The Scaly Book

The Scaly Book

Table of Contents

Part I. Introduction

Chapter 1. The First Program

The most basic expressions in Scaly are *literals*. A literal simply returns a value that is written literally into a program. So our first program is not the famous *Hello World* version in Scaly (which comes later because it is obligatory), instead we write up the shortest non-empty Scaly program imaginable:

```
0
```

Use your favorite text editor and type that lone zero digit into a file, give it a name like `shortest.scaly`, and compile it to an executable. As you might expect, the program does exactly nothing useful. Immediately after startup, it returns a `0` value to the operating system just to indicate that everything went fine. If a program is to return `0` as the last action at the end of it, that `0` literal can be omitted. That's why the shortest Scaly program actually is the empty program which contains nothing and does exactly the same as the above version - just returning `0`. Try it by deleting the zero digit, compiling and running the program.

There are also string literals like `"Hello World!"`, the two bool literals `true` and `false`, and more.

Chapter 2. Hello World

Scaly comes with a standard library which among other stuff contains a function which prints a string to the standard output (which is the output of your terminal, or a debug console, depending on the environment in which your program was started into). The function has the name `print` and accepts a string, returning nothing to the caller. We use it to write up the inevitable *Hello World* program in Scaly:

```
print "Hello World!"
```

When you run that program you will notice that no line break is printed. The reason is that we did not include one in our string literal, and the print function does not print a line break by itself. We correct that by inserting a line break literally:

```
print "Hello World!
"
```

Scaly can contain all kinds of whitespace literally. If you don’t like to span string literals more than one line for readability reasons or want more compact code, you can use an escape sequence as well:

```
print "Hello World!\n"
```

Chapter 3. Command line arguments

To return an `int` value (even an implict `0`) is part of the calling convention for stand-alone programs: Zero or more string arguments are passed to the program, and an integer value is returned which acts as an error code which might be useful in extreme error situations if all other means of diagnosis like logging or tracing fail because they do not work for some pathologic reason which can be indicated by that error code.

The arguments passed to the program are bound to one parameter which is visible at the \ top code level whose name is `arguments`. This parameter is an array of strings, and we can get its `length` field. So our next Scaly program (not much useful either) returns the number of arguments passed at the command line:

```
arguments.length
```

This expression actually consists of two parts: an item identified by the `arguments` identifier and the access of its `length` field via the dot.

Chapter 4. Code Blocks

In all but the simplest programs, you structure your code using blocks. Like in commonplace languages of C descent, a block starts with a left curly brace, followed by zero or more statements, and ends with a right curly brace. An example:

```
{
    let a: int = 2
    let b: int = 3
    a + b
}
```

But there are important differences to C and friends:

First, *code blocks are expressions* - they may return a value. Because of this, if the last statement of a code block is an expression, the value which is returned by that statement is the value which the code block returns. Obviously, our example block returns an `int` whose value is `5`, and therefore is a valid Scaly program.

Second, the statements of a block can be *executed in any order* as far as data dependencies and the pureness of called functions allow. That is the main selling point of Scaly - it automatically schedules parallel and even distributed computation wherever possible.

Chapter 5. Parallel Execution

Not only statements in a block are executed in parallel, but also *function arguments* and *operands*, as long they are *pure* and do not depend on earlier computations in the block.

A computation is called pure if it does not depend on anything else than its input parameters. (With some care, even computations that obtain information from external input can be declared pure by you if needed.)

That said, scheduling parallel computation comes at a cost - tasks have to be created and scheduled for execution by a local worker thread pool, by a GPU, or even by a cluster of remote machines. In the latter case, input data have to be serialized and sent via the network to the remote node, where the data are deserialized. When the computation is done, its results have to be sent back. Last not least the parallel work has to be synchronized.

Therefore, a Scaly implementation has to justify parallel execution at least by some heuristic reasoning, better by profiling a set of reference computation workloads. Scheduling some single floating point additions which might each take nanoseconds or less for parallel execution surely isn’t worth the overhead. Parsing a multitude of source files in contrast can be expected to speed up compiling a program, and performing heavy number crunching needed for fluid mechanics calculations in parallel would a safe bet.

Adjusting the granularity of parallel execution, however, is beyond the Scaly language specification which only states what computations can *potentially* be done in parallel, or to be exact, makes no statement about the order in which independent computations are done.

Chapter 6. Reference

This chapter describes the Scaly Programming Language.

The Scaly Programming Language is defined by the structure and semantics of the statements which can be used to write a Scaly Program.

On the lowest level, a Scaly program consists of a sequence of characters. The lexical structure of the Scaly Programming Language describes how the characters of the program code are combined to form a sequence of tokens and whitespaces.

The structure of the statements is defined by the grammar of the Scaly Programming Language.

The documentation you are reading sometimes refers to the reference implementation of the Scaly Programming Language, but an alternative implementation can choose to do things in another way, or to provide additional functionality like a command line playground which the reference implementation does not have.

Particularly, the standard library which comes with the reference implementation is not part of the Scaly Programming Language in a strict sense. Nevertheless, some elements like basic types, operators, and functions of this standard library like basic types are used in the code samples that are provided.

6.1. Program files

The reference implementation of the Scaly Programming Language, which the documentation you are reading is part of, compiles a Scaly program into an LLVM assembly language [https://llvm.org/docs/LangRef.html] module which can be processed further using the LLVM tools [https://llvm.org/docs/CommandGuide/index.html].

A Scaly program consists of one or more files which contain the program code. The Scaly compiler compiles a Scaly program to a piece of executable code. How this code can be executed depends on the implementation of the Scaly compiler. A file is an object from which a Scaly compiler can read a sequence of characters and parse them into statements. A file must contain zero or more complete statements.

6.2. Lexical structure

On the lowest level, a Scaly program is made up of a sequence of *characters* which are read from one or more files. For the Scaly Programming Language, some characters have a special meaning, which means that they control the forming of the conversion of the character sequence into a sequence of *tokens* and *whitespaces*. These tokens are then parsed into expressions which make up a Scaly program or library.

6.2.1. Whitespaces

Whitespaces are sequences of characters which have no meaning by themselves, apart from being used for separating tokens if no punctuation can be used.

Whitespace character sequences

Outside of comments or strings, the following characters form whitespace character sequences:

- space,
- tab,
- line break, and
- line feed.

Single line comments

Two forward slashes start a single line comment which spans all following characters up to and including a line break.

```
// This is a single line comment.
```

Multiple line comments

A forward slash followed immediately by an asterisk starts a multiple line comment which spans all following characters up to and including an asterisk and a forward slash immediately following.

```
/* This is a multi-line comment.
Continued comment text
The comment ends now:*/
```

Multiple line comments can be nested:

```
/* This comment is /*nested*/.*/
```

6.2.2. Tokens

`Tokens` form the building blocks of expressions. The syntactic grammar of the Scaly Programming Language is expressed in terms of tokens. Tokens can be

- literals,
- punctuation,
- keywords, and
- identifiers.

6.2.3. Literals

Literals are used to write constant values directly into the program.

A numeric literal starts with a digit, followed by zero or more digits, optionally one decimal point, optional fraction digits, an optional exponent consisting of the exponent character (`E` or `e`) and exponent digits.

If the first two characters are `0x`, the literal is a hexadecimal literal. The digits that may follow may include the characters `a`, `b`, `c`, `d`, `e`, `f`, `A`, `B`, `C`, `D`, `E`, and `F`. No decimal point or exponent is allowed for hexadecimal literals.

Some lexically valid numeric literals:

```
42
1.
0.22e4567
1E6
0xFB04
0x123abc
```

(A minus sign is not part of a number literal. It is typically implemented as a unary operator.)

