtension*⟩? ⟨*typeBody*⟩

⟨*classDeclaration*⟩ ::= ⟨*Identifier*⟩ ⟨*typeExtension*⟩? ⟨*typeBody*⟩

⟨*interfaceDeclaration*⟩ ::= ⟨*Identifier*⟩ ⟨*typeExtension*⟩? ⟨*typeBody*⟩

⟨*propertyDeclaration*⟩ ::= ⟨*Identifier*⟩ ⟨*typeBody*⟩

Figure 2: Top Level Structure

```
  }
}
def forall (p : BlockParameter(Any), b : Block(bool)) :
    bool = {
  while(move()) {
    p = get();
    if(!b())
      return false;
  }
  return true;
}

test "usage" {
  foreach x do {
    print(x)
  }
  println(forall x do { x != null; })
}
```

3 Semantics

The semantics of Tyr is loosely based on C++ and Scala.

3.1 Types of literal Values

To do (3)

The type of <Integer> is `int`, if no 'i' or no number behind the 'i' is supplied. If a number is supplied, the number will be used as argument for `Integer`(n). If the

resulting Type has a named subtype in `tyr.lang`, the subtype will be chosen. Hence, `0i8` is a `byte` of value 0. Also, 0, `0i` and `0i32` are indistinguishable.

The type of <HexInteger> is UnsignedInteger(n), where n is the number supplied via 'i' defaulting to 32.

The type of <long> is `long`.

The type of <Float> is analogous to <Integer> except that it is based on `FloatingPoint`(n) and defaults to `double`. The type is `float` if a single `f` is supplied. This rule is designed to be compatible with common programming languages.

The type of <string> is `LiteralString`.

<Identifier> literals yield identifiers. An identifier is neither a type nor a value.

3.2 Unescaping of String and Identifier Literals

Tyr uses the same escaping mechanism as Java. Unescaping happens for <string> and <identifier> before further processing.

3.3 Access Paths introduced by local `with`

Any experession that yields a scope can be imported with a with. This import results in an access path to the names imported into the local scope. If that access has a side effect, the effect may be executed at every access to the scope. Function with an implicit `this` parameter start with an implicit expression "`with this;`". The last part of an import expression must be an identifier that is used to create a name in the scope that the import occurs in.

```
with tyr.lang; // legal, makes name lang accessible
with lang.int; // legal, makes tyr.lang.int accessible
    under name int
with 7.toString(); // illegal, last part is a method
with 7.toString().bytes; // legal, will make the bytes
    of the string "7" accessebile under the name bytes;
    note: the call will be reevaluated on each usage
with 7.type; // legal, makes tyr.lang.LiteralInt
    available unter the name type
with "".type; // illegal, because the name type is
    already used in this scope (previous import)
with "".class; // legal
```

3.4 Scopes

As a consequence of assigning a type to any semantic entity, any entity is a scope.

Scope access happens via the dot-Operator ('.'). The static parent relationship of scopes forms a tree. The root is called `_root_`.

If a scope is imported via a `with`-clause, imported scopes are addressed before ascending into parents. If an imported scope is evaluated, parent scopes will be followed until a scope is encountered that does not belong to a user defined entity, i.e. that is a pure namespace. This rule ensures that importing a type makes operations from its super types visible but not from its enclosing scope.

Including inheritance, scopes form a DAG. Imports are not transitive. Therefore, the set of visible entities can be enumerated using recursive descent on the scope DAG. To do (4)

3.4.0.1 Dynamic Lookup The first lookup strategy is called dynamic lookup and is always applied first. It searches the scope of the target entity. The scope of a value is its static type. The scope and its imports are queried. If no result can be found, the super types/classes are queried recursively. If no results can be found, the super interfaces are queried. To do (5)

3.4.0.2 Static Lookup If dynamic lookup failed, a static lookup is performed. This lookup follows the scope tree, including imports done by the scope. If this lookup fails at `_root_`, a single flat lookup in `tyr.lang` is performed.

3.4.1 Namespaces

There is only one namespace for all kinds of named entities. In a lookup, fields count like parameterless methods.

3.4.2 Overloading and Overriding

Find a good rule for near and perfect matches and function application. Note to self: The first call following a member access has to be evaluated before the member access!

