The Tyr Programming Language

Timm Felden

August 25, 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

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 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 $A \sqcap B$?

2 Syntax

The syntax of Tyr is inspired by Scala and SKilL.

2.1 Literals

2.2 Grammar

2.2.1 Top Level Structure

2.2.2 Members

2.2.3 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 ....

⟨*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*⟩ ::= '`' ~['`']+ '`'

Figure 1: Literals

⟨*file*⟩ ::= ???

Figure 2: Literals

2.3 Examples

```
def foreach (p : BlockParameter(Any), b : Block(void)) {
  while(move()) {
p = get();
b();
  }
}
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 (1)

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 `with this;` expression.

3.4 Scopes

Scope access happens via the dot-Operator ('.'). The root scope is called `_root_`.

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]?

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)` $\Rightarrow \forall u, v \in \texttt{Type}.u <: v \Rightarrow T(u) <: T(v)$. 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)` $\Rightarrow \forall u, v \in \texttt{Type}.u <: v \Rightarrow T(v) <: T(u)$. 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`.

Part III
Libraries

7 IO

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

8 Collections

IterableOnce(T : Type) - static def for (p, b) - def foreach (f : LocalLambda[-> T])
    Iterator <: IterableOnce - 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 <: IterableOnce
    Seq <: Iterable
    Array <: Seq
    ArrayBuffer <: Seq
    StringBuffer <: Seq(String)
    List <: Seq
    LinkedList <: List
    Set <: Seq
    HashSet(T : Type <: CT, Eq : Type(EquivalenceRelation) <: CT = MinimalEQR) <: Set
    Map <: Iterable
    HashMap(K : Type <: CT, V : Type <: CT, Eq : Type(EquivalenceRelation(K)) <: CT) <: Set

9 Threads

- Thread - ThreadPool - Semaphore - Mutex - Barrier

10 Native

-C method placement -C++ method placement?

Part IV

Appendix

To do...

☐ 1 (p. 5): sind die ints nicht in wahrheit LiteralInt?