String literals start with a double quote and end with a double quote:

```
"This is a string"
```

All white space characters can be directly embedded in strings:

```
"A string with a
line break"
```

Tabs, carriage returns, line feeds, and NUL characters can be escaped by a back slash and the characters `t`, `r`, `n`, `0` respectively. The back slash escapes itself, and the double quote is escaped by a back slash as well.

```
"A line feed\n, an \"escaped\" string, an escaped \\ backslash,
a \ttab, and a carriage \rreturn."
```

Character literals start with a single quote, continue with the character whose value is to be written, and end with a single quote:

```
'a'
'0' // The zero digit character
```

The characters that can be escaped in string literals, and the single quote must be escaped in a character literal.

```
'\"'
'\''
'\0' // The NUL character
```

6.2.4. Punctuation and Keywords

Punctuation characters are used (alongside keywords) for building the structure of expressions.

Keywords are used (alongside punctuation characters) for building the structure of expressions.

The complete list of punctuation characters and keywords is contained in the Grammar Reference.

6.2.5. Identifiers

All character combinations which are not white space, literals, punctuation, or keywords, are identifiers. Some examples:

```
WindyShore
foo_bar
baz
```

Identifiers are used as names.

6.2.6. Operator

All combinations of operator characters are operators. The operator characters are `+`, `-`, `*`, `/`, `=`, `%`, `&`, `|`, `^`, `~`, `<`, and `>`. Some examples:

```
+
**
<
>>
```

Operators are used as names like identifiers.

6.3. The Program

The Scaly compiler processes a *program* which contains all code which is to be compiled in a single compiler run to an executable program, a library, or code which is compiled just-in-time (JITted). A program consists of zero or more files.

6.4. Files

A *file* is a single sequence of characters which contains zero or more characters which make up zero or more complete statements.

6.5. Statements

A Scaly program consists of *statements*. Statements are the building blocks of a Scaly program. A statement can perform computation work and either return the result of the computation to the enclosing expression or bind it to an identifier which can be referred to in the current scope.

6.6. Expressions

An expression performs actual computation work and usually returns a value as a result of that work. There are numerous forms of expressions in Scaly like *literals*, *function calls*, *operations*, and many more.

Expressions can be optionally terminated by a semicolon. Line breaks are not significant for expression termination.

6.6.1. Literal Values

The most basic expressions in Scaly are *literal values*. A literal value expression evaluates to the value that is written literally into a program.

```
1.602E-19
"baz"
'a'
```

There is no such thing as a boolean literal as a part of the language. Boolean constants can be defined by a runtime library.

6.6.2. Object expressions

An object expression is an expression which combines zero or more expressions, the so-called components, to an object. An object is a combination of data which are used together.

```
() // The empty object which contains no components.
(42,"The Answer") // Contains the number 42 and a string
(()) // A non-empty object which contains the empty object as its only component.
```

An object consisting of a single component is semantially equivalent to the component contained by that object:

```
(5) // => 5
```

The components of an object can be accessed by its index, starting with 1 at its first component. The component index must be known at compile time, it cannot be computed.

```
(7).1 // => 7
(1, (2, (3, (4)))).2.2.2 // => 4
```

The components of an object can be given a name which can be used to access them:

```
(brand = "IFA", model = "F9", year = 1952).year // => 1952
```

6.6.3. Array literals

An array literal combines zero or more expressions of the same type:

```
[2, 3, 5, 7] // An array with four components
```

A component of an array can be accessed by appending brackets which contain the index. The index is zero-based.

```
[1, 2][1] // => 2
```

6.6.4. Declarations

An declaration evaluates an expression and binds the value which was returned by that expression to a constant or variable:

```
let a = 2
// b cannot be used here
let b = 3
a // 2
var c = b
b // 3
```

The constant or variable can be used in every expression which follows its declaration in the current scope. A scope is either the global scope or the scope of a block.

6.6.5. Blocks and scopes

A block is an expression which combines zero or more expressions in a local scope. The last expression of the block is returned.

```
{ 99 } // => 99
```

A scope gives a block a name:

```
scope A {
  let b = 2
}
```

From a scope, a constant can be used after the scope declaration.

```
A.b // => 2
```

6.6.6. Function Expressions

A function expression evaluates to a function value. It consists of the `function` keyword, an object literal, and a block.

```
function (a) { a } // => function(a){a}
```

6.6.7. Function calls

A function (an expression which evaluates to a function) can be called by combining it with an object to be used as an input to the function:

```
let getItself = function (a) { a };
getItself(2) // => 2
```

6.6.8. Operator Expressions

For the Scaly programming language, an operator is a function which receives two objects as input. An operator expression consists of the `operator` keyword, two object literals, and a block.

```
operator (a) (b) { (a, b) } // => operator(a)(b){(a,b)}
```

Scaly knows no binary operator precedence, execution is left to right:

```
2 + 3 * 4 // => 20
```

Operator precedence can be done by putting binary operation into parentheses (which are technically object expressions with one component which simply expose the containing operation after evaluation):

```
2 + (3 * 4) // => 14
```

6.6.9. Operator Calls

An operator can be called by combining an object expression with the operator and a second object expression. The following example declares the >< operator which combines two expressions to an object:

```
let >< = operator (a) (b) { (a,b) }
2 >< 3 // => (2,3)
```

6.6.10. Array Operators

A special variant of the operator expression is one that combines an array literal with a block. It can combine multiple operads using one operation call.

```
let with = operator [a] { (a[0], a[1], a[2], a[3]) }
1 with 2 with 3 with 4 with 5 // => (1,2,3,4,5)
```

The order of evaluation of the operator arguments is not specified.

6.6.11. Binding and Assignment Expressions

Binding expressions bind an expression to a pattern. A pattern is commonly an identifier expression which is a name of the object to which the expression is bound. Binding expressions can be constant (using `let`), variable (using `mutable`), or inferred (using `var`):

```
var a 2
var b 3
var c a + b
c // => 5
```

Writing an equals sign after let, mutable, or var is not required by the language, but since the standard library provides the unary = operator, the binding expressions can use them. Together with optional semicolons, the code might be more readable:

```
var a = 2;
var b = 3;
var c = a + b;
c // => 5
```

An object bound to a `mutable` name can be altered by setting it to a new value using the assignment expression:

```
mutable d 6
set d d + 1
d // => 7
```

6.6.12. Function and Operator Declarations

Instead of declaring a function or using `let`, a shorter and more syntax can be used:

```
function getItself(a) { a }
operator >< (a)(b) { (a, b) }
operator with [a] { (a[0], a[1], a[2], a[3]) }
```

6.6.13. External function declarations

External functions are functions that are provided by the runtime environment. If external functions are to be made accessible from a Scaly program, they must be declared using the following syntax:

```
external _fopen(filename: pointer, mode: pointer): pointer
```

```
external _fclose(file: pointer)
```

6.6.14. Intrinsic function declarations

Intrinsic functions are functions that are provided by the compiler infrastructure. They must be declared using the following syntax:

```
intrinsic function llvm.sin.f64(value: double): double
```

6.7. The Scaly Type System

Types describe kinds of data which can be processed by a Scaly program. Most importantly, a type determines which values a variable or constant can have, and the compiler chooses a representation of the data on the hardware on which the code of a Scaly program runs. There are primitive data types like the void object, functions, and operators, integral and floating point numbers, enumerations, bit masks, characters, pointers,and complex data types like objects, arrays, and variants which combine primitive data types.

6.7.1. The Void data type

The void data type has no value. It can be used as an object that has no data, and it can be written as an empty object expression:

```
()
```

They are used as empty input or output of functions or operators or as an option of variant data types.

6.7.2. Functions

Functions are objects that contain executable code which can be called, and which receive one input object (which can be void) and return one output object (which can be void as well). The type of a function is uniquely identified by its signature, i.e., the types of its input and output. Two functions which have the same signature are of the same type.

6.7.3. Operators

Operators are objects that contain executable code which can be called, and which receive two input objects (which can be void) and return one output object (which can be void as well). The type of an operator is uniquely identified by its signature, i.e., the types of its two input objects and its output object. Two operators which have the same signature are of the same type.

