The Tyr Programming Language

Timm Felden

August 17, 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!

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.

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

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.

⟨*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

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.

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

modules, source paths, modules scopes, default scopes,

module naming convention: <organization>.<project> tyr.lang tyr.system tyr.collection skill.common

Part III
Libraries

4 IO

5 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
```

Part IV
Appendix

To do...

☐ 1 (p. 2): sind die ints nicht in wahrheit LiteralInt?