There is a function lookup mode that applies if the member access is followed by a call or in the case of operator usage. This mode yields a set of results. The evaluation of a member access in a simple expression yields a distance ordered non-empty sequence of results. Parameterless entities are at the first position. Only one parameterless entity will be presented, as it will shadow any other entity anyway. If the user of the member access expects a single entry, the first element of that sequence is chosen. Otherwise a matching element can be picked. To do (6) To do (7)

```
- 7 == -7;
int.`-`(7) == -7;
int.`-` 7 == -7;  // name resolution error?
```

3.5 Types and Classes

3.5.1 Representation of Objects

Rule: only conversions and instances of types with known representation are legal, i.e. there cannot be a variable/result/parameter of type Any. Note to self: There is a backdoor using type variables? What about p = block : Block[Any]?

3.5.2 Static and Dynamic Type of an Expression

The static type of an expression can be accessed via the field `type`. The dynamic type of an expression can be accessed via the field `class`. The dynamic type is only available for Objects. In a type safe program, the statement $\forall x.x$`.class` $<:$ x`.type` is true. For the sake of symmetry, expressions argument to an access of `type` will be evaluated.

3.6 Tests

Tests are methods in a class that have no surrounding instance, i.e. static methods. They have no name and will not be exported to other modules or to runnable programms. Tests can violate arbitrary visibility rules to simplify test code.

4 Properties

Properties are special non-instantiable types that can be inherited by other types but that will not be inherited by their subtypes. Properties can be used wherever an inheritance operator is legal. Their meaning is given by external tools. The Language defines several properties that are used to influence the behaviour of the Tyr compiler. Properties can be compared to attributes in C++ or annotations in Java.

4.1 Predefined Properties

4.1.1 native

States that the defined entity is implemented by the compiler. If the defined entity is unknown to the compiler, an error will be issued and compilation will be aborted without producing a result.

4.1.1.1 Usage Restrictions

Only types, classes and type members.

4.1.2 CT

Requires, that the value stored in the entity is a compile time constant. For instance, an integer literal or the name of a type can be supplied. This property is used to allow compile-time evaluation of type-level functions.

4.1.2.1 Usage Restrictions

Variables, parameters, values.

4.1.3 covariant

`type T(V : Type <: covariant)` $\Rightarrow \forall u, v \in$ `Type`$.u <: v \Rightarrow T(u) <: T(v)$. This means that results of a type-level function use an inheritance hierarchy that equals their arguments.

4.1.3.1 Usage Restrictions

Type parameters.

4.1.4 contravariant

`type T(V : Type <: covariant)` $\Rightarrow \forall u, v \in$ `Type`$.u <: v \Rightarrow T(v) <: T(u)$. This means that results of a type-level function use an inheritance hierarchy that equals the opposite of their arguments.

4.1.4.1 Usage Restrictions

Type parameters.

4.1.5 generic

States that only one implementation should be used to represent all values for a given type variable, i.e. the type variable is only used for type checking. The representation will use the super type of the type variable.

4.1.5.1 Usage Restrictions Type parameters with a super type with known representation.

4.1.6 public

4.1.7 protected

4.1.8 private

5 Style Guide

This section gives advice how code is to be written in the standard library. It shall be used as a guide to readable Tyr code.

5.1 Orderings

5.1.0.1 Order of Inheritance The order of inheritance should be type/class, interfaces, properties.

5.2 Naming

5.2.0.1 Capitalization Fields, functions, types and properties start with small letters. Classes, interfaces and type variables start with a capital letter.

var/val: fields type var -> type field (in vtable)

defs: def -> virtual static def -> static type (ada non-poly pointer) type def -> type method

Typen: Any (top) void (<: Any) bool Integer int byte long UnsignedInteger FloatingPoint float double pointer

class Object <: pointer String <: Object IterableOnce <: String Iterable <: IterableOnce Option <: Iterable Seq <: Iterable Array <: Iterable

Part II
Compilation

Compilation of Tyr source code is done on a per-module basis. A module is a set ouf source files with a common name. Modules can be compared to JARs in Java or libraries in C and other languages.

6 Module Specification

A module must specify a name, a source directory and a set of modules it depends on. The name of the module must not collide with any transitive dependency. The source

directory contains all sources that will be considered. The implicit scopes created by subdirectories are relative to the source directory. The dependencies must not depend transitively on this module. There is an implicit dependency to `tyr.lang` for all modules except `tyr.lang`.