6.7.4. Pointers

The `pointer` type represents an address in the address space of the machine. It can point to any kind of data, and the type or semantics of the data are not defined. A pointer can be converted to any data item without any runtime or compile time checks. The author of the code is responsible to guarantee that any instance of a pointer which is used carrys a valid address, and that the pointer is only converted to a constant or variable of the type of data which lives at that address.

Pointers should only be used when external functions are called by Scaly code or by Scaly functions that are called from external code.

6.7.5. Integer types

The standard library which comes with a Scaly compiler may define integer types which are defined by their bit width and the presence or absence of a sign. These types typically define type conversion and other functions.

Numeric literals written into a Scaly program have no type by themselves. The type of the value that is generated by the compiler is either inferred from the usage of the literal, and if that is not possible, the smallest possible integer type is assumed if the literal is an integer.

6.7.6. Floating point types

The standard library which comes with a Scaly compiler may define floting point types which are defined by their bit width. These types typically define type conversion and other functions.

Numeric literals written into a Scaly program which do not represent an integral number are given a type that is inferred from the usage of that literal. If this is not possible, a floating point type is assumed that the compiler sees fit for the runtime environment for which the compiler was provided.

6.7.7. The character type

The `char` type represents all possible character values which are possible for the runtime environment for which the Scaly compiler is implemented. The Scaly Programming Language makes no assumptions about the storage format of the character value.

The standard library which comes with a Scaly compiler may define utility functions for conversion and other tasks.

6.7.8. The array type

An array is a sequence of objects of the same type in memory. The type of the array is defined by its length and the type of the objects that are contained in the array.

An array can have either a fixed size if the length of the array is known at compile time, or a variable size if the length of the array is not known at compile time.

In the latter case, the count of the objects is stored as a packed integer in front of the sequence of the objects in memory.

The packed integer format stores stores a number to be encoded as a sequence of digits to the base of 127. Each digit is stored in one byte, the lowest digit coming first. The highest digit has its highest bit set to 0 which signals the end of the sequence of digits of the number. The highest bit of the lower digits is set to 1. The lower 7 bits of all bytes encode the value of the digit as an unsigned byte number.

Thus, array lengths from 0 to 127 are encoded in just one byte, and lengths from 128 to 16128 are encoded in two bytes, lengths from 16128 to about 2 millions take three bytes and so on.

6.7.9. The string type

The `string` type is technically an array of bytes. This means that the length of the string which is stored along with the bytes is the length of the byte sequence which represents the string and *not* the count of characters which the string contains. The difference to a byte array is the fact that the runtime library can interpret the stored byte sequence as a stream of characters and, based on that interpretation, can implement string access and manipulation functionality.

String constants in memory are stored in the UTF-8 [https://en.wikipedia.org/wiki/UTF-8] format.

6.8. The Scaly standard library

This section describes the types of the standard library that comes with the reference implementation of the Scaly Programming Language. You can use the Scaly Programming Language without the standard library, and provide your own basic types.

6.8.1. Boolean Values

There is one boolean value type, `bool`. It can have two values, `false` and `true`.

6.8.2. Integral number types

An integral number type is determined by the range of integral numbers which the number can take on. Scaly defines a set of integral number types which are characterized by their bit width, and whether they carry a sign or not:

- `byte` and `sbyte` (8 bits unsigned and signed)
- `short` and `ushort` (16 bits signed and unsigned)
- `int` and `uint` (32 bits signed and unsigned)
- `long` and `ulong` (64 bits signed and unsigned)

6.8.3. Characters

The `char` type represents exactly one character. A character is uniquely identified by its code point which is a non-negative integral number. Which code points are valid and the semantics of the individual characters are determined by the character set used by the operating system and/or the runtime library.

6.9. Grammar

The grammar of the expressions of the Scaly Programming Language is defined as an SGML [https://en.wikipedia.org/wiki/Standard_Generalized_Markup_Language] document.

The reference implementation of the Scaly Programming Language actually generates its complete parser and all AST classes directly out of the grammar description given in the follwoing sections.

6.9.1. Metagrammar

Since SGML itself is a meta-language, the language in which the grammar is formulated is expressed as the following SGML DTD which is explained below:

```
<!ELEMENT grammar - - (syntax+, keyword*, punctuation*)>

<!ELEMENT syntax - O (content)*>
<!ATTLIST syntax
  id          ID                                                    #REQUIRED
  abstract    (abstract|concrete)                                   concrete
  base        IDREF                                                 #IMPLIED
  multiple    (multiple|single)                                     single
  top         (top|nontop)                                          nontop
  program     (program|nonprogram)                                  nonprogram
>

<!ELEMENT content - O EMPTY>
<!ATTLIST content
  type        (syntax|keyword|punctuation|identifier|literal|eof) syntax
  link        IDREF                                                 #IMPLIED
  property    CDATA                                                 #IMPLIED
  multiple    (multiple|single)                                     single
  optional    (optional|required)                                   required
>

<!ELEMENT keyword - O EMPTY>
<!ATTLIST keyword
  id          ID                                                    #REQUIRED
>

<!ELEMENT punctuation - O EMPTY>
<!ATTLIST punctuation
  id          ID                                                    #REQUIRED
  value       CDATA                                                 #REQUIRED
>
```

A `grammar` contains at least one `syntax` rule, zero or more `keyword` elementss, and zero or more `punctuation` elements.

A `syntax` contains zero or more `content` elements.

A `syntax` rule can be `abstract` or `concrete`, with `concrete` as the default. An *abstract* syntax rule is a superset of other syntax rules. An example for an abstract syntax is an `Expression`. As a convention, an abstract syntax contains only links to other syntax rules which indicate what the syntax can be. An `Expression`, for instance, can be a `SimpleExpression` (being abstract itself), or a `Block` (which is concrete), or one of a number of other expressions.

A concrete syntax rule contains content elements which describe the contents of that syntax. A `Block`, for instance, contains a `leftCurly` punctuation, `multiple` `Expression` elements, and a `rightCurly` punctuation. Other content can be an `identifier`, a `keyword`, a `literal`, or an `eof` (the latter one signals the end of the file.)

A concrete syntax rule which is an instance of an abstract syntax rule, needs to indicate its `base` syntax rule.

The top-level syntax rule of a module needs the `top` attribute.

The root syntax rule of the grammar needs the `program` attribute.

A `content` item can `link` to a `syntax`, `keyword`, or `punctuation` element. The `property` attribute is the name of the syntax member variable name in the AST. If a `content` links to a `syntax`, the `multiple` attribute indicates that this syntax can occur multiple times in that context, and the `optional` attribute indicates that this syntax is optional in that context.

The `keyword` has its value as its `id`.

The `punctuation` has its value in the `value` attribute.

6.9.2. Scaly Grammar

Below the grammar of the Scaly programming language is defined in terms of the meta grammar given in the previous section. Please note that a character sequence complying to the grammar is not necessarily a valid Scaly program. All valid Scaly programs, howver, comply with this grammar. The semantic requirements for the expressions to form a valid Scaly program are described in the Program section and the sections that follow.

```
<!DOCTYPE grammar SYSTEM "grammar.dtd">

<grammar>

<syntax id = Program                                        program
    <content identifier                                     property = name
    <content link = File                  multiple          property = files

<syntax id = File                                           top
    <content link = Segment               multiple optional property = statements

<syntax id = Segment                      multiple
    <content link = Statement                               property = Step
    <content link = semicolon                       optional

<syntax id = Block                                          base = PrimaryExpression
    <content link = leftCurly
    <content link = Statement             multiple          property = statements
    <content link = rightCurly

<syntax id = Statement                    multiple abstract
    <content link = Using
    <content link = Declaration
    <content link = Expression
    <content link = Set
    <content link = Break
    <content link = Continue
    <content link = Return
    <content link = Throw

<syntax id = Using                                          base = Statement
    <content link = using
    <content link = Path                                    property = path

<syntax id = Declaration                  multiple abstract base = Statement
    <content link = Let
    <content link = Mutable
    <content link = Var
    <content link = Thread
```

```
    <content link = Class
    <content link = Constructor
    <content link = Method
    <content link = Function

<syntax id = Let                                          base = Declaration
    <content link = let
    <content link = Binding                               property = binding

<syntax id = Mutable                                      base = Declaration
    <content link = mutable
    <content link = Binding                               property = binding

<syntax id = Var                                          base = Declaration
    <content link = var
    <content link = Binding                               property = binding

<syntax id = Thread                                       base = Declaration
    <content link = thread
    <content link = Binding                               property = binding

<syntax id = Binding
    <content link = Pattern                               property = pattern
    <content link = TypeAnnotation                optional property = typeAnnotation
    <content link = Expression            multiple          property = expressions

<syntax id = Pattern                                abstract
     <content link = WildcardPattern
     <content link = IdentifierPattern
     <content link = ExpressionPattern

<syntax id = IdentifierPattern                            base = Pattern
    <content link = Path                                  property = path
    <content link = TypeAnnotation                optional property = annotationForType

<syntax id = WildcardPattern                              base = Pattern
    <content link = underscore

<syntax id = ExpressionPattern                            base = Pattern
    <content link = Expression                            property = expression

<syntax id = Expression                   multiple        base = Statement
    <content link = PrimaryExpression                     property = primary
    <content link = Postfix               multiple optional property = postfixes

<syntax id = PrimaryExpression            multiple abstract
    <content link = Name
    <content link = Constant
    <content link = If
    <content link = Switch
    <content link = For
    <content link = While
    <content link = Do
    <content link = This
    <content link = ObjectExpression
    <content link = Block
    <content link = SizeOf

<syntax id = Name                                   base = PrimaryExpression
    <content link = Path                                  property = path
    <content link = GenericArguments              optional property = generics
    <content link = LifeTime                      optional property = lifeTime
```

```
<syntax id = Constant                                  base = PrimaryExpression
    <content literal                                          property = literal

<syntax id = If                                           base = PrimaryExpression
    <content link = if
    <content link = leftParen
    <content link = Expression            multiple        property = condition
    <content link = rightParen
    <content link = Block                                 property = consequent
    <content link = Else                         optional property = elseClause

<syntax id = Else
    <content link = else
    <content link = Block                                 property = alternative

<syntax id = Switch                                       base = PrimaryExpression
    <content link = switch
    <content link = leftParen
    <content link = Expression            multiple        property = condition
    <content link = rightParen
    <content link = leftCurly
    <content link = SwitchCase            multiple        property = cases
    <content link = rightCurly

<syntax id = SwitchCase                   multiple
    <content link = CaseLabel                             property = label
    <content link = Block                                 property = content

<syntax id = CaseLabel                           abstract
    <content link = ItemCaseLabel
    <content link = DefaultCaseLabel

<syntax id = ItemCaseLabel                                base = CaseLabel
    <content link = case
    <content link = Pattern                               property = pattern
    <content link = CaseItem              multiple optional property = additionalPatterns

<syntax id = DefaultCaseLabel                             base = CaseLabel
    <content link = default

<syntax id = CaseItem                     multiple
    <content link = comma
    <content link = Pattern                               property = pattern

<syntax id = For                                          base = PrimaryExpression
    <content link = for
    <content link = leftParen
    <content identifier                                   property = index
    <content link = TypeAnnotation                optional property = typeAnnotation
    <content link = in
    <content link = Expression            multiple        property = expression
    <content link = rightParen
    <content link = Block                                 property = code

<syntax id = While                                        base = PrimaryExpression
    <content link = while
    <content link = leftParen
    <content link = Expression            multiple        property = condition
    <content link = rightParen
    <content link = Block                                 property = code

<syntax id = Do                                           base = PrimaryExpression
```

```
    <content link = do
    <content link = Block                                   property = code
    <content link = while
    <content link = leftParen
    <content link = Expression          multiple            property = condition
    <content link = rightParen

<syntax id = This                                           base = PrimaryExpression
    <content link = this

<syntax id = Postfix                    multiple abstract
    <content link = Catch
    <content link = MemberAccess
    <content link = Subscript
    <content link = As
    <content link = Is
    <content link = Unwrap

<syntax id = Catch                                          base = Postfix
    <content link = catch
    <content link = CatchPattern                            property = typeSpec
    <content link = Expression                  optional    property = handler

<syntax id = CatchPattern                       abstract
    <content link = WildCardCatchPattern
    <content link = NameCatchPattern

<syntax id = WildCardCatchPattern                           base = CatchPattern
    <content link = WildcardPattern                         property = pattern

<syntax id = NameCatchPattern                               base = CatchPattern
    <content link = Name                        optional    property = member
    <content link = leftParen
    <content identifier                         optional    property = errorName
    <content link = rightParen

<syntax id = MemberAccess                                   base = Postfix
    <content link = dot
    <content identifier                                     property = member

<syntax id = Subscript                                      base = Postfix
    <content link = leftBracket
    <content link = Expression          multiple optional   property = firstItems
    <content link = ObjectItem          multiple optional   property = additionalItemses
    <content link = rightBracket

<syntax id = As                                             base = Postfix
    <content link = as
    <content link = Type                                    property = typeSpec

<syntax id = Is                                             base = Postfix
    <content link = is
    <content link = Type                                    property = typeSpec

<syntax id = Unwrap                                         base = Postfix
    <content link = exclamation

<syntax id = ObjectExpression                               base = PrimaryExpression
    <content link = leftParen
    <content link = Expression          multiple optional   property = firstItems
    <content link = ObjectItem          multiple optional   property = additionalItemses
    <content link = rightParen
```

```
<syntax id = ObjectItem                  multiple
    <content link = comma
    <content link = Expression           multiple optional property = expression

<syntax id = SizeOf                                        base = PrimaryExpression
    <content link = sizeof
    <content link = Type                                   property = typeSpec

<syntax id = Set                                           base = Statement
    <content link = set
    <content link = Expression           multiple          property = lValue
    <content link = colon
    <content link = Expression           multiple          property = rValue

<syntax id = Break                                         base = Statement
    <content link = break
    <content link = Expression           multiple          property = lValue

<syntax id = Continue                                      base = Statement
    <content link = continue

<syntax id = Return                                        base = Statement
    <content link = return
    <content link = Expression           multiple          property = expression

<syntax id = Throw                                         base = Statement
    <content link = throw
    <content link = Expression           multiple          property = expression

<syntax id = Class                                         base = Declaration
    <content link = class
    <content link = Path                                   property = path
    <content link = GenericParameters              optional property = generics
    <content link = Object                         optional property = contents
    <content link = Extends                        optional property = baseClass
    <content link = Expression                     optional property = body

<syntax id = Path
    <content identifier                                    property = name
    <content link = Extension            multiple optional property = extensions

<syntax id = Extension                   multiple
    <content link = dot
    <content identifier                                    property = name

<syntax id = GenericParameters
    <content link = leftBracket
    <content identifier                                    property = name
    <content link = GenericParameter     multiple optional property = additionalGeneric
    <content link = rightBracket

<syntax id = GenericParameter            multiple
    <content link = comma
    <content identifier                                    property = name

<syntax id = Extends
    <content link = extends
    <content link = Path                                   property = path

<syntax id = Object
    <content link = leftParen
    <content link = Component            multiple optional property = components
```

```
    <content link = rightParen

<syntax id = Component                    multiple
    <content identifier                                    property = name
    <content link = TypeAnnotation                optional property = typeAnnotation
    <content link = comma                         optional

<syntax id = Constructor                                   base = Declaration
    <content link = constructor
    <content link = Object                        optional property = input
    <content link = Block                                  property = body

<syntax id = Method                                        base = Declaration
    <content link = method
    <content link = Procedure                              property = procedure

<syntax id = Function                                      base = Declaration
    <content link = function
    <content link = Procedure                              property = procedure

<syntax id = Procedure
    <content identifier                                    property = name
    <content link = Object                        optional property = input
    <content link = TypeAnnotation                optional property = output
    <content link = Throws                        optional property = throwsClause
    <content link = Block                                  property = body

<syntax id = TypeAnnotation
    <content link = colon
    <content link = Type                                   property = typeSpec

<syntax id = Type
    <content identifier                                    property = name
    <content link = GenericArguments              optional property = generics
    <content link = TypePostfix           multiple optional property = postfixes
    <content link = LifeTime                      optional property = lifeTime

<syntax id = Throws
    <content link = throws
    <content link = Type                                   property = throwsType

<syntax id = GenericArguments
    <content link = leftBracket
    <content link = Type                                   property = typeSpec
    <content link = GenericArgument       multiple optional property = additionalGeneric
    <content link = rightBracket

<syntax id = GenericArgument              multiple
    <content link = comma
    <content link = Type                                   property = typeSpec

<syntax id = TypePostfix                  multiple abstract
    <content link = Optional
    <content link = IndexedType

<syntax id = Optional                                      base = TypePostfix
    <content link = question

<syntax id = IndexedType                                   base = TypePostfix
    <content link = leftBracket
    <content link = Type                          optional property = typeSpec
    <content link = rightBracket
```

```
<syntax id = LifeTime                                abstract
    <content link = Root
    <content link = Local
    <content link = Reference
    <content link = Thrown

<syntax id = Root                                            base = LifeTime
    <content link = dollar

<syntax id = Local                                           base = LifeTime
    <content link = at
    <content identifier                                      property = location

<syntax id = Reference                                       base = LifeTime
    <content link = backtick
    <content literal                                 optional property = age

<syntax id = Thrown                                          base = LifeTime
    <content link = hash

<keyword id = using
<keyword id = let
<keyword id = mutable
<keyword id = var
<keyword id = thread
<keyword id = set
<keyword id = class
<keyword id = extends
<keyword id = constructor
<keyword id = method
<keyword id = function
<keyword id = this
<keyword id = sizeof
<keyword id = catch
<keyword id = throws
<keyword id = as
<keyword id = is
<keyword id = if
<keyword id = else
<keyword id = switch
<keyword id = case
<keyword id = default
<keyword id = for
<keyword id = in
<keyword id = while
<keyword id = do
<keyword id = break
<keyword id = continue
<keyword id = return
<keyword id = throw

<punctuation id = semicolon         value = ";"
<punctuation id = leftCurly         value = "{"
<punctuation id = rightCurly        value = "}"
<punctuation id = leftParen         value = "("
<punctuation id = rightParen        value = ")"
<punctuation id = leftBracket       value = "["
<punctuation id = rightBracket      value = "]"
<punctuation id = dot               value = "."
<punctuation id = comma             value = ","
<punctuation id = colon             value = ":"
```

```
<punctuation id = question          value = "?"
<punctuation id = exclamation       value = "!"
<punctuation id = at                value = "@"
<punctuation id = hash              value = "#"
<punctuation id = dollar            value = "$"
<punctuation id = underscore        value = "_"
<punctuation id = backtick          value = "`"

</grammar>
```