The name of a module creates corresponding scopes. I.e. a definition inside the source directory of `tyr.lang` is a member of the scope `tyr.lang`. Module names, like scope names, are all lower case. The naming convention for modules is *<organization>.<project>*. Examples are `tyr.lang`, `tyr.collection` or `skill.common`.

7 Elaboration of Compilation Steps

In this section the order and strategy of elaboration of compilation steps will be discussed.

7.1 Overview of Phases

1. Parsing: per pile (text -> AST)
2. Module AST: merge ASTs into a module AST
3. Global and Member aggregation: Collect names and types of visible entities defined in global scopes and type/class/... members.
4. File-Import Resolution: Resolve imports, check for duplicate names in a scope, resolve all-imports. (does not include targets of imports and import expressions)
5. Create evaluation order for the next step; abort and report cycles if any.
6. Name, Type, Import Expression and Operator Resolution: Bind names, infer and check types, turn operator applications into calls.
7. Removal of Unresolved expressions (i.e. check their absence?)
8. Property checks: Abstract, Native
9. Access Checks: Check visibility
10. Removal of Block Parameters: Inline methods taking block parameters.
11. Removal of Imports: Remove all entities introduced for implicit scope access via imports.
12. Shadow Checks: Calculate warnings for shadowed entities
13. IR generation: Create tl-file that can be passed to other compilations or the code generator.

7.2 Evaluation Order for Name Resolution

Path imports require that expressions used in the path can be evaluated. The problem with import paths and operator usage is that new types can be introduced by them and, hence, new dependencies can arise after definining an evaluation order. To do (8)

7.3 File-Import Resolution

Check for duplicates can be performed without knowing the targets of an import expression, because they end with the name that the target will be refered under. Names of declared entities are statically known anyway.

Note This step does not include duplicate names inside of block expressions.

7.4 Removal of Block Parameters

This step may introduce illegal usage of internal state. Hence, access checks must happen before.

Part III
Libraries

8 IO

- Path (VFS) - File (cfile) - MappedFile (mmap) - Console

9 Collections

IterableOnce(T+ : Type) - static def for (p, b) - def foreach (f : LocalLambda[-> T])
Iterator(T+) <: IterableOnce(T+) - empty() - move() : bool - get() - for (p, b) = if(!empty) do
EquivalenceRelation(T : Type) - equals(T, T) : bool - hash(T) : int
MinimalEQR <: EquivalenceRelation - equals := == - hash := .to(Int)
Iterable(T+) <: IterableOnce(T)
Seq(T+) <: Iterable(T)
Array(T) <: Seq(T)
ArrayBuffer(T) <: Seq(T)
StringBuffer <: Seq(String)
List(T) <: Seq(T)
LinkedList(T) <: List(T)
Set(T) <: Seq(T)
HList(T+) <: Seq(T)
HashSet(T : Type <: CT, Eq : Type(EquivalenceRelation) <: CT = MinimalEQR) <: Set
Map(K,V) <: Iterable(struct(K,V))
HashMap(K : Type <: CT, V : Type <: CT, Eq : Type(EquivalenceRelation(K)) <: CT) <: Set

10 Threads

- Thread - ThreadPool - Semaphore - Mutex - Barrier

11 Native

-C method placement -C++ method placement?

Part IV
Appendix

To do…

- ☐ 1 (p. 3): fix wording for scope/package/module/library
- ☐ 2 (p. 5): property mit type body wirkt irgendwie komisch; sollte es nicht eher ohne body, dafür aber mit parametrn/extension sein? sollten properties nicht funktional/tupel sein?
- ☐ 3 (p. 6): sind die ints nicht in wahrheit LiteralInt?
- ☐ 4 (p. 8): if an import ends with an `_all_` then all members of the target entity will be imported
- ☐ 5 (p. 8): specify that super types are linearized in topological order to prevent distant interfaces shadowing closer interfaces; note to self: in order to keep memory complexity of this feature linear, we have to assign numbers in a single pass before looking at expressions; evluation is performed by a heap containing all unevaluated super types; the key in the heap is said number
- ☐ 6 (p. 8): rules?
- ☐ 7 (p. 8): Single result? Implicit conversions?
- ☐ 8 (p. 11): irgendwie habe ich gerade nicht das gefühl, dass das aufgeht; vermutlich muss man die regel aufstellen, dass es keine abhängikeit an das aktuelle modul geben darf, das wäre aber enttäuschend unscharf