6.10. Classes

Classes are a way of organizing data. Classes can have members — either primitive ones like strings or numbers, or other classes, or arrays of them.

Immutable object items — either declared locally, object members or elements of an array or dictionary — can be assigned to either other immutable objects or fresh objects created with `new`.

Mutable object items (local, members or elements) cannot be assigned to other existing objects. They can only be assigned to fresh objects created with `new`.

In addition to the simple assignment operator `=`, there is also a movement operator `=!` which moves the object from the right hand side expression (which must be mutable, and optional) to the left hand item which must be mutable or variable either.

There is also a swap operator `<=>` which swaps the left hand item with the right hand item. For swapping, the items must both be mutable.

Object arguments are passed by reference to a function. If you pass a mutable object to a function, and the function alters that object, the changes are visible to the caller after the function returned.

Classes have an important characteristic: they are self-contained data sets which means that members of an object cannot point to anything outside the object tree, and an object cannot be pointed to by anything else but the owner (if it is a class member or an array element).

Since mutable objects cannot be assigned to, classes organize their data in a strictly hierarchical way. They can be easily serialized to JSON, XML, or any other text-based or binary hierarchical representation of the data they contain. Since classes are serializable, their data can be transmitted over the network to other nodes if your program runs on a supercomputer, or sent to powerful GPU hardware on your local machine. Thus, programs using classes for passing data scale well in a distributed or heterogeneous environment.

Because classes are so easily mapped to JSON or XML, building web services with Scaly is a breeze - simply design a set of functions using classes for data transfer, and you are done.

6.11. Objects

An object is created by calling a constructor of its class. A constructor returns a new object. This object can then be assigned to an item:

```
mutable p: Foo = new Foo(42)
```

An object can be declared *mutable* or *immutable*. An immutable item is declared with the `let` keyword. Neither the object it references can be changed, nor the item itself:

```
let i: Foo = new Foo(43)
// i.bar = 44 // The object cannot be changed
// i = new Foo(44) // The item cannot be reassigned
```

A mutable object is declared with the `mutable` keyword. A mutable item allows for changing the object to which its reference points, and for reassigning another reference to it:

```
mutable m: Foo = Foo(45)
m.bar = 46 // The object can be changed
m = Foo(47) // The item can be reassigned
```

You can copy an immutable object by assigning it to another immutable object.

```
let a: Foo = i
let b: Foo = v
v = i
```

6.12. Parents

A class can define exactly one parent member. A parent member is recognized by an `@` at its type declaration:

```
class Parent {
    let children: Child[]
}

class Child {
    let parent: Parent@
}
```

If you assign an object with a parent member to a member of a object to contain it, this parent member is automatically set to the containing object. If the child is added to an array member, the parent member of the child points to the object which contains that array member:

```
mutable p: Parent = new Parent()
p.Children.push(new Child()) // The parent property of the new Child points to p
```

A parent member is useful if your algorithm walks a tree up and down, or to implement doubly-linked lists.

A parent members can only be accessed if the object holding it is immutable:

```
if p.length() > 0 {
  let c: Child = p.children[0]
  // let r = c.parent  // Error: Can't access a parent of a mutable object
}
```

6.13. Object Age

In Scaly, *objects live on the stack*, either defined as local items, or owned by or referenced by other objects, arrays, or dictionaries, which in turn live somewhere on the stack directly or indirectly (held by other objects).

If a block is left, the memory of all objects which were created in this block is recycled. Therefore, a reference must not be held by an item that outlives the object it references:

```
let a: Foo&
{
    let b: Foo = Foo()
    // a = &b // If b goes out of scope, a would point to recycled memory
}
```

To make `b` assignable to `a`, its declaration can be moved to the outer block:

```
let a: Foo&
let b: Foo
{
    b = Foo()
    a = &b
}
```

The lifetime of an object is determined by the place where a reference to it is declared. The older an object, the longer it lives. Since older data live at least as long as younger data, it can never happen that references to dead data are accessible.

The age of data depends on where it is declared. Items declared in a local block are younger than items in the enclosing block. Parameters that are passed to a function are older than its local items:

```
function f(a: Foo) {
    let b: Bar = Bar()
    {
        let c: Caf = Caf()
    }
}
```

In this example, `a` is oldest, `b` is younger than `a`, and `c` is youngest.

A reference returned by a function is assumed to be *fresh* by default. This means that the function creates the object (either by calling an object constructor or another function which returns a reference to a fresh object). The caller of such a function then assigns the returned reference to an item whose location determines the age of the object:

```
function g(): Foo {
    return Foo(42) // Fresh object created and returned
}

function h() {
    let k: Foo = g() // The object lives here, accessible via k
}
```

If a function is to return an object which is not fresh, the age of such a returned object must be made explicit by an *age tag* which is written after the type of the return value.

An age tag starts with a single quote (`&`) and continues with digits which form a nonnegative number. Leading zero digits are not allowed. `&0` is a valid age tag, `&42` is a valid age tag as well, whereas `&01` is not a valid age tag.

Since Scaly does not know global mutable data, there must be one or more parameters from which to take the returned reference in some way, age tag numbers are used to express age relations between the parameters of a function. The higher the age tag value is, the younger is the tagged reference:

```
function superiorFoo(fooOld: Foo&1, fooYoung: Foo&2) -> Foo&2 {
    if fooOld.number > fooYoung
        fooOld
    else
        fooYoung
}
```

In this example, the returned refrerence can be taken from any of the two parameters, and so its age must be that of the youngest parameter.

The following example checks assignments for age validity:

```
function bar(mutable foo: Foo&1, mutable bar: Bar&2) {
    bar.foo = foo // Valid because foo is declared older
    // foo.bar = bar // Invalid because bar is younger
}
```

If age tags are omitted, the age of the parameters is irrelevant:

```
function baz(p: Foo, q: Foo) -> bool {
    p.number > q
}
```

The age of a member is assumed to be the same as the age of the object containing it (even though the object it points to might be older). Similarly, the age of an array element is assumed to be that of the array, and the age of a dictionary key or value is assumed to be that of the dictionary.

Part II. Design

Chapter 7. Why Scaly?

It’s a bit hard these days to justify developing a new programming language. And yet, there seems to be a sweet spot for Scaly which wants to be a unique language - a language which enables you to easily develop programs which run in parallel in a heterogeneous and distributed environment with outstanding performance.

In the following, the main design points of Scaly are explained and how Scaly compares to existing programming languages.

If we consider where processor development has gone in the last ten years, we see that the performance of a single processor core had not increased as it used to in the decades before. Instead, the progress made in semiconductor industry was used to increase the number of cores we find in processors used in servers, workstations, laptops, tablets and even smartphones.

Today’s most popular programming languages, however, descend from the C programming language which was not designed from the start with parallel and distributed computing in mind. Even though some of these languages have mitigated some problems which plagued legions of C and C++ programmers, they do not support you by design in writing programs which are free of concurrency bugs.

7.1. Avoiding Data Races and Concurrency Problems

One big problem is that these languages permit code which accesses writable data from more than one thread at the same time. Such code frequently suffers from *data races*. A data race causes your code to depend on timing.

Often, bugs of this kind are likely to reveal themselves not before your code runs in production at your customer’s site or at a server in the cloud runnning your customer’s web app. And to make things worse, the failures are typically unable to be reproduced in your development environment. That’s why you did not find these bugs in the first place while developing and testing your code. And even if you *are* able to reproduce the problem, you might not be able to find the reason by stepping through your code or even by tracing, because data races depend on timing - and the timing of code execution typically is different while debugging code, or when writing trace records.

The only way to get rid of data races is to avoid accessing writable data from more than one thread at the same time. There are two things that can be done about that:

1. Synchronize accesses to writable data

2. Forbid making writable data accessible for more then one thread

Without going into much detail, it can be said that trying to synchronize accesses to writable data can lead to all kinds of locking problems - deadlocks cause two or more of your threads to wait forever, making them fail to succeed in their computation, livelocks make your threads burn your valuable CPU time without making any progress. As with data races, these failures are hard to find, to reproduce and to debug.

Even in the absence of these problems, locking in general is an impediment to performance because it lets your threads wait frequently for locks held by other threads to be released.

A better way is the second - never making writable data accessible to more than one thread. There are two ways to acheiving that goal:

1. Make all data immutable

2. Do not share mutable data among threads

7.1.1. Functional Languages

Programming languages which know only immutable data have been known for decades. Pure functional languages like Haskell [https://www.haskell.org/] do not offer a way to alter (or to mutate) a datum after it has been created.

The price paid for this is pretty high, though. With a pure functional language, you cannot use many coding patterns which are commonplace in the most popular programming languages like variables which can be re-assigned or explicit loops. Your data must be read-only. Loops have to carefully be transformed into recursions (only to end up as loops again on machine code level). Performing even simple I/O tasks requires advanced concepts like monades. This might be the reason why this kind of programming language never gained top popularity.

Two languages, Rust [http://rust-lang.org] and ParaSail [http://parasail-lang.org] have recently emerged which choose the second option - they allow you to use mutable data but ensure that mutable data cannot be accessed by more than one thread at the same time.

7.1.2. Rust

The Rust programming language [http://rust-lang.org], sponsored by the Mozilla foundation, avoids sharing mutable data by using the concept of ownership and borrowing. This concept, after some time needed getting used to it while fighting the borrow checker, works perfectly with data which live on the stack.

There are many problems, however, which are hard to solve relying on data which directly live on the stack alone. Try to write a a compiler that way, for instance. When it comes to more complex data structures like growable trees or directed graphs in general, it turns out that this concept is of limited use. Nodes in such structures typically cannot live on the stack directly. Instead, they have to expressed with helpers like `Box`, `Cell`, or `RefCell` which hold them. These either forbid using the data they hold from more than one location, or make use of hidden sychronization mechanisms which might hamper performance by locking contention.

And what’s more, the Rust standard library allocates objects which cannot live on the stack dynamically allocated from a heap. This in turn means that a mechanism is needed to release the memory they use them when they are no longer needed. All of the mechanisms employed eventually use the underlying allocation library to free each object individually. Since big data structures usually are disposed as a whole, there is no need for de-allocating each and every node of the structure individually. Here, Scaly can do much better, as we will see now.

7.2. Efficient Memory Management

All programming languages enjoying top popularity solve the problem of recycling the memory of objects which are no longer used in one three ways:

1. You are required to release the memory of heap-allocated objects by writing explicit code.

2. You are offered to use reference counting

3. The objects are garbage-collected when they are no longer used.

The first option is the one used traditionally by C (`free`) and C++ (`delete`) and leads notoriously to problems like memory leaks (if you forget to release the memory of your objects when they are no longer used), or worse, memory corruptions (if you release their memory too soon). Especially the latter problem is likely to go undetected until your code is deployed at your customer or at your cloud provider and crash the process which runs your code.

Many programs written in C++ in a modern programming style, or programs written in Rust or Swift [https://developer.apple.com/swift/] make use of the second option. Each object is accompanied by a counter keeping track of the number of references to that object, and if the counter reaches `0`, the object is de-allocated. (`unique_ptr` in C++ and `Box` in Rust could be interpreted as a special case where only one reference is possible.)

Apart from the overhead which is caused by the reference counter which is needed for every shared object, and the fact that special measures need to be taken for getting rid of objects which take part in reference cycles, it is simply inefficient to release each and every object taking part in a data structure which is disposed individually.

That is why the third option, garbage collection, is used by many popular programming languages like Java, C#, and JavaScript. Garbage collection works in the background, detecting objects which are no longer accessible and recycling their memory. All the problems and difficulties with the first two approaches are solved. And, since allocation is typically implemented by essentially advancing a pointer to a heap location, allocating an object is rather fast compared to traditional heap allocation methods which at first have to find a suitable memory location for the object.

Unfortunately, garbage collection introduces two major problems:

1. When a garbage collection occurs, one or more threads are halted once or multiple times until the garbage collection is over. Since garbage collections can take many milliseconds, the responsiveness of your app can be impaired. Garbage collection times of many seconds have been reported.

2. Through a garbage gollection, all objects are traversed for accessibility checks, and heaps need to be compacted to avoid fragmentation which involves shifting surviving objects. This way of churning the whole data (at least that of the youngest generation) defeats caching.

The bad thing about these two phenomena is that they are not visible when dealing with small amounts of data which entirely fit in first or second level caches, and whose data can be garbage-collected in less than a few milliseconds.

Then, when you have bought into the promise of getting memory management for free, you scale up your code into serving hundreds of users simutaneously, allocating gigabytes per second, growing the

heaps of your process way beyond the size of your caches - then you run into the problems described above.

If you are lucky, your customer is ready to buy more server machines, or your cloud business cashflow allows for throwing in more expensive EC2 instances behind your load balancer (hampering your or your customer's profit).

If the problems, however, occur on some backbone service, or if the end-users of your code are online gamers which are annoyed by latencies of a few hundred milliseconds, throwing in more Intel power does not help, and the business which is backed by your code is in trouble.

Fortunately, garbage collection is not the last word here. Scaly uses region-based memory management [https://en.wikipedia.org/wiki/Region-based_memory_management] which avoids all problems of the memory management strategies described above.

The idea of region-based memory managenent is not new, and related concepts, known as memory pools or arenas are used, for instance by the Apache web server. The Rust standard library offers `Arena` and `TypedArena` which you can use. On deletion, however, the whole arena is traversed in order to call `drop` on objects which implement the `Drop` trait. As with a garbage collection, this might defeat caching. Apart from that, arenas are still an unstable feature of Rust's standard library, and whether they will enter the stable state or will be dropped remains to be seen.

Few languages are known where region-based memory management is an integral part of the design and works for you behind the scenes. The Cyclone [https://cyclone.thelanguage.org/] programming language (which is no longer supported) makes use of it, and, very recently, ParaSail.

7.2.1. ParaSail

The ParaSail [http://parasail-lang.org] is a new program language using region based memory memory management, and, in fact, its objectives match most of Scaly's features.

Departing from the traditional patterns comes at a price - and that price, of course, depends on the way you go. Opting for Haskell, as an example, requires abandoning many programming patterns commonly used in mainstream programming languages, and leaning new ways of programming like using recursions and monads. That price is fairly high, and the popularity of Haskell is still far behind that of the mainstream languages. According to Google Trends, it actually stalled in the last years and is now challenged by Rust's rapidly growing popularity.

ParaSail, instead, takes another approach to safe parallel programming. When programming in ParaSail, you can keep an imperative programming style. The price you pay for migrating from a language like Java or C# is essentially the following:

1. You cannot use global data.

2. No pointers or references can be used, they are replaced by expandable objects.

3. You cannot use run-time exceptions.

ParaSail shares the avoidance of traditional run-time exceptions with Scaly, so we can focus on the first two points.

First, ParaSail does not allow global data. All data must be passed as parameters to your function.

Scaly allows global data as long as they are immutable. As an example, program configuration data, command-line arguments, or user context information during a web request can be offered for read-only access. If these data are accessible as immutable global data, they do not have to be passed with the help of parameters to your function.

Second, since ParaSail does not allow you to use explicit references, only strictly hierarchcal data can be expressed directly, as is the case with Scaly's structures. For using graphs in ParaSail, you have to make use of ParaSail's generalized indexing facility. This effectively requires you to implement your graphs on top of numerically indexed arrays, and to manage orphaned graph nodes manually - which does not support you avoiding memory leaks.

In Scaly, in addition to structures you can use classes which are accessed through references. You can build up a mutable graph (your code will not fork in that phase, continuing to be executed by a single thread in respect to that graph), return it as immutable, and then using it from multiple tasks.

As an example, if you build a compiler, your code could parse hundreds of source code files, working in parrallel with multiple tasks on them using mutable classes for building up abstract syntax trees (AST) of the files. Then, the parse trees are bundled to a single immutable AST of the whole program. Then you would perform semantic analysis which could scale up by forking tasks, because the AST data are immutable now. (In fact, the reference implementation of the Scaly programming language is designed that way.)

The bottom line is, with Scaly you do not have to sacrifice programming with references like with ParaSail when you write code targeting a multithreaded environment.

When it comes to heterogeneous and distributed environments, however, things are different. Data have to be serialized and transmitted to a GPU or to peer nodes in a cluster or supercomputer. Scaly's classes and references are no help here, because Scaly's class objects cannot be serialized. But you can still use Scaly's structure objects then which scale well in a heterogeneous or distributed environment.

7.3. Safety Against Program Failures

When you write code in Scaly, you can be sure that your program will not suddenly terminate because of a fatal error which makes continuing impossible. There are only three conditions which cause your code to stop executing:

1. The hardware, the operating system, or unsafe code crashes your process.

2. Your program runs out of memory.

3. Your program causes a stack overflow.

As long as your program lives in a healthy environment, does not call buggy unsafe code, is able to allocate memory and does not exhaust stack space, Scaly guarantees that your program will never stop working unexpectedly. That is a huge advantage over most mainstream programming languages!

C and C++ allow your program bugs to crash your process. C++, Java, C#, JavaScript, and Python programs are terminated if their code fails to catch an exception. Even a program written in a language as new as Rust can panic, i.e., stop working because of a run-time error.

Scaly does not know exeptions or run-time errors. To panic and quit is no option for Scaly. If your code does fails to handle an error that can occur, Scaly won't compile it.

Swift's error handling is a bit advanced. It requires you to handle errors which are raised. Unfortunately, you can circumvent that noble principle by writing `try!` before an expression that might throw, leading to a runtime error. Apart from that, Swift cannot handle exceptions which were raised by Objective-C code and which will be handled as run-time errors.

Of all other languages mentioned so far, ParaSail is the only one which eliminates run-time exceptions completely, replacing them with strong compile-time checking of preconditions.

7.4. Conclusion

To sum up everything that was said so far: Of all programming languages mentioned here, only the approach of ParaSail comes close to Scaly's design principles. And yet, omitting globals and assignable references completely is not necessary as long the globals are immutable and if you accept that code portions which build up data structures using classes do not spawn new tasks as long as the data are visible in a mutable way. Apart from that, ParaSail is still at a very early stage, and whether implementations emerge which deliver on the performance promises still remains to be seen.

All other languages mentioned here do not offer the safety and ease of parallel and distributed programming of Scaly.

Chapter 8. Memory Management

Scaly uses region-based memory management [https://en.wikipedia.org/wiki/Region-based_memory_management] where objects conceptually live on the stack. No heap is used, and therefore, neither garbage collection nor reference counting is required. Objects are accessed via references within well-defined constraints which ensure safe parallel execution of the code.

8.1. Regions

With Scaly's region-based memory management, objects are not allocated from a global heap, but from a region. A region is formed each time program execution enters a block in which at least one fresh object is assigned to a reference which is defined in that block. An example:

```
function f() {
    let a = new A()
    let b = createB()
    {
        let c = new C()
        {
            c.d = new D()
        }
    }
}
```

When a thread enters `f`, the outermost block of that function allocates a region in which two objects are created: one object of type `A` to which the reference `a` points is created with `new`, and then one object of type `B` to which the reference `b` points, is created by a function `createB()` (which might create the object by itself using `new` or call another function which creates the object).

Then, the next block is entered, and a new region is allocated in which an object of type `C` is created and then assigned to the reference `c`.

The innermost block of the function, however, creates no region because no object is assigned to a reference which is defined in that block. The new `D` object is created in `C`'s region instead since it is assigned to `d` which is a member of `d`.

When the block which contains `c` is left, its region is recycled, and both the object to which `c` points and the object to which its member reference `d` points cease to exist.

When the thread leaves `f`, the region which was created by its outermost block is recycled, too, and the objects to which `a` and `b` pointed, vanish as well.

References like `a`, `b`, and `c` which are defined in a block are called *root references*. A root reference cannot be returned by a function because the region in which the object lives to which that reference points is already recycled since its block is left before the function returns.

References like `d` are called *local references*. A local reference points to an object which lives in the region of a containing object as a member, array element, or dictionary key or value. A local reference can only be returned from a function if it is contained in an object which was created before the function was entered.

The bootstrapping compiler which is currently the only (known) implementation of Scaly is written in C++. It uses a special form of the C++ `new` operator which is called *placement new* to create new objects in the appropriate region. The runtime library which comes with the compiler provides the necessary mechanisms for managing the memory in regions.

8.2. Pages

This compiler would compile the above function to a C++ function to something like the following:

```
void f() {
    _Region _region; _Page* _p = _region.get();
    A* a = new(_p) A();
    B* b = createB(_p);
    {
        _Region _region; _Page* _p = _region.get();
        C* c = new(_p) C();
        {
            c->d = new(c->getPage()) D();
        }
    }
}
```

When the function is entered, a `_Region` object is created for its block to provide memory for allocating objects which are rooted in this block. The memory of a region is divided in pages of constant size (typically 4 or 8 kB). Therefore, a `_Page` object `_p` is obtained from that region. Then a new object of type `A` is created in `_p` using the placement variant of `new`.

All classes implement the *placement new* operator by getting memory from the page in a way similar to this code (some error handling removed for brevity):

```
void* Object::operator new(size_t size, _Page* page) {
    return page->allocateObject(size); }
```

Next, the `createB` function is used to create a new `B` object in our current page `_p`. Since `createB` needs to know in which page the object is to be created, it is passed to that function. The function `createB` can use *placement new* to create the object or use another function, passing the pointer to the page to it.

Then, the next block is entered, and a new region is created and a page is obtained from that page which is used to create a new `C` object.

In the innermost block, a `D` object is created in the page of `c`. For that reason, `c` is asked to provide its page for creating the object in. All classes implement a function `_getPage()` for this purpose:

```
_Page* Object::_getPage() {
    return _Page::getPage(this);
}
```

Since pages are always aligned and have a size of a multiple of 2, the page of an address can be calculated easily by setting the relevant lower bits of the address to zero:

```
_Page* _Page::getPage(void* address) {
    return (_Page*) (((intptr_t)address) & ~(intptr_t)(_pageSize - 1));
}
```

8.3. Root Page Allocation

We have seen that each region provides a page upon block entry. The `_Region` class allocates its page out of a large chunk of memory which is thread local:

```
__thread _Page* __CurrentPage = 0;
```

At thread start, before regions kick in, this memory is allocated:

```
posix_memalign((void**)&__CurrentPage, _pageSize, _pageSize * _maxStackPages);
```

The `_Region` class itself actually is only a very small helper. At block entry, it shifts the `__CurrentPage` pointer up and initializes a `_Page` object at this position:

```
_Region::_Region(){
    __CurrentPage = (_Page*)(((char*)__CurrentPage) + _pageSize);
    __CurrentPage->reset();
}
```

At block exit, the extension Pages and exclusive Pages of the page (we will come to this later) are deallocated, and the `__CurrentPage` pointer is shifted down:

```
_Region::~_Region() {
    __CurrentPage->deallocateExtensions();
    __CurrentPage = (_Page*)(((char*)__CurrentPage) - _pageSize);
}
```

This way, the memory area accessed by the `__CurrentPage` pointer during thread execution forms a logical stack, and the pages which live in this area are called *root pages* (as opposed to *extension pages* which we will cover in a later section of this chapter). Allocating a fresh region is done by shifting a thread-local pointer.

8.4. Object Allocation

As we have seen, the `_Region` class itself only shifts the current root page pointer up and down the root page stack as thread execution enters and leaves blocks which declare root references.

Most of the memory management logic is performed by the `_Page` class. This class provides memory for allocating objects and, if required, extends the available memory by allocating a new page if its own memory is exhausted.

A page is a memory area which is aligned to multiples of the page size which is usually 4 or 8 kB. This memory area is controlled by a `_Page` object which lives at the start address of the page and contains (among a few other things) an offset to the next object to be allocated, counted from the page start address:

```
int nextObjectOffset;
```

If the space fits for a particular object to allocate, allocation is done by adding to `nextObjectOffset` and returning the current location. Here a code snippet of the `allocateObject` method of `_Page` which is used by the placement new operator of `Object`:

```
void* location = getNextObject();
```

```
void* nextLocation = align((char*)location + size);
if (nextLocation <= (void*) getNextExclusivePageLocation()) {
    setNextObject(nextLocation);
    return location; }
```

Here are two of the three helpers used (`getNextExclusivePageLocation()` will be explained later):

```
void* _Page::getNextObject() {
    return (char*)this + nextObjectOffset; }

void _Page::setNextObject(void* object) {
    nextObjectOffset = (char*)object - (char*)this; }
```

With this code, allocating an object is done taking the following steps:

1. The `location` of the next object to be allocated is calculated by adding `nextObjectOffset` to the start of the page

2. The upper boundary of the object is calculated by adding the object size to `location` and aligning the result

3. If the upper boundary is within limits, `nextObjectOffset` is calculated, and `location` is returned as the new object address.

Thus, object allocation typically boils down to adding to an offset and checking it.

8.5. Extension pages

An extension page is allocated and used if the current page cannot provide enough memory to allocate an object of the desired size. If the size required exceeds the maximum size which can be stored in a fresh page, an oversized page is allocated directly from the operating system via `posix_memalign` and is registered with the current page as an *exclusive page*. We will discuss exclusive pages in one of the following sections. If the size required does fit in a fresh page, an *extension page* is allocated by the following method of the `_Page` class:

```
_Page* _Page::allocateExtensionPage() {
    _Page* extensionPage = __CurrentTask->getExtensionPage();
    if (!extensionPage)
        return 0;
    extensionPage->reset();
    *getExtensionPageLocation() = extensionPage;
    currentPage = extensionPage;
    return extensionPage;
}
```

The `getExtensionPage()` method of the `_Task` class which is called first, allocates a page out of a thread-local page pool whose start-up size is 4096 pages (one chunk) and which can extend itself by adding more chunks. The pages in a chunk are bit mapped in an allocation map of that chunk, and allocating a page in a chunk is a quick constant-time action.

Then, after the extension page is initialized with *placement new*, its address is stored at a memory location at the very end of the page to be accessed at deallocation time:

```
_Page** _Page::getExtensionPageLocation() {
    return ((_Page**) ((char*) this + _pageSize)) - 1; }
```

Before `allocateExtensionPage()` returns, the extension page location is stored also in the `currentPage` member as a shortcut to a page where allocation space should be available. This member is updated also if a further allocation attempt results in an object which lives in an even newer extension page.

This way, allocation can start from a root page forming a chain of extension pages if many objects are allocated in the region of that root page. When this region is left, the `deallocateExtensions()` method deallocates all extension pages by freeing them in the thread local page pool.

8.6. Exclusive pages

We have seen that a region can grow by extending its root page with an extension page, which can in turn be extended by another extension page, and so on. When thread execution leaves the region, all extension pages are deallocated from the page pool. But while our region is alive, we have seen no way for deleting objects from a region — but that is what we need for `mutable` object data, because otherwise we would leak memory by abandoning objects pointed to by `mutable` references if we set those references to fresh objects.

For this reason, fresh objects which are assigned to `mutable` references, get their own (root) page, a so-called *exclusive page*. An exclusive page can be easily deallocated if its leading object is deleted while its region remains active.

Here are the relevant code lines which allocate and register an exclusive page:

```
_Page* exclusivePage = __CurrentTask->getExtensionPage();
exclusivePage->reset();
*getNextExclusivePageLocation() = exclusivePage;
exclusivePages++;
```

The number of active exclusive pages is stored in the `int` member `exclusivePages` of the page, and the pointer to the next exclusive page address is calculated by the following helper method:

```
_Page** _Page::getNextExclusivePageLocation() {
    return getExtensionPageLocation() - exclusivePages - 1; }
```

Exclusive pages are allocated and initialized in the same way as an extension page, but they are registered not at the very end of the current page but in the next lower addresses. Several exclusive pages can be allocated and registered with a page, while their locations are stored in an area which grows downward from the end of that page (while the space allocated by objects grows from the start of the page).

If an exclusive page is deallocated, it is unregistered with the page holding it by shifting all lower exclusive page addresses up if applicable and decrementing `exclusivePages`.

8.7. Oversized pages

String and array buffers can have sizes that exceed the net space of a freshly allocated page. If we are to store such a thing, we allocate the necessary space directly from the operating system:

```
_Page* page;
posix_memalign((void**)&page, _pageSize, size + sizeof(_Page));
page->reset();
```

```
page->currentPage = nullptr;
*getNextExclusivePageLocation() = page;
exclusivePages++;
return ((char*)page) + sizeof(_Page);
```

Thus, we initialize the memory area as a page, mark it as oversized by setting `currentPage` to `nullptr`, register the page as an exclusive page and return the memory address directly after the page object data.